

Méritos alegados en el CV de la solicitud para
el acceso a los cuerpos docentes universitarios
de Profesor Titular de Universidad. Plaza
2020-02 de la Universidad de Zaragoza

Darío Suárez Gracia

29 de mayo de 2020

Descripción

Este índice incluye todos los justificantes de méritos ordenados según las categorías del Curriculum Vitae Normalizado del candidato.

Si usted desea ver un mérito concreto, por favor haga click sobre la sección en la que esté interesado dentro del índice.

Índice de justificantes por apartado

1 Cargos y actividades desempeñados con anterioridad	3
2 Formación académica recibida	4
2.1 Titulación universitaria	4
2.1.1 Estudios de 1º y 2º ciclo, y antiguos ciclos (Licenciados, Diplomados, Ingenieros Superiores, Ingenieros Técnicos, Arquitectos)	4
2.1.2 Doctorados	6
2.1.3 Otros títulos	9
2.1.4 Conocimiento de idiomas	10
3 Actividad docente	11
3.1 Formación académica impartida	11
3.1.1 Dirección de tesis doctorales y/o proyectos fin de carrera	14
4 Experiencia científica y tecnológica	17
4.1 Grupos/equipos de investigación, desarrollo o innovación	18
4.2 Actividad científica o tecnológica	19

4.2.1 Proyectos de I+D+i financiados en convocatorias competitivas de Administraciones o entidades públicas y privadas	19
4.2.2 Contratos, convenios o proyectos de I+D+i no competitivos con Administraciones o entidades públicas o privadas	38
5 Resultados	42
5.1 Propiedad industrial e intelectual	43
6 Actividades científicas y tecnológicas	390
6.1 Publicaciones, documentos científicos y técnicos	391
6.2 Trabajos presentados en congresos nacionales o internacionales	423
6.3 Gestión de I+D+i y participación en comités científicos	459
6.3.1 Comités científicos, técnicos y/o asesores	460
6.3.2 Organización de actividades de I+D+i	461
6.3.3 Gestión de I+D+i	462
6.3.4 Evaluación y revisión de proyectos y artículos de I+D+i	463
6.4 Otros méritos	474
6.4.1 Estancias en centros de I+D+i públicos o privados	475
6.4.2 Resumen de otros méritos	480

1 Cargos y actividades desempeñados con anterioridad



Qualcomm Technologies, Inc.
5775 Morehouse Drive
San Diego, CA 92121
United States

December 30, 2015

To Whom It May Concern:

This letter is to verify that Dario Suarez Gracia was employed as a regular full-time employee at Qualcomm Technologies, Inc. ("Qualcomm") from October 01, 2012 to September 04, 2015. The job title at the time of termination was Engineer, Staff.

If you have any further questions regarding this matter, please call our Verification of Employment number at (858) 651-1740.

Sincerely,

A handwritten signature in black ink that appears to read "Laura Edwards".

Laura Edwards

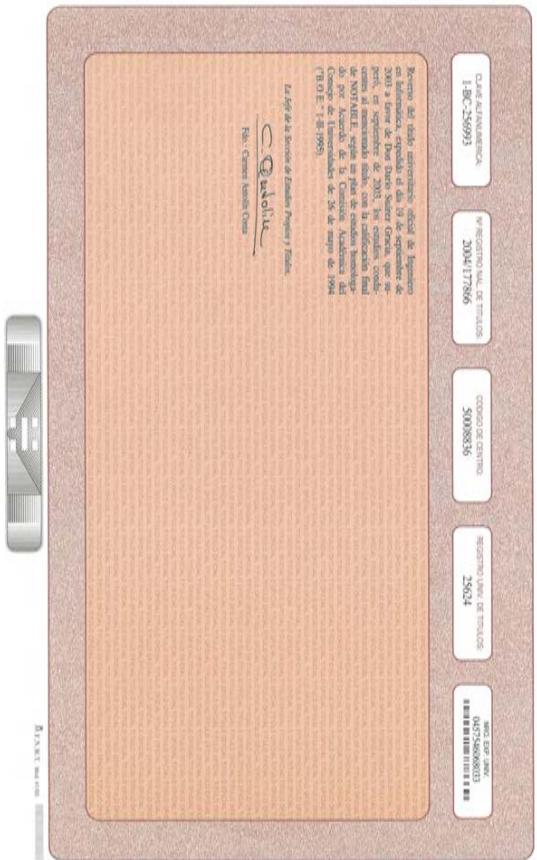
On behalf of Qualcomm Technologies, Inc.

2 Formación académica recibida

2.1 Titulación universitaria

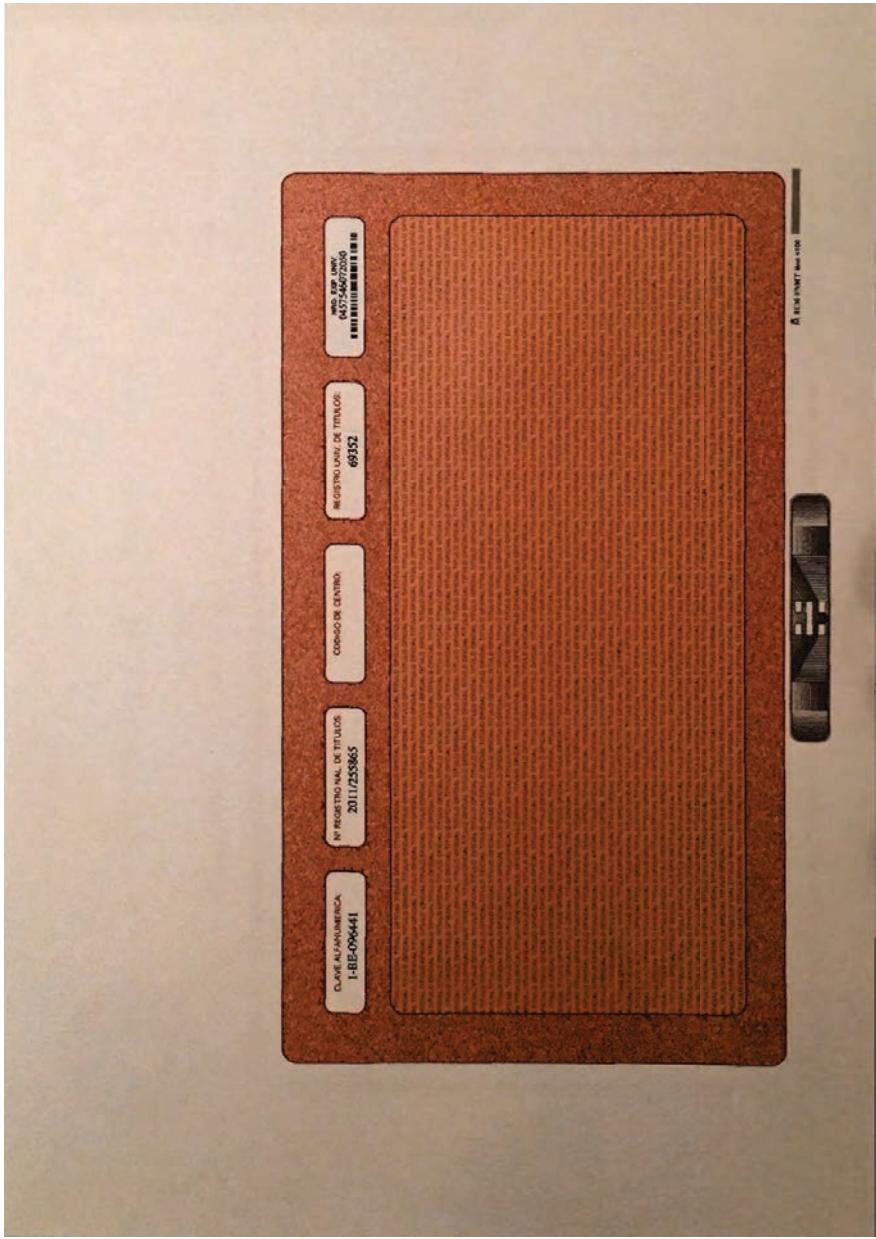
2.1.1 Estudios de 1º y 2º ciclo, y antiguos ciclos (Licenciados, Diplomados, Ingenieros Superiores, Ingenieros Técnicos, Arquitectos)





2.1.2 Doctorados







Universidad
Zaragoza

1542

**DOCTORADO
EUROPEO**

Don José Vela Tejada, Secretario de la Comisión de Doctorado de la Universidad de Zaragoza

CERTIFICA:

Que D. Darío Suárez Gracia ha obtenido la mención de Doctor por la Universidad de Zaragoza (Doctorado Europeo) tras la elaboración, defensa y aprobación de su Tesis Doctoral, en virtud de la normativa aprobada a tal efecto por la Junta de Gobierno de esta Universidad los días 25 y 26 de abril de 1996.

Título de la Tesis: A Tiled Cache Organization.

Directores de la Tesis: Dra. Dña. Teresa Monreal Amal y Dr. D. Víctor Viñals Yúfera

Fecha y lugar de la defensa pública: 11 de noviembre de 2011 en Zaragoza.

Los informes previos han sido elaborados por:

1. Dr. D. Jacobo Torán, Institut für Theoretische Informatik, Universität Ulm (Alemania)
2. Dr. D. Georgios Keramidas, Universidad de Patras (Grecia).

Composición del tribunal que ha juzgado la Tesis:

Presidente: Dr. D. Julio Ramón Beivide Palacio

Secretaria: Dra. Dña. María Villarroya Gaudó

Vocales: Dr. D. Clemente Rodríguez Lafuente, Dr. D. José Ignacio Navarro Más y Dr. D. Giorgos Dimitrakopoulos

La Tesis Doctoral ha sido defendida en idioma español y parcialmente en idioma inglés.

Para la preparación de la Tesis se ha realizado la siguiente estancia:

Del 12 de julio al 15 de octubre de 2009, Foundation for Research and Technology - Hellas (Forth), Institute of Computer Science (Grecia).

Y para que así conste a los efectos oportunos, se expide la presente certificación, en Zaragoza a 21 de noviembre de 2011.

D.P.B.

La Presidente de la Comisión

El Secretario de la Comisión

Fdo.: M^a Pilar Diago Diago

Fdo.: José Vela Tejada



2.1.3 Otros títulos



UNIVERSIDAD DE ZARAGOZA

El Rector de la Universidad de Zaragoza
otorga el presente

DIPLOMA DE ESTUDIOS AVANZADOS

a

Don Darío Suárez Gracia

Ingeniero en Informática
por la Universidad de Zaragoza

por haber superado, conforme a lo previsto en el Real Decreto 778/98 de 30 de abril
(BOE 1 de mayo), la evaluación de los períodos de docencia e investigación
en el **Programa de Doctorado de Ingeniería de Sistemas e Informática**
del Departamento de Informática e Ingeniería de Sistemas,
el día 15 de septiembre de 2005, con la calificación de Sobresaliente,
que le acredita la suficiencia investigadora en el Área de Conocimiento de

Arquitectura y Tecnología de Computadores

De acuerdo con el artículo 6 del citado Real Decreto el presente
Diploma será homologable en todas las universidades españolas.

Zaragoza, 28 de septiembre de 2005

El interesado,

Handwritten signature of Darío Suárez Gracia.

Darío Suárez Gracia

El Rector,

Handwritten signature of Felipe Pérez Calvo.

Felipe Pérez Calvo

El Presidente de la Comisión
de Doctorado,

Handwritten signature of Ángel Escobar Chico.

Ángel Escobar Chico

Reg. 2190

Siglo XXI

2.1.4 Conocimiento de idiomas

3 Actividad docente

3.1 Formación académica impartida



Copia auténtica de documento firmado digitalmente. Puede verificar su autenticidad en <http://www.unizar.es/cv/25d6960c99413a142c0a41d0be30a9>



Secretario General
Universidad
Zaragoza

C E R T I F I C A:

Que D. Darío Suárez Gracia, de acuerdo con la documentación existente en esta Universidad, ha desempeñado la actividad docente que se especifica a continuación,

CURSO ACADÉMICO / CUATR.	PUESTO OCUPADO	ASIGNATURA	PLAN DE ESTUDIOS/TITULACIÓN	CHARÁTER CURSO	HORAS TEÓRICAS	HORAS PRACTICAS	TOTAL HORAS IMPARTIDAS POR CURSO
2008-2009		Arquitectura De Computadores (12015)	Ingeniero en Informática	Troncal 1º e 2º	0 / 60	60	60
2009-2010	Ayudante	Arquitectura De Computadores (12015)	Ingeniero en Informática	Troncal 1º e 2º	0 / 60	60	60
2010-2011	Ayudante	Arquitectura Y Organización De Computadores (32025)	Graduado en Ingeniería Informática	Formación 1º - S2	0 / 60	60	60
2011-2012		Arquitectura Y Organización De Computadores (32025)	Graduado en Ingeniería Informática	Formación 1º - S2	0 / 30	30	60
		Arquitectura Y Organización De Computadores (32025)	Graduado en Ingeniería Informática	Obligatoria 2º	0 / 30	30	60
		Administración De Sistemas (30216)	Graduado en Ingeniería Informática	Obligatoria 2º	60 / 45	105	
2015-2016	Ayudante Doctor	Redes Y Sistemas Distribuidos (62223)	Máster Universitario en Ingeniería Informática	Opcional 1º e 51	22,5 / 7,5	30	206
		Proyecto Hardware (30220)	Graduado en Ingeniería Informática	Obligatoria 3º	0 / 54	54	
		Garantía Y Seguridad (30242)	Graduado en Ingeniería Informática	Opcional 4º e 52	15 / 2	17	
		Administración De Sistemas (30216)	Graduado en Ingeniería Informática	Obligatoria 2º	60 / 45	105	
2016-2017	Contratado Doctor Interno	Proyecto Hardware (30220)	Graduado en Ingeniería Informática	Obligatoria 3º	0 / 54	54	210
		Garantía Y Seguridad (30242)	Graduado en Ingeniería Informática	Opcional 4º e 52	19 / 2	21	
		Redes Y Sistemas Distribuidos (62223)	Máster Universitario en Ingeniería Informática	Opcional 1º e 51	22,5 / 7,5	30	

12



CSV: 25d6960c99413a142c0a41d0be30a9	Organismo: Universidad de Zaragoza	Página: 1 / 2
Firmado electrónicamente por	Cargo o Rol	Fecha
GONZALO LÓPEZ NICOLÁS JUAN GARCÍA BLASCO	Coordinador Programa de Doctorado de Ingeniería de Sistemas e Informática Secretario General Universidad de Zaragoza	03/02/2020 19:09:00 04/02/2020 08:01:00



Copia auténtica de documento firmado digitalmente. Puede verificar su autenticidad en <http://www.dte.unizar.es/csv/250d696d0a99413a142c0a41d0be30a9>

**Secretario General
Universidad
Zaragoza**



CURSO ACADEMICO/ CUATR.	PUESTO OCUPADO	A SIGNATURA	PLAN DE ESTUDIOS/TITULACION	Carácter CURSO	Horas Teoría/Prácticas	TOTAL HORAS IMPARTIDAS	TOTAL HORAS IMPARTIDAS POR CURSO
2017-2018		Administración De Sistemas (30216) Proyecto Hardware (30220) Garantía Y Seguridad (30242) Redes Y Sistemas Distribuidos (62223) Trabajo Fin de Grado (19935) Redes De Computadores (30211) Administración De Sistemas (30216) Proyecto Hardware (30220) Trabajo Fin de Grado (Computación)	Graduado en Ingeniería Informática Graduado en Ingeniería Informática Graduado en Ingeniería Informática Master Universitario en Ingeniería Informática Graduado en Ingeniería Química Graduado en Ingeniería Informática Graduado en Ingeniería Informática Graduado en Ingeniería Informática Graduado en Ingeniería Informática	Obligatoria 2º Obligatoria 3º Obligatoria 4º Obligatoria 1º Obligatoria 2º Obligatoria 3º Obligatoria 2º Obligatoria 3º Obligatoria 2º	45 / 55 0 / 54 15 / 2 22,5 / 7,5 0 / 6 0 / 30 40 / 0 0 / 54 0 / 18	100 54 17 30 6 30 40 54 18	201
2018-2019	Contratado Doctor Internio	Garantía Y Seguridad (30242) Trabajo Fin de Grado (Ingeniería de Computadores) (30269) Trabajo Fin de Grado (Tecnologías de la Información) (31027) Redes Y Sistemas Distribuidos (62223) Trabajo Fin de Máster (62233)	Graduado en Ingeniería Informática Graduado en Ingeniería Informática Graduado en Ingeniería Informática Master Universitario en Ingeniería Informática Master Universitario en Ingeniería Informática	Obligatoria 4º Obligatoria 2º Obligatoria 2º Obligatoria 3º Obligatoria 2º	15 / 2 45 / 52 40 / 0 0 / 12 22,5 / 7,5 0 / 7,5	17 232,5 40 12 30 7,5	

y para que así conste y a los efectos oportunos, expido y firmo el presente certificado en Zaragoza, a 3 de febrero de 2020.

Firmado electrónicamente y con autenticidad contrastable según al artículo 27.3.c) de la Ley 39/2015 por Gonzalo López Nicolás, Profesor Secretario del Departamento de Informática e Ingeniería de Sistemas y Juan García Blasco, Secretario General de la Universidad de Zaragoza

Plaza Basilio Paraíso, 7/50005 Zaragoza
Telf. 97661000
E-mail: SECRETARIOGENERAL@UNIZAR.ES



CSV: 250d696d0a99413a142c0a41d0be30a9	Organismo: Universidad de Zaragoza	Página: 2 / 2
Firmado electrónicamente por	Cargo o Rol	Fecha
GONZALO LÓPEZ NICOLÁS JUAN GARCÍA BLASCO	Coordinador Programa de Doctorado de Ingeniería de Sistemas e Informática Secretario General Universidad de Zaragoza	03/02/2020 19:09:00 04/02/2020 08:01:00

DON ADOLFO FERNANDEZ FERNANDEZ, COMANDANTE DEL CUERPO GENERAL DEL EJÉRCITO DE TIERRA, ESCALA DE OFICIALES, TRANSMISIONES, JEFE DEL NÚCLEO DE LOGÍSTICA DE LA PLANA MAYOR DE DIRECCIÓN DE LA ACADEMIA DE INGENIEROS, DEL QUE ES JEFE INTERINO EL TENIENTE CORONEL DEL CUERPO GENERAL DEL EJÉRCITO DE TIERRA, ESCALA DE OFICIALES, TRANSMISIONES, DON JAIME QUESADA ORDINAS.

CERTIFICA: Que, D. Dario Suárez Gracia (18443694V), ha sido profesor del I Curso Avanzado de Ciberdefensa, convocado por Resolución 220/091467/16 (BOD. 123) de fecha 24 de junio de 2016, desarrollado en la Academia de Ingenieros del Ejército de Tierra (Hoyo de Manzanares) bajo la dirección de la Escuela Superior de las Fuerzas Armadas, en el cual, ha impartido un total de 10 horas lectivas del módulo de Seguridad en el software.

Y para que conste, firma el presente, con el visto bueno del Señor Teniente Coronel Jefe Interino de la PLMD, en Hoyo de Manzanares (Madrid), a quince de diciembre de dos mil diecisésis.



Vº, Bº.
EL TCOL. JEFE INTERINO DE LA PLMD



D. JAIME QUESADA ORDINAS

3.1.1 Dirección de tesis doctorales y/o proyectos fin de carrera



TESIS DOCTORALES - TESEO

Título: EXPLOITING NATURAL ON-CHIP REDUNDANCY FOR ENERGY EFFICIENT MEMORY AND COMPUTING

Nombre: Ferrerón Labari, Alexandra

Universidad: Universidad de Zaragoza

Departamento: Informática e ingeniería de sistemas

Fecha de lectura: 25/11/2016

Mención a doctor europeo: concedido

Programa de doctorado: Programa Oficial de Doctorado en Ingeniería de Sistemas e Informática

Dirección:

> **Director:** Darío Suárez Gracia

> **Director:** JESÚS ALASTRUEY BENEDÉ

Tribunal:

> **presidente:** Ulya R. Karpuzcu

> **secretario:** JOSE LUIS BRIZ VELASCO

> **vocal:** ALBERTO ROS BARDISA

Descriptores:

> INFORMATICA

> ARQUITECTURA DE ORDENADORES

El fichero de tesis ya ha sido incorporado al sistema

> <http://zaguan.unizar.es/record/57881>

Localización: UNIVERSIDAD DE ZARAGOZA

Resumen: Power density is currently the primary design constraint across most computing segments and the main performance limiting factor. For years, industry has kept power density constant, while increasing frequency, lowering transistors supply (V_{dd}) and threshold (V_{th}) voltages. However, V_{th} scaling has stopped because leakage current is exponentially related to it. Transistor count and integration density keep doubling every process generation (Moore's Law), but the power budget caps the amount of hardware that can be active at the same time, leading to dark silicon. With each new generation, there are more resources available, but we cannot fully exploit their performance potential. In the last years, different research trends have explored how to cope with dark silicon and unlock the energy efficiency of the chips, including Near-Threshold voltage Computing (NTC) and approximate computing.

NTC aggressively lowers V_{dd} to values near V_{th} . This allows a substantial reduction in power, as dynamic power scales quadratically with supply voltage. The resultant power reduction could be used to activate more chip resources and potentially achieve performance improvements. Unfortunately, V_{dd} scaling is limited by the tight



TESIS DOCTORALES - TESEO

functionality margins of on-chip SRAM transistors. When scaling Vdd down to values near-threshold, manufacture-induced parameter variations affect the functionality of SRAM cells, which eventually become not reliable.

A large amount of emerging applications, on the other hand, features an intrinsic error-resilience property, tolerating a certain amount of noise. In this context, approximate computing takes advantage of this observation and exploits the gap between the level of accuracy required by the application and the level of accuracy given by the computation, providing that reducing the accuracy translates into an energy gain. However, deciding which instructions and data and which techniques are best suited for approximation still poses a major challenge.

This dissertation contributes in these two directions. First, it proposes a new approach to mitigate the impact of SRAM failures due to parameter variation for effective operation at ultra-low voltages. We identify two levels of natural on-chip redundancy: cache level and content level. The first arises because of the replication of blocks in multi-level cache hierarchies. We exploit this redundancy with a cache management policy that allocates blocks to entries taking into account the nature of the cache entry and the use pattern of the block. This policy obtains performance improvements between 2% and 34%, with respect to block disabling, a technique with similar complexity, incurring no additional storage overhead. The latter (content level redundancy) arises because of the redundancy of data in real world applications. We exploit this redundancy compressing cache blocks to fit them in partially functional cache entries. At the cost of a slight overhead increase, we can obtain performance within 2% of that obtained when the cache is built with fault-free cells, even if more than 90% of the cache entries have at least a faulty cell.

Then, we analyze how the intrinsic noise tolerance of emerging applications can be exploited to design an approximate Instruction Set Architecture (ISA). Exploiting the ISA redundancy, we explore a set of techniques to approximate the execution of instructions across a set of emerging applications, pointing out the potential of reducing the complexity of the ISA, and the trade-offs of the approach. In a proof-of-concept implementation, the ISA is shrunk in two dimensions: Breadth (i.e., simplifying instructions) and Depth (i.e., dropping instructions). This proof-of-concept shows that energy can be reduced on average 20.6% at around 14.9% accuracy loss.



Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

Dª ESMERALDA MAINAR MAZA, Secretaria de la Escuela de Ingeniería y Arquitectura de la Universidad de Zaragoza,

CERTIFICA: Que, según los antecedentes obrantes en esta Secretaría, D/Dª **DARIO SUÁREZ GRACIA**, consta como director/ponente de los Trabajos Fin de Grado/Máster que se relacionan a continuación:

TÍTULO EFFICIENT INSTRUCTION AND DATA CACHING FOR HIGH PERFORMANCE LOW-POWER EMBEDDED SYSTEMS
MÁSTER UNIVERSITARIO EN INGENIERÍA DE SISTEMAS E INFORMÁTICA FECHA DEFENSA Y CALIFICACIÓN 05/10/2012 9,0 MAT. HONOR

TÍTULO CHARACTERIZATION OF INTERCONNECTION NETWORKS IN CMPS USING FULL-SYSTEM SIMULATION
MÁSTER UNIVERSITARIO EN INGENIERÍA DE SISTEMAS E INFORMÁTICA FECHA DEFENSA Y CALIFICACIÓN 05/10/2012 8,7 NOTABLE

TÍTULO SISTEMA DE INFORMACIÓN/ENTRETENIMIENTO PARA VEHÍCULO CON SOPORTE DE VOZ Y DIAGNÓSTICO
GRADO EN INGENIERÍA INFORMÁTICA FECHA DEFENSA Y CALIFICACIÓN 16/02/2017 8,2 NOTABLE

TÍTULO RECONOCIMIENTO DE OBJETOS EN ANDROID PARA APLICACIONES DE ASISTENCIA
GRADO EN INGENIERÍA INFORMÁTICA FECHA DEFENSA Y CALIFICACIÓN 14/12/2016 8,5 NOTABLE

TÍTULO ACCELERADORES HARDWARE PARA VISIÓN POR COMPUTADOR
GRADO EN INGENIERÍA INFORMATICA FECHA DEFENSA Y CALIFICACIÓN 10/07/2017 9,6 MATRÍCULA DE HONOR

Y, para que conste, a petición del interesado, expido la presente certificación con el VºBº del Sr. Director y sello del Centro, en Zaragoza a día 08 de septiembre de 2017.

VºBº
EL DIRECTOR,

Fdo.: José Angel Castellanos Gómez

LA SECRETARIA,

Fdo.: Esmeralda Mainar Maza

El Funcionario,

Fdo.: Pedro Felipe Sisamón



Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

4 Experiencia científica y tecnológica

4.1 Grupos/equipos de investigación, desarrollo o innovación

Darío Suárez Gracia - HiPEAC

<https://www.hipeac.net/~dario/>



*European Network on High Performance and
Embedded Architecture and Compilation*

Activities

HiPEAC



Darío Suárez Gracia

Profile

Publications ([/~dario/publications/](#))

[\(/acco](#)

[/login](#)

[/?next:](#)

[/~daric](#)

[\(/network/institutions/250/university-of-zaragoza/\)](#)



Institution: **University of Zaragoza** ([\(/network/institutions/250/university-of-zaragoza/\)](#))



Personal website (<http://webdiis.unizar.es/~dario/>)



HiPEAC **member** since April 2016.

HiPEAC Collaboration Grants

July 2009 - Oct. 2009

VLSI implementation of Light NUCAs @ Foundation for Research & Technology Hellas (FORTH)

[\(/network/institutions/280/foundation-for-research-technology-hellas-forth/\)](#)

1 of 2

02/10/2017, 17:18

4.2 Actividad científica o tecnológica

4.2.1 Proyectos de I+D+i financiados en convocatorias competitivas de Administraciones o entidades públicas y privadas



Prof. Pablo Ibáñez Marín
Área de Arquitectura y Tecnología de Computadores
Dpto. Informática e Ingeniería de Sistemas
Universidad de Zaragoza

Zaragoza, 20 de enero de 2012

Yo, Pablo Ibáñez Marín, con DNI 72964290, e investigador principal del Grupo de Arquitectura de Computadores de la Universidad de Zaragoza en el proyecto de investigación:

- 16 nodos de computación tipo Altix XE 210 con 2 procesadores de doble núcleo, Universidad de Zaragoza/Gobierno de Aragón, Julio 2007

Declaro que Darío Suárez Gracia, con DNI 18443694 ha participado de manera activa en el mismo y para que así conste firmo este certificado.



Fdo. Pablo Ibáñez Marín

Maria de Luna, 1 - 50018 ZARAGOZA (ESPAÑA)
Teléfonos: (+34) 976 761949 - (+34) 976 762406
Fax: (+34) 976 761914
e-mail: secinf@unizar.es
<http://diis.unizar.es/>

unizar.es



EL SECRETARIO GENERAL DE LA UNIVERSIDAD DE
ZARAGOZA, POR DELEGACIÓN EL VICESECRETARIO
GENERAL, VÍCTOR ESCARTÍN ESCUDÉ,

CERTIFICA: Que, según los antecedentes que obran en esta Secretaría General de mi cargo (Servicio de Gestión de la Investigación), **DARIÓ SUÁREZ GRACIA**, consta como investigador colaborador del proyecto/grupo de investigación Ref. JIUZ-2019-TEC-08, denominado: "JIUZ-2019-TEC-08: Técnicas de Redistribución de Registros en GPUs con Fallos Permanentes", desde el 1 de enero de 2020 hasta el 31 de diciembre de 2020, con dedicación única, financiado por FUNDACIÓN BANCARIA IBERCAJA, con una subvención prevista para la Universidad de Zaragoza de DOS MIL EUROS (2.000€), del que es investigador responsable Alejandro Valero Bresó.

Y para que conste, a petición del interesado, expide y firma el presente certificado en Zaragoza, a veintiuno de febrero de dos mil veinte.

(Firmado por delegación, resolución de 25/5/2016, el Vicesecretario General)



Vicerrectorado de
Política Científica
Universidad Zaragoza

Fecha: 4 de diciembre de 2015
Nº Rfa.: Vicerrectorado de Política Científica
LMG/mjs.

Destinatario:
GRAN TEJERO, RUBÉN
Dpto.: Informática e Ingeniería de Sistemas
Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

Asunto.: Resolución a la solicitud de ayudas a Proyectos de Investigación, desarrollo e innovación para jóvenes investigadores. (Convocatoria 2014)

La Comisión de Investigación de la Universidad de Zaragoza, en su sesión celebrada el día 21 de octubre de 2015, acordó financiar el proyecto de investigación: "JIUZ-2015-TEC-06: AUTOMATIZACIÓN DE LA GENERACIÓN DE LAS RESTRICCIONES DEL PROGRAMACIÓN LINEAL ENTERA PARA EL CÁLCULO DEL WCET EN SISTEMAS DE TIEMPO REAL ESTRÍCTO CON UN CACHE DE DATOS ACDC", con la cantidad de 2.000€.

Lo que le notifico como Presidente de la Comisión de Investigación, recordándole el compromiso de presentar una memoria de dicho Proyecto a la conclusión del mismo, en la que se incluirá la labor realizada y la producción científica/técnica obtenida.

Contra la presente resolución que no agota la vía administrativa de conformidad con el artº. 114 de la Ley 30/1992 de 26 de noviembre, de Régimen Jurídico de las Administraciones Públicas y del Procedimiento Administrativo Común, modificada por la Ley 4/1999 de 13 de enero, se podrá interponer recurso de alzada ante el Sr. Rector Magfco., en el plazo de un mes.

El Vicerrector de Política Científica



Luis Miguel García Vinuesa



Universidad
Zaragoza

EL SECRETARIO GENERAL DE LA UNIVERSIDAD DE
ZARAGOZA, POR DELEGACIÓN EL VICESECRETARIO
GENERAL, VÍCTOR ESCARTÍN ESCUDÉ,

CERTIFICA: Que, según los antecedentes que obran en esta Secretaría General de mi cargo (Oficina de Transferencia de Resultados de Investigación), **DARÍO SUÁREZ GRACIA**, consta como investigador responsable del proyecto/grupo de investigación Ref. RTC-2017-6421-7, denominado: "ANALIZAR EL COMPORTAMIENTO DE CONSUMIDORES CON MEDIDAS SIMULTÁNEAS DE FISIOLOGÍA, LOCALIZACIÓN Y VISIÓN POR COMPUTADOR RTC-2017-6421-7", desde el 1 de enero de 2018 hasta el 31 de diciembre de 2020, financiado por MINISTERIO DE ECONOMÍA Y COMPETITIVIDAD, con un importe previsto para la Universidad de Zaragoza de CIENTO SETENTA Y TRES MIL SEISCIENTOS SETENTA Y TRES EUROS CON TRES CÉNTIMOS (173.673,3€).

Y para que conste, a petición del interesado, expide y firma el presente certificado en Zaragoza, a veinte de febrero de dos mil veinte.

(Firmado por delegación, resolución de 25/5/2016, el Vicesecretario General)



EL SECRETARIO GENERAL DE LA UNIVERSIDAD DE
ZARAGOZA, POR DELEGACIÓN EL VICESECRETARIO
GENERAL, VÍCTOR ESCARTÍN ESCUDÉ,

CERTIFICA: Que, según los antecedentes que obran en esta Secretaría General de mi cargo (Servicio de Gestión de la Investigación), **DARÍO SUÁREZ GRACIA**, consta como investigador responsable del proyecto/grupo de investigación Ref. JIUZ-2017-TEC-09, denominado: "JIUZ-2017-TEC-09: ESTRATEGIAS DE MEJORA SOFTWARE Y HARDWARE PARA SISTEMAS ALTAMENTE HETEROGRÉNEOS: PLANIFICACIÓN DINÁMICA Y REDES ON CHIP.", desde el 1 de enero de 2018 hasta el 31 de diciembre de 2018, con dedicación única, financiado por FUNDACIÓN BANCARIA IBERCAJA, con una subvención prevista para la Universidad de Zaragoza de DOS MIL EUROS (2.000€).

Y para que conste, a petición del interesado, expide y firma el presente certificado en Zaragoza, a veintiuno de febrero de dos mil veinte.

(Firmado por delegación, resolución de 25/5/2016, el Vicesecretario General)



EL SECRETARIO GENERAL DE LA UNIVERSIDAD DE
ZARAGOZA, POR DELEGACIÓN EL VICESECRETARIO
GENERAL, VÍCTOR ESCARTÍN ESCUDÉ,

CERTIFICA: Que, según los antecedentes que obran en esta Secretaría General de mi cargo (Servicio de Gestión de la Investigación), **DARIÓ SUÁREZ GRACIA**, consta como investigador colaborador del proyecto/grupo de investigación Ref. T58_17R, denominado: "GRUPO DE REFERENCIA GRUPO DE ARQUITECTURA DE COMPUTADORES DE LA UNIZAR(gaZ)", desde el 1 de enero de 2017 hasta el 31 de diciembre de 2019, financiado por GOBIERNO DE ARAGÓN, con una subvención prevista para la Universidad de Zaragoza de TREINTA Y SIETE MIL SETECIENTOS NOVENTA Y DOS EUROS (37.792€), del que es investigador responsable Víctor Viñals Yufera.

Y para que conste, a petición del interesado, expide y firma el presente certificado en Zaragoza, a veintiuno de febrero de dos mil veinte.

(Firmado por delegación, resolución de 25/5/2016, el Vicesecretario General)



Universidad
Zaragoza

EL SECRETARIO GENERAL DE LA UNIVERSIDAD DE
ZARAGOZA, POR DELEGACIÓN EL VICESECRETARIO
GENERAL, VÍCTOR ESCARTÍN ESCUDÉ,

CERTIFICA: Que, según los antecedentes que obran en esta Secretaría General de mi cargo (Servicio de Gestión de la Investigación), **DARÍO SUÁREZ GRACIA**, consta como investigador colaborador del proyecto/grupo de investigación Ref. 287298, denominado: "JIUZ-2018-TEC-13:CARACTERIZACIÓN DEL ENVEJECIMIENTO DE LOS TRANSISTORES UTILIZADOS EN ACCELERADORES PARA REDES NEURONALES Y DISEÑO DE MECANISMOS ARQUITECTÓNICOS PARA COMBATIR SU EFECTO", desde el 1 de enero de 2019 hasta el 31 de diciembre de 2019, con dedicación única, financiado por FUNDACIÓN BANCARIA IBERCAJA, con una subvención prevista para la Universidad de Zaragoza de DOS MIL EUROS (2.000€), del que es investigador responsable Alejandro Valero Bresó.

Y para que conste, a petición del interesado, expide y firma el presente certificado en Zaragoza, a veintiuno de febrero de dos mil veinte.

(Firmado por delegación, resolución de 25/5/2016, el Vicesecretario General)



Universidad
Zaragoza

JUAN FRANCISCO HERRERO PEREZAGUA, SECRETARIO
GENERAL DE LA UNIVERSIDAD DE ZARAGOZA,

CERTIFICA: Que D. DARÍO SUÁREZ GRACIA, según los antecedentes que obran en esta Secretaría General de mi cargo (Servicio de Gestión de la Investigación), estuvo incluido en el equipo investigador que obtuvo una Subvención del Ministerio de Educación y Ciencia, durante el periodo comprendido entre el día 1 de agosto de 2005 y el 12 de diciembre de 2007, con dedicación única, para llevar a cabo el proyecto TIN2004-07739-C02-02, titulado:

el Dr. D.
Fdo: David Floria Vela

"COMPUTACIÓN DE ALTAS PRESTACIONES IV. JERARQUÍA
DE MEMORIA DE ALTAS PRESTACIONES".

del que fue investigador responsable D. Víctor Viñals Yufera.

Y para que conste, a petición del interesado, expide y firma el presente certificado en Zaragoza, a once de enero de dos mil doce.

V.º B.º
LA VICERRECTORA DE INVESTIGACIÓN,

Fdo: María Blanca Ros Latienda

Universidad
Zaragoza

In Latorre



Universidad
Zaragoza

JUAN FRANCISCO HERRERO PEREZAGUA, SECRETARIO
GENERAL DE LA UNIVERSIDAD DE ZARAGOZA,

CERTIFICA: Que D. DARÍO SUÁREZ GRACIA, según los antecedentes que obran en esta Secretaría General de mi cargo (Servicio de Gestión de la Investigación), estuvo incluido en el equipo investigador que obtuvo una Subvención del Ministerio de Educación y Ciencia, durante el periodo comprendido entre el día 1 de octubre de 2007 y el 30 de septiembre de 2010, con dedicación única, para llevar a cabo el proyecto TIN2007-66423, titulado:

Darío Suárez Gracia
Fdo: David Floria Vela

"JERARQUÍA DE MEMORIA DE ALTO RENDIMIENTO".

del que fue investigador responsable D. Víctor Viñals Yúfera.

Y para que conste, a petición del interesado, expide y firma el presente certificado en Zaragoza, a once de enero de dos mil doce.

V.º B.º
LA VICERRECTORA DE INVESTIGACIÓN,

Maria Blanca Ros Latienda
Fdo: María Blanca Ros Latienda



Universidad
Zaragoza

JUAN FRANCISCO HERRERO PEREZAGUA, SECRETARIO
GENERAL DE LA UNIVERSIDAD DE ZARAGOZA,

CERTIFICA: Que D. DARÍO SUÁREZ GRACIA, según los antecedentes que obran en esta Secretaría General de mi cargo (Servicio de Gestión de la Investigación), está incluido en el equipo investigador que ha obtenido una Subvención del Ministerio de Ciencia e Innovación, durante el periodo comprendido entre el dia 1 de enero de 2011 y el 31 de diciembre de 2013, con dedicación única, para llevar a cabo el proyecto TIN2010-21291-C02-01, titulado:

Darío Suárez Gracia
Fdo: David Flora Vela
"JERARQUÍA DE MEMORIA".

del que es investigador responsable D. Victor Viñals Yufera.

Y para que conste, a petición del interesado, expide y firma el presente certificado en Zaragoza, a once de enero de dos mil doce.

V.º B.^o
LA VICERRECTORA DE INVESTIGACIÓN,

Maria Blanca Ros Latienda
Fdo: María Blanca Ros Latienda



Universidad
Zaragoza

1542

JUAN FRANCISCO HERRERO PEREZAGUA, SECRETARIO
GENERAL DE LA UNIVERSIDAD DE ZARAGOZA,

CERTIFICA: Que D. DARÍO SUÁREZ GRACIA, según los antecedentes que obran en esta Secretaría General de mi cargo (Servicio de Gestión de la Investigación), estuvo incluido durante el periodo comprendido entre el 1 de enero de 2011 y el 31 de diciembre de 2013, en el Grupo de Investigación Consolidado,

Fdo: Aurelio Ocaña Martínez

"GRUPO DE ARQUITECTURA DE COMPUTADORES DE LA
UNIVERSIDAD DE ZARAGOZA (GAZ)".

que fue reconocido por el Gobierno de Aragón por Resolución de 15 de abril de 2011, del Director General de Investigación, Innovación y Desarrollo, del Departamento de Ciencia, Tecnología y Universidad (B.O.A. de 9 de mayo de 2011).

Y para que conste, a petición del interesado, expide y firma el presente certificado en Zaragoza, a veintiuno de mayo de dos mil catorce.

V.º B.
EL VICERRECTOR DE POLÍTICA
CIENTÍFICA,

Fdo: Luis Miguel García Vinuesa



Departamento de
Informática e
Ingeniería de Sistemas
Universidad Zaragoza

Prof. Víctor Viñals Yúfera
Área de Arquitectura y Tecnología de Computadores
Dpto. Informática e Ingeniería de Sistemas
Universidad de Zaragoza

El abajo firmante, Víctor Viñals Yúfera, con DNI 37318202N, e investigador principal del Grupo de Arquitectura de Computadores de la Universidad de Zaragoza en los siguientes proyectos de investigación:

- HiPEAC₃ Network of Excellence. High-Performance Embedded Architectures and Compilers 3, FET ICT-287759. Desde enero de 2012 a diciembre de 2015.
- HiPEAC₂ Network of Excellence. High-Performance Embedded Architectures and Compilers 2. Desde marzo de 2008 hasta marzo de 2011.
- HiPEAC Network of Excellence. High-Performance Embedded Architectures and Compilers. IST-004408. Desde marzo de 2004 hasta marzo de 2008.
- Reconocimiento y financiación de grupo Consolidado de investigación. **gaz: Grupo de Arquitectura de Computadores de la UZ**, incluido en el listado de Unidades Operativas de Investigación, bajo la tipología "Grupo de Investigación Consolidado". Desde enero 2005 hasta la actualidad, renovado anualmente en concurrencia competitiva.

Declaro que Dario Suárez Gracia, con DNI 18443694 ha participado de manera activa en todos los proyectos anteriores, contribuyendo a los objetivos planteados en todos ellos.
Y para que así conste firmo este certificado.

En Zaragoza, a 4 de junio de 2014



Departamento de
Informática e Ingeniería
de Sistemas
Universidad Zaragoza

Fdo. Víctor Viñals Yúfera
Catedrático de Arquitectura
y Tecnología de Computadores

Maria de Luna, 1 - 50018 ZARAGOZA (ESPAÑA)
Teléfonos: (+34) 976 761949 - (+34) 976 762406
Fax: (+34) 976 761914
e-mail: secin@unizar.es
<http://diis.unizar.es/>

unizar.es



UNIVERSIDAD DE ZARAGOZA

JUAN FRANCISCO HERRERO PEREZAGUA, SECRETARIO
GENERAL DE LA UNIVERSIDAD DE ZARAGOZA,

CERTIFICA: Que, según los antecedentes que obran en esta Secretaría General de mi cargo (Unidad de Proyectos Internacionales de Investigación), D.
DARÍO SUÁREZ GRACIA NIF: 18443694V, participa como investigador colaborador, en el proyecto de investigación nº EFA35/08 -PIREGRID-, financiado por la Comisión Europea a través de Fondos FEDER, cuyas entidades participantes son el Instituto Universitario de Investigación de Biocomputación y Física de Sistemas Complejos-BIFI, el Instituto Universitario de Investigación en Ingeniería de Aragón- I3A, la Université de Pau et des Pays d'Addour (UPPA), la Université Paul Sabatier Toulouse II-IRIT y la Chambre de Commerce et d'Industrie Pau Béarn-CCI, con un presupuesto total de ciento cincuenta y tres mil quinientos diez euros (153.510,00 €) del que la subvención prevista para la Universidad de Zaragoza es de noventa y nueve mil setecientos ochenta y un euros y cincuenta céntimos (99.781,50 €).

La investigadora responsable es Dña. María Villarroya Gaudó, y la duración prevista comprende el periodo del uno de junio de dos mil nueve al treinta y uno de mayo de dos mil doce.

Y para que conste, a petición del interesado, expide y firma el presente certificado en Zaragoza, a diecisésis de enero de dos mil doce.

V.º B.º
LA VICERRECTORA DE INVESTIGACIÓN

Fdo.: MARÍA BLANCA ROS LATIENDA



MINISTERIO
DE ECONOMÍA
Y COMPETITIVIDAD

SECRETARIA DE ESTADO DE INVESTIGACIÓN,
DESARROLLO E INNOVACIÓN
SECRETARÍA GENERAL DE CIENCIA,
TECNOLOGÍA E INNOVACIÓN
DIRECCIÓN GENERAL DE INVESTIGACIÓN
CIENTÍFICA Y TÉCNICA
SUBDIRECCIÓN GENERAL
DE PROYECTOS DE INVESTIGACIÓN

INFORME INTERMEDIO DE PROYECTOS COORDINADOS DE I+D+i

Como paso previo a la realización del informe, se ruega lean detenidamente las **instrucciones de elaboración de los informes de seguimiento científico-técnico de proyectos** disponible al final de este informe.

Se recomienda leer atentamente la información solicitada en los distintos apartados del informe, revisar la memoria y el presupuesto solicitado inicialmente y justificar adecuadamente todas aquellas actividades o gastos que haya sido necesario realizar para la consecución de los objetivos y que no estuvieran previstos o suficientemente detallados en la memoria inicial

A. Datos de coordinación

Nota: Relacione los subproyectos que participan en el proyecto coordinado

Proyecto coordinador (1)	Referencia de proyecto:
Investigador Principal 1	Víctor Viñals Yúfera
Investigador Principal 2*	Pablo Enrique Ibáñez Marín
Entidad	Universidad de Zaragoza
Centro	Instituto Universitario de Investigación en Ingeniería de Aragón
Subproyecto (2)	Referencia de proyecto:
Investigador Principal 1	Fernando Vallejo Alonso
Investigador Principal 2*	Ramón Beivide Palacio
Entidad	Universidad de Cantabria
Centro	Dpto. Electrónica y Computadores

Nota: Cree tantas tablas como subproyectos formen parte de proyecto coordinado.

* Rellenar si procede.

B. Datos del subprojeto

Relacione los datos actuales del subprojeto. En caso de que haya alguna modificación, indíquelo en la casilla B2

B1. Datos del proyecto	
Referencia proyecto	TIN2013-46957-C2-1-P
Título	JERARQUÍA DE MEMORIA Y APLICACIONES
Investigador Principal 1	Víctor Viñals Yúfera
Investigador Principal 2*	Pablo Enrique Ibáñez Marín
Entidad	Universidad de Zaragoza
Centro	Instituto Universitario de Investigación en Ingeniería de Aragón
Fecha de inicio	01/01/2014
Fecha final	31/12/2016
Duración	3 Años
Total concedido	129.300,60 €

* Rellenar si procede.

B2. Descripción de modificaciones en los datos iniciales del subprojeto (Cambio de IP, entidad, centro, modificación del periodo de ejecución....)

NO



C. Personal activo en el subproyecto

Tiene que relacionar la situación de todo el personal de las entidades participantes que haya prestado servicio en el proyecto en el período que se justifica, o que no haya sido declarado anteriormente, y cuyos costes (dietas, desplazamientos, etc.) se imputen al mismo.

C1. Equipo de investigación

Incluido en la solicitud original

	Nombre	NIF/NIE	Función en el proyecto	Fecha de alta	Fecha de baja	Observaciones
1	Jesús Alastruey Benedé	18036419A	Investigación para lograr los objetivos O1 y O11.			
2	Luis Carlos Aparicio Cardiel	25443527F	Investigación para lograr el objetivo O2.			
3	José Luis Briz Velasco	16528614D	Investigación para lograr el objetivo O3.			
4	Pablo Ibáñez Marín	72964290X	Investigador Principal, leader de los objetivos O1, O5 y O11			
5	Teresa Monreal Amal	72964553C	Investigación para lograr el objetivo O1.			
6	Luis M. Ramos Martínez	34103189P	Investigación para lograr los objetivos O1 y O9.			
7	Jesús Javier Resano Ezcaray	29119419Q	Investigación para lograr los objetivos O4 y O10.			
8	Clemente Rodríguez Lafuente	16514369R	Investigación para lograr el objetivo O2.			
9	Benjamín Sahelices Fernández	09770867F	Investigación para lograr el objetivo O5.			
10	Juan Segarra Flor	52944402N	Investigación para lograr el objetivo O2.			
11	Enrique Torres Moreno	16020342Z	Investigación para lograr los objetivos O4 y O10.			
12	Maria Villarroya Gaudio	25480031X	Investigación para lograr el objetivo O7.			
13	Víctor Viñals Yúfera	37318201N	Investigador Principal, leader de los objetivos O2, O3, O4 y O10. Investigación para lograr los objetivos O1 y O7.			

No incluido en la solicitud original

	Nombre	NIF/NIE	Función en el proyecto	Fecha de alta	Fecha de baja	Observaciones
1						
2						

C2. Equipo de Trabajo



	Nombre	NIF/NIE	Función en el proyecto	Fecha de alta	Fecha de baja	Observaciones
1	Jorge Albericio Latorre	73088149	Investigación para lograr el objetivo O1.	01/01/2014		Incluido en la solicitud inicial como investigador Doctor
2	Javier Diaz Maag		Investigación para lograr el objetivo O1.	01/01/2014		
3	Alexandra Ferrerón Labarí	76923792Q	Investigación para lograr el objetivo O1.	01/01/2014		
4	Pablo García Risueño		Investigación para lograr el objetivo O11.	01/01/2014		Incluido en la solicitud inicial como investigador Doctor
5	María Jesús Garzaran		Investigación para lograr el objetivo O9.	01/01/2014		Incluido en la solicitud inicial como investigador Doctor
6	Rubén Gran Tejero	72971781A	Investigación para lograr los objetivos O2 y O9.	01/01/2014		Incluido en la solicitud inicial como investigador Doctor
7	Maria Cruz Izu Belloso	29143497J	Investigación para lograr el objetivo O7.	01/01/2014		Incluido en la solicitud inicial como investigador Doctor
8	Carl Christian Kjelgaard Mikkelsen		Investigación para lograr el objetivo O11.	01/01/2014		Incluido en la solicitud inicial como investigador Doctor
9	Francisco Javier Olivito del Ser	17760528C	Investigación para lograr los objetivos O4 y O10.	01/01/2014		
10	Marta Ortín Obón	73005776G	Investigación para lograr el objetivo O7.	01/01/2014		
11	Alba Pedro Zapater	73002477V	Investigación para lograr el objetivo O2.	13/01/2015		Alta nueva en el equipo de trabajo
12	Xuehai Qian		Investigación para lograr el objetivo O5.	01/01/2014		Incluido en la solicitud inicial como investigador Doctor
13	Alamelu Sankaranarayanan		Investigación para lograr el objetivo O3.	01/01/2014		
14	María Astón Serrano Gracia	18053660V	Investigación para lograr el objetivo O11.	01/01/2014	01/10/2014	Baja por cambio de entidad (BSC-UPC, Barcelona)
15	Dario Suárez Gracia	18443694V	Investigación para lograr los objetivos O1 y O7.	01/01/2014		Incluido en la solicitud inicial como investigador Doctor

Nota: Cree tantas filas como necesite.

Las "Altas" y "Bajas" de nuevos investigadores en el **equipo de investigación** deben tramitarse de acuerdo con las instrucciones de ejecución y justificación expuesta en la página web del ministerio. La incorporación de personal que participe en el proyecto en el **equipo de trabajo** no necesita autorización por parte del ministerio, pero su actividad debe incluirse y justificarse en este informe

D. Progreso y resultados del proyecto coordinado

D1. Desarrollo de los objetivos planteados en el proyecto coordinado (a rellenar por el coordinador)
Describa los objetivos del proyecto coordinado y el grado de cumplimiento de los mismos (porcentaje)

3 / 22



MINISTERIO
DE ECONOMÍA, INDUSTRIA
Y COMPETITIVIDAD



Referencia: TIN2016-76635-C2-1-R

Título: ARQUITECTURA Y PROGRAMACION DE COMPUTADORES ESCALABLES DE ALTO RENDIMIENTO Y

BAJO CONSUMO

Entidad Solicitante: UNIVERSIDAD DE ZARAGOZA

Centro: INSTITUTO UNIVERSITARIO DE INVESTIGACION EN INGENIERIA DE ARAGON

Investigador/a Principal : PABLO IBÁÑEZ MARÍN

Duración (en años): 3

La finalidad del presente escrito es informarle de la composición del equipo de investigación que, según consta en nuestra base de datos, va a participar en el desarrollo del proyecto de referencia, **TIN2016-76635-C2-1-R**, del que es usted investigador/a principal.

El artículo 7 de la convocatoria establece que:

a) Los requisitos y el régimen de compatibilidad y dedicación de los investigadores principales y del resto de los componentes del equipo de investigación y, en su caso, del equipo de trabajo, serán los establecidos en cada actuación. Dichos requisitos deberán cumplirse el día en que finalice el plazo de presentación de solicitudes y mantenerse hasta la fecha final del plazo solicitado de ejecución de la actuación, salvo las excepciones previstas, en su caso, en las diferentes actuaciones.

b) El incumplimiento de los requisitos citados por parte de alguno de los otros miembros del equipo de investigación determinará la exclusión de dicho investigador de todos los proyectos o actuaciones solicitados en los que figure.

c) La entidad solicitante será la responsable de verificar el cumplimiento de las condiciones de titulación, vinculación, compatibilidad, dedicación y cualesquier otros requisitos exigidos al personal investigador participante en esta convocatoria, debiendo comunicar al órgano instructor cualquier variación en un plazo de 10 días a partir de aquél en que se produzca el cambio, en la forma que se determina en el **artículo 15**.

Los investigadores que figuran "excluidos" en esta comunicación podrán, no obstante, formar parte del equipo de trabajo del proyecto. La pertenencia al equipo de trabajo de un proyecto de investigación no queda anotada en nuestra base de datos por lo que no se responderá a solicitudes de cambio de investigadores del equipo de investigación al equipo de trabajo del proyecto. Las tareas de investigación, así como los gastos generados en relación con un proyecto de investigación de cualquier investigador que no pertenezca al equipo de investigación del proyecto, deberán reflejarse expresamente en los sucesivos y preceptivos informes de seguimiento que remitirá el investigador principal a la **Subdivisión de Programas Temáticos Científico-Técnicos** en los períodos de justificación científico-técnica.

El investigador principal podrá solicitar la inclusión de los investigadores que figuran, en su caso, en la relación del equipo de investigación como "excluidos" cuando cumplan con los requisitos exigidos en la convocatoria. La solicitud se hará mediante el envío de una "Instancia genérica" a través de Facilit@ (<https://sede.micinn.gob.es/facilita>), que deberá completar el representante legal de la entidad solicitante mediante su firma electrónica.

Se recuerda que no se autorizan bajas en proyectos anteriores para participar en la presente convocatoria.

dtpc@mineco.es



MINISTERIO
DE ECONOMÍA, INDUSTRIA
Y COMPETITIVIDAD



EQUIPO DE INVESTIGACIÓN REGISTRADO EN LA SOLICITUD:

NOMBRE Y APELLIDOS	EXCLUIDO (SÍ/NO)	DEDICACIÓN
PABLO IBÁÑEZ MARÍN (IP)	NO	1,0
JOSE LUIS BRIZ VELASCO (RESTO EQUIPO)	NO	1,0
VICTOR VIÑALS YUFERA (RESTO EQUIPO)	NO	1,0
JESUS JAVIER RESANO EZCARAY (IP2)	NO	1,0
BENJAMIN SAHÉLICES FERNANDEZ (RESTO EQUIPO)	NO	0,5
TERESA MONREAL ARNAL (RESTO EQUIPO)	NO	0,5
CLEMENTE RODRIGUEZ LAFUENTE (RESTO EQUIPO)	NO	1,0
SIHAM TABIK (RESTO EQUIPO)	SÍ	0,5
DARIO SUAREZ GRACIA (RESTO EQUIPO)	SÍ	1,0
RUBEN GRAN TEJERO (RESTO EQUIPO)	NO	1,0
LUIS CARLOS APARICIO CARDIEL (RESTO EQUIPO)	NO	1,0
MARIA VILLARROTA GAUDO (RESTO EQUIPO)	NO	1,0
ENRIQUE TORRES MORENO (RESTO EQUIPO)	NO	1,0
JUAN SEGARRA FLOR (RESTO EQUIPO)	NO	1,0
LUIS MANUEL RAMOS MARTINEZ (RESTO EQUIPO)	NO	1,0
JESUS JAVIER ALASTRUEY BENEDE (RESTO EQUIPO)	NO	1,0

CAUSAS DE LA EXCLUSIÓN DEL INVESTIGADOR:

NOMBRE Y APELLIDOS	CAUSA DE EXCLUSIÓN

dtpc@mineco.es



**Vicerrectorado de
Política Científica
Universidad Zaragoza**

Fecha: 20 de octubre de 2017
Nº Rfa.: Vicerrectorado de Política Científica
LMG/mjs.

Destinatario:
SUÁREZ GRACIA, DARÍO
Dpto.: Informática e Ingeniería de Sistemas
Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

Asunto.: Resolución a la solicitud de ayudas a Proyectos de Investigación, desarrollo e innovación para jóvenes investigadores. (Convocatoria 2017)

La Comisión de Investigación de la Universidad de Zaragoza, en su sesión celebrada el dia 4 de octubre de 2017, acordó financiar el proyecto de investigación: "JIUZ-2017-TEC-09: ESTRATEGIAS DE MEJORA SOFTWARE Y HARDWARE PARA SISTEMAS ALTAMENTE HETEROGENEOS: PLANIFICACIÓN DINAMICA Y REDES ON CHIP", con la cantidad de **2.000 €**.

Lo que le notifico como Presidente de la Comisión de Investigación, recordándole el compromiso de presentar una memoria de dicho Proyecto a la conclusión del mismo, en la que se incluirá la labor realizada y la producción científica/técnica obtenida.

Contra la presente resolución, que no agota la vía administrativa, de conformidad con los artículos 121 y 122 de la Ley 39/2015, de 1 de octubre, del Procedimiento Administrativo Común de las Administraciones Públicas, se podrá interponer recurso de alzada ante el Rector Magnífico de la Universidad de Zaragoza, en el plazo de un mes.

El Vicerrector de Política Científica


Universidad
Zaragoza
Luis Miguel García Vinuesa

Pl. Basilio Paraíso, 4. Edif. Paraninfo. Universidad de Zaragoza
Tl. (+34) 976 761012. vpcientifica@unizar.es

4.2.2 Contratos, convenios o proyectos de I+D+i no competitivos con Administraciones o entidades públicas o privadas



EL SECRETARIO GENERAL DE LA UNIVERSIDAD DE ZARAGOZA, POR DELEGACIÓN EL VICESECRETARIO GENERAL, VÍCTOR ESCARTÍN ESCUDÉ,

CERTIFICA: Que, según los antecedentes que obran en esta Secretaría General de mi cargo (Oficina de Transferencia de Resultados de Investigación), **DARÍO SUÁREZ GRACIA**, consta como investigador responsable del proyecto/grupo de investigación Ref. 2020/0076, denominado: "ESTUDIO DEL RENDIMIENTO DE UN SISTEMA DE ADQUISICIÓN Y ANÁLISIS DE SEÑALES DE FIBRA ÓPTICA", desde el 1 de febrero de 2020 hasta el 30 de septiembre de 2020, financiado por ARAGON PHOTONICS LABS, S.L., con un importe previsto para la Universidad de Zaragoza de CINCO MIL OCHENTA Y DOS EUROS (5.082€).

Y para que conste, a petición del interesado, expide y firma el presente certificado en Zaragoza, a veinte de febrero de dos mil veinte.

(Firmado por delegación, resolución de 25/5/2016, el Vicesecretario General)



1542

Universidad
Zaragoza

EL SECRETARIO GENERAL DE LA UNIVERSIDAD DE
ZARAGOZA, POR DELEGACIÓN EL VICESECRETARIO
GENERAL, VÍCTOR ESCARTÍN ESCUDÉ,

CERTIFICA: Que, según los antecedentes que obran en esta Secretaría General de mi cargo (Oficina de Transferencia de Resultados de Investigación), **DARÍO SUÁREZ GRACIA**, consta como investigador responsable del proyecto/grupo de investigación Ref. 2018/0429, denominado: "COLABORACIÓN PARA LA REALIZACIÓN DE PRUEBAS TÉCNICAS EN PROCESO DE SELECCIÓN DE PROMOCIÓN INTERNA DE AST", desde el 2 de abril de 2018 hasta el 4 de septiembre de 2018, financiado por ARAGONESA DE SERVICIOS TELEMATICOS, con un importe previsto para la Universidad de Zaragoza de MIL QUINIENTOS EUROS (1.500€).

Y para que conste, a petición del interesado, expide y firma el presente certificado en Zaragoza, a veinte de febrero de dos mil veinte.

(Firmado por delegación, resolución de 25/5/2016, el Vicesecretario General)



Oficina de
Transferencia de Resultados
de Investigación. OTRI
UniversidadZaragoza

Dª. Raquel Rodríguez Bailera, Directora Técnica de la Oficina de Transferencia de Resultados de Investigación (O.T.R.I.) de la Universidad de Zaragoza,

HACE CONSTAR

Que, según figura en nuestra documentación, **D. Darío Suárez Gracia** con NIF 18443694V, perteneciente al Departamento de Informática e Ingeniería de Sistemas de la Escuela de Ingeniería y Arquitectura de la Universidad de Zaragoza, ha sido Investigador Principal del proyecto gestionado por esta oficina cuyas características se detallan a continuación:

Código O.T.R.I.	Título del proyecto	Entidad financiadora	Duración
2016/0344	HIGH PERFORMANCE LOW POWER COMPUTER VISION FOR AUGMENTED REALITY	EONITE PERCEPTION INC	De 16/08/2016 A 15/08/2017

Lo que se hace constar, a solicitud del interesado, en Zaragoza, a uno de septiembre de dos mil diecisiete.


Raquel Rodríguez Bailera
Oficina de Transferencia de Resultados de Investigación. OTRI
UniversidadZaragoza



Oficina de
Transferencia de Resultados
de Investigación. OTRI
UniversidadZaragoza

Dª. Raquel Rodríguez Bailera, Directora Técnica de la Oficina de Transferencia de Resultados de Investigación (O.T.R.I.) de la Universidad de Zaragoza,

HACE CONSTAR

Que, según figura en nuestra documentación, **D. Darío Suárez Gracia** con NIF 18443694V, perteneciente al Departamento de Informática e Ingeniería de Sistemas de la Escuela de Ingeniería y Arquitectura de la Universidad de Zaragoza, ha sido Investigador Principal del proyecto gestionado por esta oficina cuyas características se detallan a continuación:

Código O.T.R.I.	Título del proyecto	Entidad financiadora	Duración
2016/0344	HIGH PERFORMANCE LOW POWER COMPUTER VISION FOR AUGMENTED REALITY	EONITE PERCEPTION INC	De 16/08/2016 A 15/08/2017

Lo que se hace constar, a solicitud del interesado, en Zaragoza, a uno de septiembre de dos mil diecisiete.


Raquel Rodríguez Bailera
Oficina de Transferencia de Resultados de Investigación. OTRI
UniversidadZaragoza

5 Resultados

5.1 Propiedad industrial e intelectual



(21) **United States Patent**
Suarez Garcia et al.

(10) **Patent No.:** US 9,357,397 B2
(43) **Date of Patent:** May 31, 2016

(54) **METHODS AND SYSTEMS FOR DETECTING MALWARE AND ATTACKS THAT TARGET BEHAVIORAL SECURITY MECHANISMS OF A MOBILE DEVICE**

(71) **Applicant:** QUALCOMM Incorporated, San Diego, CA (US)

(72) **Inventors:** Darlin Suarez Garcia, Santa Clara, CA (US); Rajarsi Gupta, Sunnyvale, CA (US); Alexander Gantman, Solana Beach, CA (US)

(73) **Assignee:** QUALCOMM Incorporated, San Diego, CA (US)

(*) **Notice:** Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 103 days.

(21) **Appl. No.:** 14/038,638

(22) **Filed:** Jul. 23, 2014

(16) **Prior Publication Data**
US 2016/0029221 A1 Jun. 28, 2016

(51) **Int. Cl.:**
H04M 1/66 (2006.01)
H04M 3/16 (2006.01)
H04W 12/00 (2009.01)
G06F 21/56 (2013.01)
H04L 29/06 (2006.01)
G06F 21/57 (2013.01)
H04W 12/12 (2009.01)
H04W 24/06 (2009.01)

(52) **US. Cl.:**
CPC: *H04W 12/10* (2013.01) & *G06F 21/561* (2013.01); *G06F 21/566* (2013.01) & *G06F 21/571* (2013.01); *H04L 63/1416* (2013.01); *H04L 63/1433* (2013.01); *H04W 12/12* (2013.01); *H04W 24/06* (2013.01)

(58) **Field of Classification Search:**

None

See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

7,577,134 B2 8,25299 Sinha et al.
7,591,958 B2 8,3911 Venkatasubramanian et al.
7,598,560 B1 6,9211 Gunaswamy

(Continued)

FOREIGN PATENT DOCUMENTS

WO 2006088536 A2 9, 2006
WO 2013173881 A1 11, 2013
WO 201406500 A1 5, 2014

OTHER PUBLICATIONS

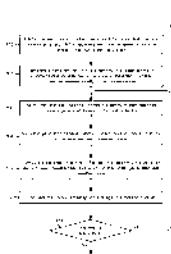
International Search Report and Written Opinion - PCT/US2013/039285 ISA/JPPO Sep. 16, 2013

Primary Examiner: Erika A Washington
Attorney, Agent, or Firm: The Marbury Law Group, PLLC

(57) **ABSTRACT**

A behavior-based security system of a computing device may be protected from non-benign behavior, malware, and other attacks by configuring the device to work in conjunction with another component (e.g., a server) to monitor the accuracy and performance of the security system, and determine whether the system is working correctly, efficiently, or as expected. This may be accomplished via the server generating artificial attack software, sending the generated artificial attack software to the mobile device to simulate non-benign behavior in the mobile device, such as a cyber attack, and determining whether the behavior-based security system of the mobile device responded adequately to the simulated non-benign behavior. The server may send a dead-man signal to the mobile device in response to determining that the behavior-based security system of the mobile device did not respond adequately to the simulated non-benign behavior.

26 Claims, 8 Drawing Sheets



US 9,357,397 B2

Page 2

(56)	References Cited	2007-0174917 AL*	7,2667, Gummoway	10501 63 1437
	U.S. PATENT DOCUMENTS	2011-0138170 AL	6,3911, Davis et al	736,25
	2012-0137464 AL	5,3912, Blodgett		
2010-0621048 AL	1,2000, Cvek et al	2011-0041036 AL	3,3913, Hingue et al	
2007-1045030 AL*	6,2007, Sinha	2011-0075353 AL	3,3914, Barbato et al	
	1041 63 1433	455 410	*	cited by examiner

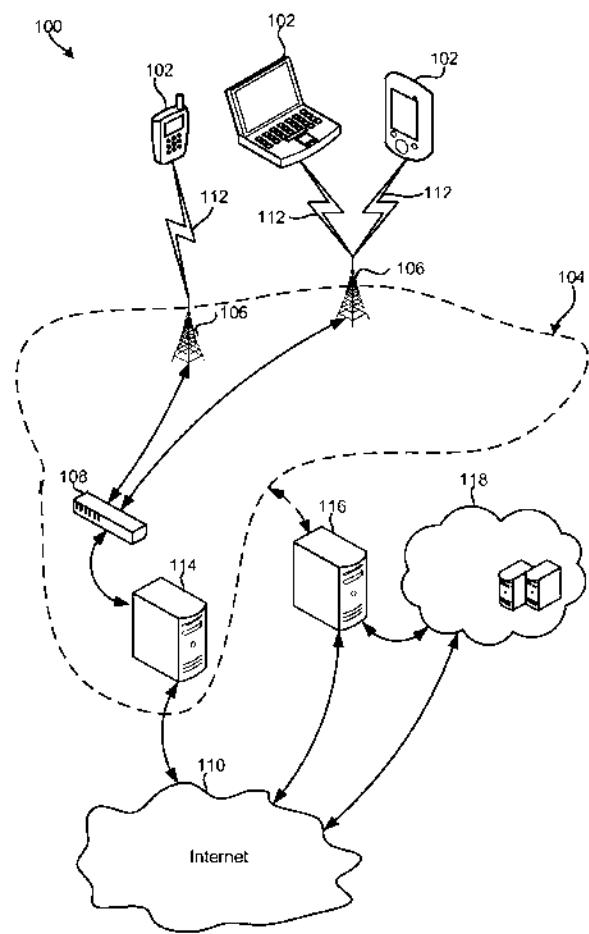


FIG. 1

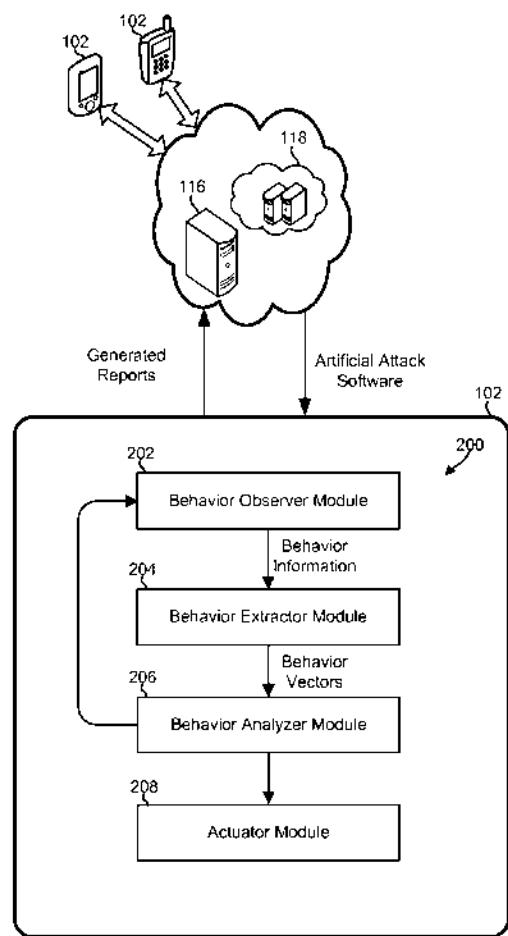


FIG. 2

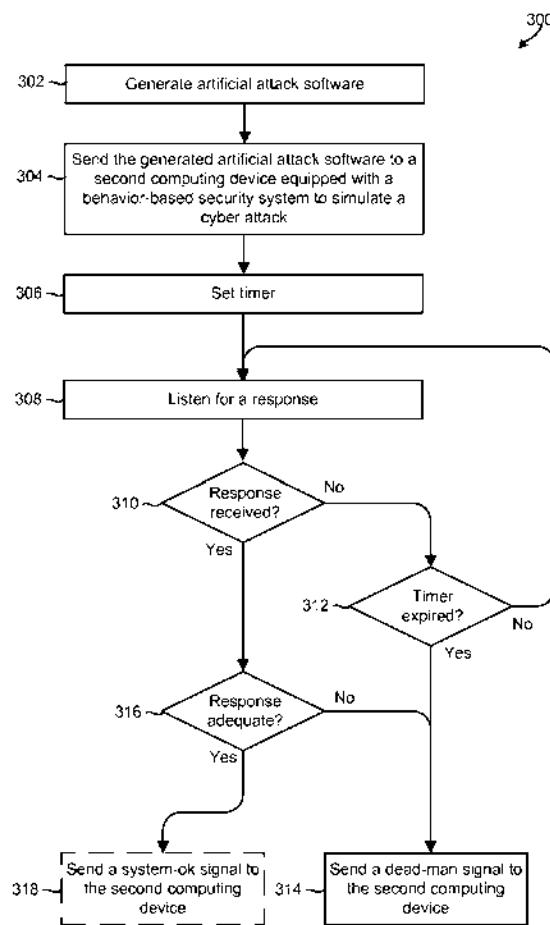


FIG. 3

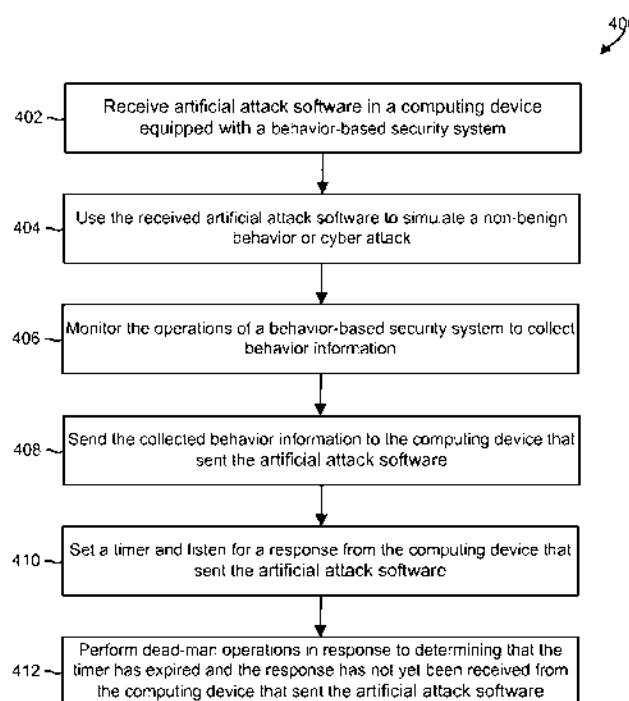


FIG. 4

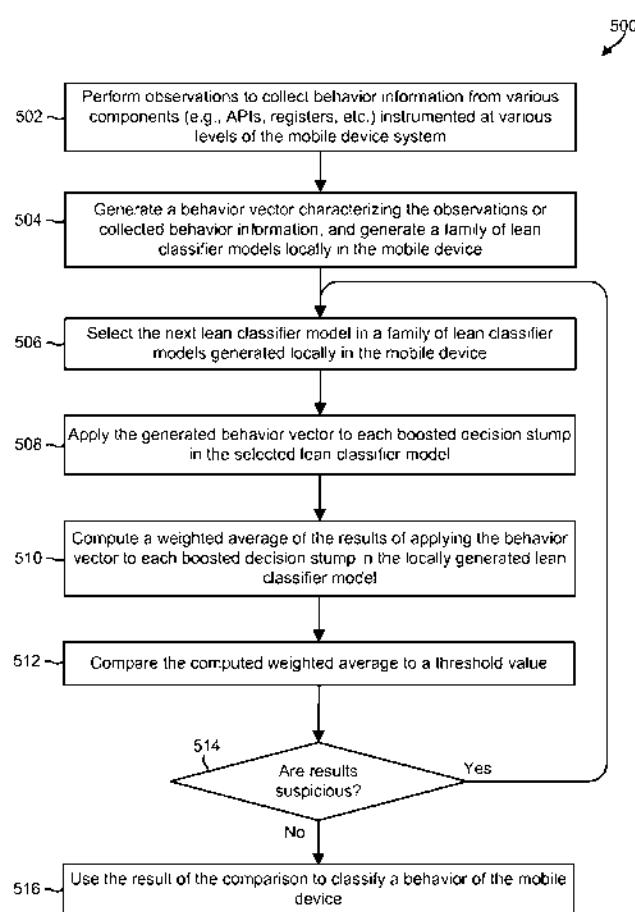


FIG. 5

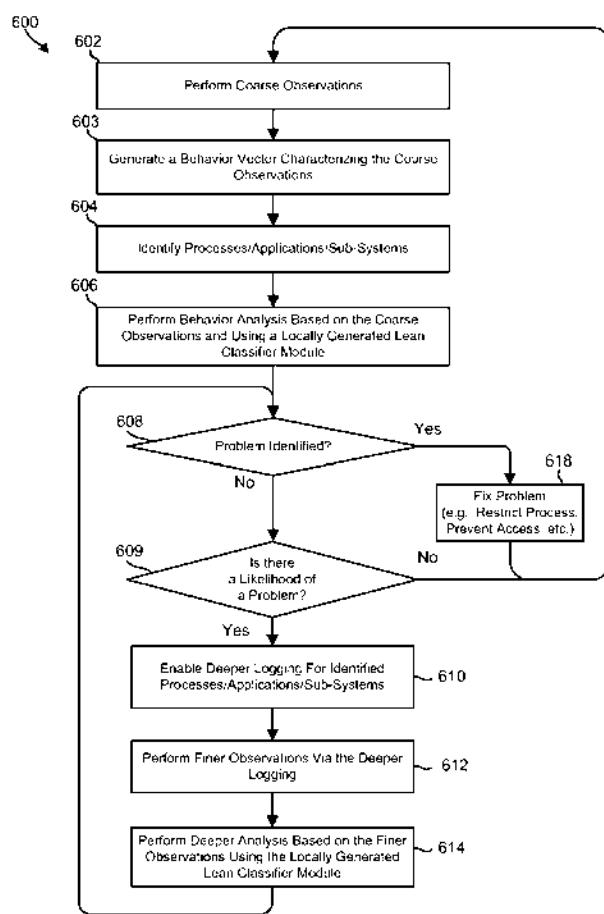


FIG. 6

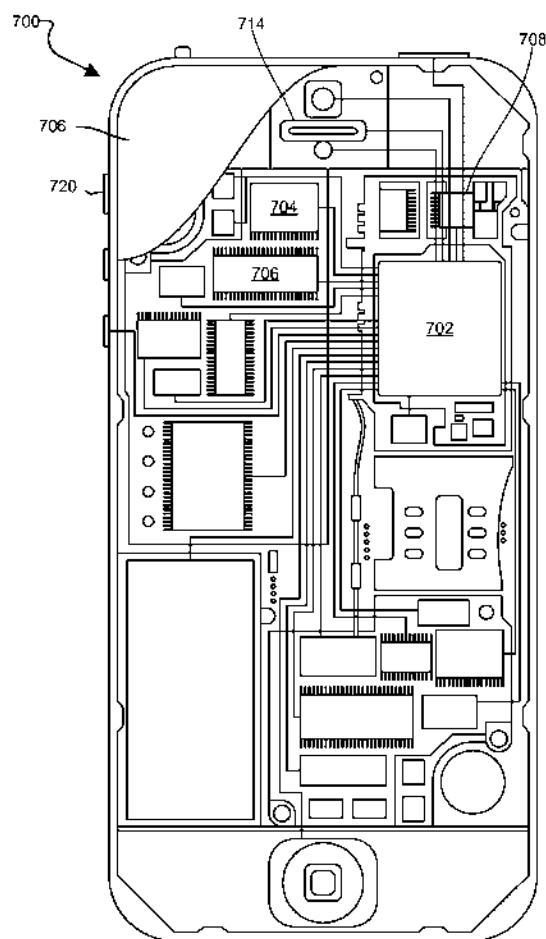


FIG. 7

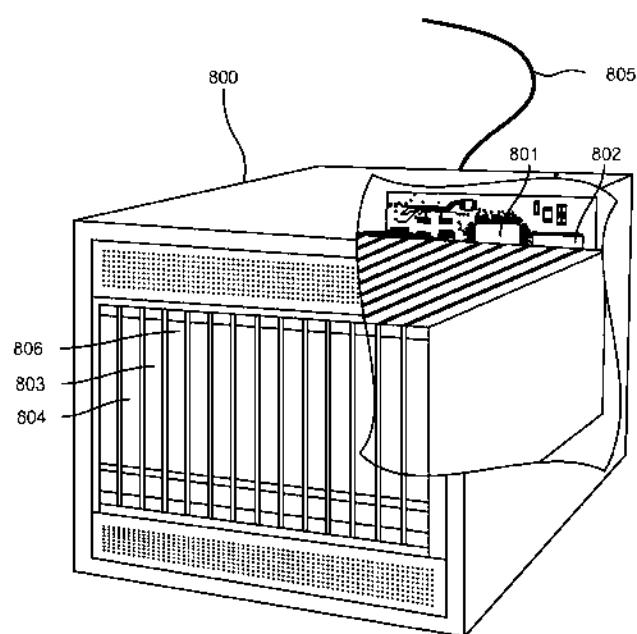


FIG. 8

**METHODS AND SYSTEMS FOR DETECTING
MALWARE AND ATTACKS THAT TARGET
BEHAVIORAL SECURITY MECHANISMS OF
A MOBILE DEVICE**

B. BACKGROUND

Cellular and wireless communication technologies have seen explosive growth over the past several years. Wireless service providers now offer a wide array of features and services that provide their users with unprecedented levels of access to information, resources and communications. To keep pace with these advancements, consumer electronic devices (e.g., cellular phones, watches, headphones, remote controls, etc.) have become more powerful and complex than ever, and now commonly include powerful processors, large memories, and other resources that allow for executing complex and powerful software applications on these devices. These devices also enable their users to download and execute a variety of software applications from application download services (e.g., Apple® App Store, Windows® Store, Google® play, etc.) or the Internet.

Due to these and other improvements, an increasing number of mobile and wireless device users now use their devices to store sensitive information (e.g., credit card information, contacts, etc.) and/or to accomplish tasks for which security is important. For example, mobile device users frequently use their devices to purchase goods, send and receive sensitive communications, pay bills, manage bank accounts, and conduct other sensitive transactions. Due to these trends, mobile devices are quickly becoming the next frontier for malware and cyber attacks. Accordingly, new and improved security solutions that better protect resource-constrained computing devices, such as mobile and wireless devices, will be beneficial to consumers.

SUMMARY

The various aspects include methods of using behavior-based security system to intelligently and efficiently identify, prevent, and/or detect the conditions, factors, and/or behaviors that often degrade a mobile device's performance and/or power utilization levels over time. The various aspects also include methods of applying the behavior-based security system of the mobile device, which may include generating artificial attack software configured to simulate a non-benign behavior in a mobile device, sending the generated artificial attack software to the mobile device so as to simulate a non-benign behavior in the mobile device, determining whether the behavior-based security system of the mobile device responded adequately to the simulated non-benign behavior, and sending a dead-man signal to the mobile device in response to determining that the behavior-based security system of the mobile device did not respond adequately to the simulated non-benign behavior.

In an aspect, determining whether the behavior-based security system of the mobile device responded adequately to the simulated non-benign behavior may include receiving behavior information from the mobile device in response to sending the generated artificial attack software to the mobile device, generating a behavior vector based on the received behavior information, applying the generated behavior vector to a classifier model to generate a result, and using the generated result to determine whether the behavior-based security system of the mobile device responded adequately to the simulated non-benign behavior.

In a further aspect, the method may include setting a timer by the processor, wherein determining whether the behavior-based security system of the mobile device responded adequately to the simulated non-benign behavior may include determining whether a response was received from the mobile device before an expiration of the timer. In a further aspect, the method may include sending a system-ok signal to the mobile device in response to determining that the behavior-based security system of the mobile device responded adequately to the simulated non-benign behavior.

In a further aspect, sending the dead-man signal to the mobile device may include sending a communication message that includes information suitable for causing the mobile device to perform dead-man operations. In a further aspect, generating the artificial attack software may include receiving a corpus of behavior information from many mobile devices, analyzing the corpus of behavior information to identify a non-benign behavior, and generating the artificial attack software to include information suitable for causing a mobile device processor of the mobile device to perform the identified non-benign behavior. In a further aspect, generating the artificial attack software may include generating artificial attack software that includes information suitable for causing a mobile device processor of the mobile device to perform operations associated with a known cyber attack.

In a further aspect, the method may include receiving the artificial attack software in a processor of the mobile device, installing the received artificial attack software in the mobile device, monitoring operations of the behavior-based security system to collect behavior information, and sending the collected behavior information to the processor. In a further aspect, the method may include setting by the mobile device processor a timer in response to sending the collected behavior information, and performing by the mobile device processor dead-man operations in response to determining that a system-ok signal has not been received by the mobile device and that the timer has expired. In a further aspect, the simulated non-benign behavior may include simulated behaviors of a near-field communication (NFC) transaction, and the dead-man signal may include information suitable for causing the mobile device to prevent completion of an near-field communication (NFC) transaction.

Further aspects include a computing device that includes a processor configured with processor-executable instructions to perform various operations, including generating artificial attack software configured to simulate a non-benign behavior in a mobile device, sending the generated artificial attack software to the mobile device so as to simulate the non-benign behavior in the mobile device, determining whether the behavior-based security system of the mobile device responded adequately to the simulated non-benign behavior, and sending a dead-man signal to the mobile device in response to determining that the behavior-based security system of the mobile device did not respond adequately to the simulated non-benign behavior.

In at aspect, the processor may be configured with processor-executable instructions to perform operations such that determining whether the behavior-based security system of the mobile device responded adequately to the simulated non-benign behavior includes receiving behavior information from the mobile device in response to sending the generated artificial attack software to the mobile device, generating a behavior vector based on the received behavior information, applying the generated behavior vector to a classifier model to generate a result, and using the generated result to determine whether the behavior-based security system of the mobile device responded adequately to the simulated non-benign behavior.

simulated non-benign behavior. In a further aspect, the processor may be configured with processor-executable instructions to perform operations that include setting a timer by the processor. In an aspect, the processor may be configured with processor-executable instructions to perform operations such that determining whether the behavior-based security system of the mobile device responded adequately to the simulated non-benign behavior includes determining whether a response was received from the mobile device before an expiration of the timer.

In a further aspect, the processor may be configured with processor-executable instructions to perform operations that include sending a system-link signal to the mobile device in response to determining that the behavior-based security system of the mobile device responded adequately to the simulated non-benign behavior. In a further aspect, the processor may be configured with processor-executable instructions to perform operations such that sending the dead-man signal to the mobile device includes sending a communication message that includes information suitable for causing the mobile device to perform dead-man operations.

In a further aspect, the processor may be configured with processor-executable instructions to perform operations such that generating the artificial attack software includes receiving a corpus of behavior information from many mobile devices, analyzing the corpus of behavior information to identify a non-benign behavior, and generating the artificial attack software to include information suitable for causing a mobile device processor of the mobile device to perform the identified non-benign behavior. In a further aspect, the processor may be configured with processor-executable instructions to perform operations such that generating the artificial attack software includes generating the artificial attack software to include information suitable for causing a mobile device processor of the mobile device to perform operations associated with a known cyber attack.

Further aspects include a non-transitory computer readable storage medium having stored thereon processor-executable software instructions configured to cause a processor to perform operations for analyzing a behavior-based security system of a mobile device, the operations including generating artificial attack software configured to simulate a non-benign behavior in a mobile device, sending the generated artificial attack software to the mobile device so as to simulate a non-benign behavior in the mobile device, determining whether the behavior-based security system of the mobile device responded adequately to the simulated non-benign behavior, and sending a dead-man signal to the mobile device in response to determining that the behavior-based security system of the mobile device did not respond adequately to the simulated non-benign behavior.

In a aspect, the stored processor-executable software instructions may be configured to cause a processor to perform operations such that determining whether the behavior-based security system of the mobile device responded adequately to the simulated non-benign behavior includes receiving behavior information from the mobile device in response to sending the generated artificial attack software to the mobile device, generating a behavior vector based on the received behavior information, applying the generated behavior vector to a classifier model to generate a result, and using the generated result to determine whether the behavior-based security system of the mobile device responded adequately to the simulated non-benign behavior.

In a further aspect, the stored processor-executable software instructions may be configured to cause a processor to perform operations that include setting a timer by the processor

wherein determining whether the behavior-based security system of the mobile device responded adequately to the simulated non-benign behavior includes determining whether a response was received from the mobile device before an expiration of the timer. In a further aspect, the stored processor-executable software instructions may be configured to cause a processor to perform operations that include sending a system-link signal to the mobile device in response to determining that the behavior-based security system of the mobile device responded adequately to the simulated non-benign behavior.

In a further aspect, the stored processor-executable software instructions may be configured to cause a processor to perform operations such that sending the dead-man signal to the mobile device includes sending a communication message that includes information suitable for causing the mobile device to perform dead-man operations. In a further aspect, the stored processor-executable software instructions may be configured to cause a processor to perform operations such that generating the artificial attack software includes receiving a corpus of behavior information from many mobile devices, analyzing the corpus of behavior information to identify a non-benign behavior, and generating the artificial attack software to include information suitable for causing a mobile device processor of the mobile device to perform the identified non-benign behavior. In a further aspect, the stored processor-executable software instructions may be configured to cause a processor to perform operations such that generating the artificial attack software includes generating the artificial attack software to include information suitable for causing a mobile device processor of the mobile device to perform operations associated with a known cyber attack.

Further aspects include a computing device that includes means for generating artificial attack software configured to simulate a non-benign behavior in a mobile device, means for sending the generated artificial attack software to the mobile device so as to simulate a non-benign behavior in the mobile device, means for determining whether the behavior-based security system of the mobile device responded adequately to the simulated non-benign behavior, and means for sending a dead-man signal to the mobile device in response to determining that the behavior-based security system of the mobile device did not respond adequately to the simulated non-benign behavior.

In an aspect, means for determining whether the behavior-based security system of the mobile device responded adequately to the simulated non-benign behavior includes means for receiving behavior information from the mobile device in response to sending the generated artificial attack software to the mobile device, means for generating a behavior vector based on the received behavior information, means for applying the generated behavior vector to a classifier model to generate a result, and means for using the generated result to determine whether the behavior-based security system of the mobile device responded adequately to the simulated non-benign behavior. In a further aspect, the computing device includes means for setting a timer by the processor, and the means for determining whether the behavior-based security system of the mobile device responded adequately to the simulated non-benign behavior include means for determining whether a response was received from the mobile device before an expiration of the timer.

In a further aspect, the computing device includes means for sending a system-link signal to the mobile device in response to determining that the behavior-based security system of the mobile device responded adequately to the simulated non-benign behavior. In a further aspect, means for

sending the deal-mail signed to the mobile device includes sending a communication message that includes information suitable for causing the mobile device to perform deal-mail operations. In a further aspect, means for generating the artificial attack software include means for receiving a corpus of behavior information from many mobile devices, means for analyzing the corpus of behavior information to identify a non-benign behavior, and means for generating the artificial attack software to include information suitable for causing a mobile device processor of the mobile device to perform the identified non-benign behavior.

BRIEF DESCRIPTION OF THE DRAWINGS

The accompanying drawings, which are incorporated herein and constitute part of this specification, illustrate exemplary aspects of the invention, and together with the general description given above and the detailed description given below, serve to explain the features of the invention.

FIG. 1 is a communication system block diagram illustrating network components of an example telecommunication system that is suitable for use with the various aspects.

FIG. 2 is a block diagram illustrating example logical components and information flows in an aspect computing device configured to use behavioral analysis and machine learning techniques to classify behaviors as benign or non-benign.

FIGS. 3 and 4 are process flow diagram illustrating methods of evaluating the accuracy and performance of a behavior-based security system of a computing device in accordance with the various aspects.

FIG. 5 is a process flow diagram illustrating a method of generating classifier models that are suitable for use in classifying device behaviors in accordance with the various aspects.

FIG. 6 is a process flow diagram illustrating a method for performing behavioral monitoring and analysis operations in accordance with an aspect.

FIG. 7 is a component block diagram of a mobile device suitable for use in an aspect.

FIG. 8 is a component block diagram of a server device suitable for use in an aspect.

DETAILED DESCRIPTION

The various aspects will be described in detail with reference to the accompanying drawings. Wherever possible, the same reference numerals will be used throughout the drawings to refer to the same or like parts. References made to particular examples and implementations are for illustrative purposes, and are not intended to limit the scope of the invention or the claims.

In overview, the various aspects include methods, and devices configured to implement the methods, for protecting a behavior-based security system of a computing device from malware and cyber attacks, and ensuring that the system is working correctly, efficiently, or as expected.

Computing devices (e.g., mobile or other resource-constrained computing devices) may be equipped with a behavior-based security system that is configured to use behavioral analysis and machine learning techniques to intelligently and efficiently identify, prevent, and/or correct the conditions, factors, and/or behaviors that often degrade a computing device's performance and/or power utilization levels over time. For example, the behavior-based security system may use behavioral analysis techniques to quickly and efficiently determine whether a software application, process, activity, or device behavior is benign or non-benign. The behavior-

based security system may then cause the computing device to perform various actions or operations to correct, heal, cure, isolate, or otherwise fix the identified problems (e.g., behaviors determined to be non-benign).

While the above-mentioned behavior-based security system is generally very effective for preventing the degradation in performance and power utilization levels of a computing device over time, a malicious software application might attempt to circumvent or evade detection by this system by altering, modifying, suppressing, uninstalling, stopping, or otherwise attacking the behavioral monitoring and analysis system of the device. To prevent such attacks, the various aspects may configure the computing device to work in conjunction with an attack simulation module to monitor the accuracy and performance of the behavior-based security system, and to determine whether the system is working correctly, efficiently, or as expected. In various aspects, the attack simulation module may include software, hardware, or a combination of hardware and software, and may be included in a network server, another computing device, a hypervisor running on the same computing device, or the computing device itself.

In the various aspects, the attack simulation module may be configured to generate and send artificial attack software to a computing device that is equipped with a behavior-based security system ("target device") to simulate an artificial attack on the target device. The target device may be configured to receive and install the artificial attack software, monitor the operations of the behavior-based security system to collect behavior information, and send the collected behavior information to the attack simulation module (or to a component or computing device that sent the artificial attack). The attack simulation module may receive and use the behavior information to determine whether the behavior-based security system in the target device correctly and adequately identified and responded to the artificial attack. In an aspect, this may be accomplished by generating behavior vectors based on the behavior information, and applying the behavior vectors to classifier models so as to determine whether the target device responded correctly, efficiently, or as expected to the artificial attack.

In addition to assessing the correct status and operation of the behavior-based security system, attack simulation module may also be configured to collect additional information (e.g., metadata on the operations and characteristics of the target device), and use this information to better evaluate the accuracy and performance of the behavior-based security system. For example, the attack simulation module may be configured to cause the target device to track the duration of time and/or energy consumed when evaluating or responding to the artificial attack. The target device may send this information along with the behavior information and/or analysis results (e.g., as part of the same communication, or in the same communication message, in a different communication message, etc.) to the attack simulation module, which may receive and use this additional information/metadata to better determine whether the target device responded efficiently or as expected to the artificial attack.

In the various aspects, the attack simulation module may be configured to determine that there has been a disruption to the target device's behavior-based security system in response to determining that the target device did not respond correctly, efficiently, or as expected to the artificial attack. That is, the network server may determine that there is a high probability that the behavior-based security system of the target device has been compromised or improperly modified, has been infected with malware, or is under attack in response to deter-

mining that the operations performed in the target device for the behaviors of the target device after the installation of the artificial attack software are not consistent with the specific behaviors expected from a similarly equipped device under similar attack conditions. The attack simulation module may also be configured to determine that there has been a disruption to the target device's behavior-based security system in response to determining that a suitable response was not received within a certain time period after the artificial attack software was sent to the target device.

In an aspect, the attack simulation module may be configured to send or transmit a "dead-man" signal to the target device in response to determining that there has been a disruption to the behavior-based security system of the target device. In an aspect, the dead-man signal may include information suitable for causing the target device to perform dead-man operations, which may include powering off the target device, rebooting the target device, or performing other operations to identify, prevent, and/or correct problems or behaviors of the behavioral monitoring and analysis system in the target device.

In an aspect, the attack simulation module may be configured to send or transmit a "system-OK" signal to the target device in response to determining that the target device's behavior-based security system is functioning normally, correctly, or as expected. In this aspect, the target device may be configured to set a timer for receiving a "system-OK" signal after sending the results of the analysis operations to the network server, and to perform the dead-man operations in response to determining that the timer has expired and a "system-OK" signal has not yet been received from the network server.

The word "exemplary" is used herein to mean "serving as an example, instance, or illustration." Any implementation described herein as "exemplary" is not necessarily to be construed as preferred or advantageous over other implementations.

The term "performance-degradation" is used herein to refer to a wide variety of undesirable operations and characteristics of a computing device, such as longer processing times, slower real-time responsiveness, lower battery life, loss of private data, malicious economic activity (e.g., sending unauthorized premium SMS messages, denial of service (DoS), poorly written or designed software applications, malicious software, malware, viruses, fragmented memory, operations relating to commanding the device or utilizing the device for spying or hidden activities, etc.). Also, behaviors, activities, and conditions that degrade performance for any of these reasons are referred to herein as "not benign" or "non-benign."

The terms "wireless device," "mobile device," "mobile computing device," and "user equipment" may be used generally and interchangeably herein to refer to any one or all of cellular telephones, smartphones, personal or mobile multimedia players, personal data assistants (PDAs), laptop computers, tablet computers, smartbooks, ultrabooks, palm-top computers, wireless electronic mail receivers, multimedia Internet enabled cellular telephones, wireless gaming controllers, and similar electronic devices which include a memory, a programmable processor for which performance is important, and operate under battery power such that power conservation methods are of benefit. While the various aspects are particularly useful in mobile devices, which are resource-constrained systems, the aspects are generally useful in any computing device that includes a processor and executes software applications.

Generally, the performance and power efficiency of a mobile device degrade over time. Recently, anti-virus companies (e.g., McAfee, Symantec, etc.) have begun marketing mobile anti-virus, firewall, and encryption products that aim to slow this degradation. However, many of these solutions rely on the periodic execution of a computationally-intensive scanning engine on the mobile device, which may consume many of the mobile device's processing and battery resources, slow or render the mobile device useless for extended periods of time, and/or otherwise degrade the user experience. In addition, these solutions are typically limited to detecting known viruses and malware, and do not address the multiple complex factors and/or interactions that often combine to contribute to a mobile device's degradation over time (e.g., when the performance degradation is not caused by viruses or malware). For these and other reasons, existing anti-virus, firewall, and encryption products do not provide adequate solutions for identifying the numerous factors that may contribute to a mobile device's degradation over time, for preventing mobile device degradation, or for efficiently restoring an aging mobile device to its original condition.

The variants aspects overcome these and other limitations of existing solutions by equipping computing devices (e.g., mobile devices) with a behavior-based security system, such as a comprehensive behavioral monitoring and analysis system, that is configured to use machine learning and/or behavioral analysis techniques to intelligently and efficiently identify, prevent, and/or correct the conditions, factors, and/or device behaviors that often degrade a mobile device's performance and/or power utilization levels over time.

In various aspects, the behavioral monitoring and analysis system may include an observer process, daemon module, or sub-system (herein collectively referred to as a "module"), a behavior extractor module, and an analyzer module. The observer module may be configured to instrument or coordinate various application programming interfaces (APIs), registers, counters or other components therein (herein collectively "instrumented components") at various levels of the computing device system. The observer module may continuously (or near continuously) monitor activities of the computing device by collecting behavior information from the instrumented components, which may be accomplished by reading information from API log files stored in a memory of the computing device.

The observer module may communicate (e.g., via a memory write operation, function call, etc.) the collected behavior information to the behavior extractor module, which may use the collected behavior information to generate behavior vectors that each represent or characterize many or all of the observed behaviors that are associated with a specific software application, module, component, task, or process of the mobile device. Each behavior vector may encapsulate one or more "behavior features." Each behavior feature may be an abstract number that represents all or a portion of an observed behavior. In addition, each behavior feature may be associated with a data type that identifies a range of possible values, operations that may be performed on those values, meanings of the values, etc. The data type may include information that may be used to determine how the feature value (or feature value) should be measured, analyzed, weighted, or used.

The behavior extractor module may communicate (e.g., via a memory write operation, function call, etc.) the generated behavior vectors to the analyzer module. The analyzer module may apply the behavior vectors to classifier models to

determine whether a software application or device behavior is benign or non-benign (e.g., malicious, poorly written, performance-degrading, etc.).

A classifier model may be a behavior model that includes data and/or information structures (e.g., feature vectors, behavior vectors, component lists, decision trees, decision nodes, etc.) that may be used by the computing device processor to evaluate a specific feature or aspect of the device's behavior. A classifier model may also include decision criteria for monitoring and/or analyzing a number of features, factors, data points, entries, APIs, states, conditions, behaviors, software applications, processes, operations, components, etc. (herein collectively referred to as "features") in the computing device.

A classifier model may be categorized as a full classifier model or a lean classifier model. A full classifier model may be a robust data model that is generated as a function of a large training dataset, which may include thousands of features and billions of entries. A lean classifier model may be a more focused data model that is generated from a reduced dataset that includes or prioritizes tests on the features entries that are most relevant for determining whether a particular mobile device behavior is non-benign. A locally generated lean classifier model is a lean classifier model that is generated in the computing device. As an example, a computing device may be configured to receive a full classifier model from a network server, generate a lean classifier model in the computing device based on the full classifier, and use the locally generated lean classifier model to classify a behavior of the device as being either benign or non-benign (i.e., malicious, performance-degrading, etc.).

The analyzer module may be configured to notify the observer module in response to determining that a device behavior is suspicious (i.e., in response to determining that the results of the analysis operations are not sufficient to classify the behavior as either benign or non-benign). In response, the observer module may adjust the granularity of its observations (i.e., the level of detail at which computing device features are monitored) and/or change the factors behaviors that are observed based on information received from the analyzer module (e.g., results of the real-time analysis operations generate or collect new or additional behavior information, and send the new additional information to the analyzer module for further analysis). Such feedback communications between the observer and analyzer modules enable the computing device processor to recursively increase the granularity of the observations (i.e., make finer or more detailed observations) or change the features behaviors that are observed until behavior is classified as either benign or non-benign, until a processing or battery consumption threshold is reached, or until the computing device processor determines that the source of the suspicious or performance-degrading computing device behavior cannot be identified from further increases in observation granularity. Such feedback communications also enable the computing device to adjust or modify the classifier models locally in the computing device without consuming an excessive amount of the computing device's processing, memory, or energy resources.

The above-mentioned behavioral monitoring and analysis system may allow a computing device to identify and react to performance-limiting and undesirable operating conditions without consuming an excessive amount of its processing, memory, or energy resources. This is particularly useful in complex and resource constrained systems, such as mobile computing devices which have relatively limited processing, memory, and energy resources.

In mobile computing devices there are a large variety of factors that may contribute to the degradation in performance and power utilization levels of such devices over time, including poorly written or designed software applications, malware viruses, fragmented memory, background processes, etc. Due to the number, variety, and complexity of these factors, it is often not feasible to evaluate all of the factors that may contribute to the degradation in performance and/or power utilization levels of the complex yet resource-constrained systems of modern mobile computing devices. As such, it is difficult for users, operating systems, and/or application programs (e.g., anti-virus software, etc.) to accurately and efficiently identify the sources of such problems. As a result, mobile device users have few remedies for preventing the degradation in performance and power utilization levels of a mobile device over time, or for restoring an aging mobile device to its original performance and power utilization levels.

To provide better performance in view of these facts, the mobile device may be equipped with the above-described behavioral monitoring and analysis system so that it may quickly determine whether a particular mobile device behavior, condition, sub-system, software application, or process is benign or non-benign without consuming an excessive amount of its processing, memory, or energy resources.

While the above-described behavioral monitoring and analysis system is generally very effective for identifying non-benign behaviors, such as those caused by malware and cyber attacks, the system is itself susceptible to attacks and malware. A malicious software application may circumvent or evade detection by the behavioral monitoring and analysis system by altering, modifying, suppressing, uninstalling, disabling, stopping, or otherwise attacking the various components or modules (e.g., observer, analyzer, etc.) that together provide the comprehensive behavioral monitoring and analysis system. A behavioral monitoring and analysis system might also be compromised by a software update or removal of a software or operating system module that is made without malicious intent.

To identify, prevent, and/or respond to malware, attacks or other events compromising the behavioral monitoring and analysis system, the mobile device may be configured to work in conjunction with a network server (or another computing device, a component in the same computing device, etc.) to monitor the accuracy and performance of the behavioral monitoring and analysis system of the mobile device, and determine whether the system is working correctly, effectively, or unexpected. The network server may be configured to generate and send an attack software to the mobile device and listen for a response from the mobile device within an expected response time. If a suitable response is not received from the mobile device within the expected response time, the network server may determine there has been a disruption to the device's behavioral monitoring and analysis system, which may indicate that the system has been compromised, has been infected with malware, is experiencing an attack, or is otherwise not functioning correctly or as expected.

In response to determining that the behavioral monitoring and analysis system has been compromised, the network server may transmit a "dead-man" signal to the mobile device to cause that device to perform various operations to identify, fix, or respond to problems associated with its behavioral monitoring and analysis system. The "dead-man" signal may also cause the mobile device to send a text message, voice message, email, or other types of notification to the user to notify that user of the attack. If the mobile device includes a

service operation system, the "dead-man" signal may also cause the device to capture and send an image or screenshot to the network server for analysis.

The mobile device may be configured to receive and install artificial attack software from the network server (so as to simulate an artificial attack on the mobile device), monitor the operations of the behavioral monitoring and analysis system to collect behavior information and/or analysis results, and send the collected information to the network server (so as to the computing device that sent the artificial attack). The network server may use the information received from the mobile device to determine whether the behavioral monitoring and analysis system was able to correctly adequately identify and respond to the artificial attack. For example, the network server may compare apply the received information to classifier models to determine whether the mobile device responded correctly, efficiently, or as expected to the artificial attack.

The network server may determine that the mobile device's behavioral monitoring and analysis system has been compromised in response to determining that the system did not respond correctly, efficiently, or as expected to the artificial attack. The network server may transmit a "dead-man" signal configured to cause the mobile device to perform dead-man operations in response to determining that there has been a disruption to the mobile device's behavioral monitoring and analysis system.

In an aspect, the network server may also be configured to transmit a "system-OK" signal to the mobile device in response to determining that the target device's behavioral monitoring and analysis system is functioning normally, correctly, or as expected. In this aspect, the mobile device may be configured to set a timer for receiving a "system-OK" signal after sending the results of the analysis operations to the network server, and to perform the dead-man operations in response to determining that the timer has expired and a "system-OK" signal has not yet been received from the network server.

In various aspects, the network server may be configured to generate the artificial attack software to include executable code or scripts that cause the mobile device to perform various operations associated with known malware and cyber attacks, or operations that are known cause a well-defined reaction in properly functioning behavioral monitoring and analysis systems. In an aspect, the network server may generate artificial attack software so that it simulates behaviors that are associated with a known cyber attack, excluding the operations that are harmful to the mobile device or its user. In another aspect, the network server may be configured to generate artificial attack software so that it simulates a real attack (including harmful operations) or a non-benign behavior. In this aspect, the network server may set a timer for the expected response time so that the dead-man signal is transmitted to the mobile device before the simulated attack causes damage to the device.

In various aspects, the network server may be configured to generate artificial attack software based on information received from an antivirus server, a cloud computing device, and/or another network server. The network server may also generate artificial attack software based on behavior information and/or results of behavior analyses provided by many mobile devices. For example, the network server may crowd source, combine, amalgamate, or correlate the analysis results received from many mobile devices to identify non-benign behaviors, and generate the artificial attack software to simulate the identified non-benign behaviors.

In various aspects, the network server may be configured to send artificial attack software to the mobile device repeatedly, periodically, at set intervals, randomly, pseudo-randomly, etc. The network server may be configured to generate many different types of artificial attack software, each of which simulates a different type of attack. The network server may send different type of the artificial attack software to the same mobile device so as to simulate different types of attacks in the same device.

In an aspect, the mobile device may be configured to perform any or all of the above-mentioned operations as part of its normal behavioral monitoring and analysis operations. For example, the mobile device may receive and install the attack software as part of its normal operations for receiving software updates or classifiers from the network server, and report the results of the analysis operations to the network server as part of its regular reporting operations. In another aspect, the mobile device may be equipped with an artificial attack module that is configured to operate independent of the other modules of the comprehensive behavioral monitoring and analysis system. The artificial attack module may be configured to focus its operations on simulating attacks and reporting results to the network server.

In an aspect, the mobile device may be configured to receive the artificial attack software from another mobile device and/or via a peer-to-peer communication link. For example, each of a plurality of mobile devices may be configured to perform behavior observation and analysis operations to identify non-benign behaviors, generate artificial attack software based on one or more of the identified non-benign behaviors, send the generated artificial attack software to the mobile device, set a timer, listen for a response, transmit a dead-man or system-OK signal, and/or perform any or all of the operations of the network server discussed above.

In a peer, mobile devices may be configured to establish or join a trust network that includes a plurality of pre-screened or trusted mobile devices. In various aspects, establishing or joining a trust network may include each mobile device performing group formation operations that include establishing communication links to the other mobile devices another via peer-to-peer, WiFi-Direct, or other similar technologies. Mobile devices may also be connected via a shared secure network, enterprise virtual private network, and other similar technologies or group classifications. In an aspect, a trusted network may include mobile devices that are the same network or which have direct communication links. In an aspect, each mobile device in a trust network may be configured to send and receive artificial attack software, as well as behavior information and analysis results, to and from any or all of the other mobile devices in that trust network.

In various aspects, the operations discussed above with reference to the network server and/or attack simulation module may instead be performed by the computing device that includes the behavioral monitoring and analysis system being evaluated. For example, the operations of the network server attack simulation module may be performed by a service or server (or guest server) that runs on a hypervisor or virtual machine monitor (VMM) of the computing device that includes the behavioral monitoring and analysis system. As another example, the operations of the network server attack simulation module may be implemented via software that runs on the same operating system as the behavioral monitoring and analysis system being evaluated. In an aspect, the computing device may also include a hypervisor or VMM that is configured to manage or guarantee the correct behaviors or operations of the service or server that performs the operations discussed above.

The various aspects may be implemented within a variety of communication systems, such as the example communication system 100 illustrated in FIG. 1. A typical cell telephone network 104 includes a plurality of cell base stations 106 coupled to a network operations center 108, which operates to connect voice calls and data between mobile devices 102 (e.g., cell phones, laptops, tablets, etc.) and other network destinations, such as via telephone land lines (e.g., a POTS network, not shown) and the Internet 110. Communications between the mobile devices 102 and the telephone network 104 may be accomplished via two-way wireless communication links 112, such as 4G, 3G, CDMA, TDMA, UMTS and/or other cell telephone communication technologies. The telephone network 104 may also include one or more servers 114 coupled in or within the network operations center 108 that provide a connection to the Internet 110.

The communication system 100 may further include network servers 116 connected to the telephone network 104 and to the Internet 110. The connection between the network servers 116 and the telephone network 104 may be through the Internet 110 or through a private network (as illustrated by the dashed arrow). A network server 116 may also be implemented as a server within the network infrastructure of a cloud service provider network 118. Communication between the network server 116 and the mobile devices 102 may be achieved through the telephone network 104, the Internet 110, private network (not illustrated), or any combination thereof.

The network server 116 may be configured to receive information on various conditions, features, behaviors, and corrective actions from many mobile devices 102 or a central database or cloud service provider network 118, and use this information to generate data, algorithms, classifiers, or behavior models (herein collectively "classifier models") that include data and/or information structures (e.g., feature vectors, behavior vectors, component lists, etc.) that may be used by a processor of a computing device to evaluate a specific aspect of a mobile device's behavior.

In an aspect, the network server 116 may be configured to send artificial attack software to the mobile device 102, set a response timer, receive a response message from the mobile device 102, determine whether the response message was received before the expiration of the timer, compare the information included in the received response message to one or more classifier models, and use the results of the comparison to determine whether a behavioral monitoring and analysis system of the mobile device 102 correctly identified and responded to the attack. The network server 116 may be configured to transmit a "system-OK" signal to the mobile device in response to determining that response message was received before the expiration of the timer and/or that the information included in the received response message indicates that the mobile device 102 correctly identified and responded to the artificial attack. The network server 116 may also be configured to transmit a "dead-man" signal to the mobile device 102 in response to determining that response message was not received before the expiration of the timer, or that the information included in the received response message indicates that the mobile device 102 did not correctly identify or respond to the artificial attack.

In an aspect, the network server 116 may be configured to generate a full classifier model. The network server 116 may be configured to use the full classifier models to analyze or classify behaviors of the mobile device 102 and/or the behaviors of the behavioral monitoring and analysis system of the mobile device 102. The network server 116 may also be configured to send the full classifier models to the mobile

device 102. In an aspect, the network server 116 may be configured to generate the full classifier model to include all or most of the features, data points, and/or factors that could contribute to the degradation of any of a number of different makes, models, and configurations of mobile devices 102. In various aspects, the network server may be configured to generate the full classifier model to describe or express a large corpus of behavior information as a finite state machine, decision nodes, decision trees, or in any informative structure that can be modified, edited, augmented, or otherwise used to quickly and efficiently generate learner classifier models.

The mobile device 102 may be configured to receive and install the artificial attack software, monitor the operations of the behavioral monitoring and analysis system, and send the results of the operations performed by the system back to the network server or the computing device that sent the artificial attack. The mobile device 102 may also be configured to receive a full classifier model from the network server 116, and use the received full classifier model to monitor, analyze, and/or classify the behaviors of the mobile device 102. The mobile device 102 may be further configured to use the full classifier model to generate more focused classifier models that account for the specific features and functionalities of the software applications of the mobile device 102. For example, the mobile device 102 may generate application-specific and/or application-type-specific classifier models (i.e., data or behavior models) that preferentially or exclusively identify or evaluate the conditions or features of the mobile device that are relevant to a specific software application or to a specific type of software application (e.g., games, navigation, finance, etc.) that is installed on the mobile device 102 or stored in a memory of the mobile device 102. The mobile device 102 may use these locally generated classifier models to perform real-time behavior monitoring and analysis operations.

In various aspects, the network server 116 and/or mobile device 102 may include an attack simulation module, which may be a thread, process, daemon, module, sub-system, or component that is implemented in software, hardware, or a combination thereof. In addition, the attack simulation module may be implemented within parts of the operating system (e.g., within the kernel, in the kernel space, in the user space, etc.) or within separate programs or applications, in specialized hardware buffers or processors, or any combination thereof. In an aspect, all or portions of the attack simulation module may be implemented as software instructions executing on one or more processors or processing cores of the mobile device 102 and/or network server 116.

FIG. 2 illustrates example logical components and information flow in an aspect computing device that includes a comprehensive behavioral monitoring and analysis system 200 configured to use behavioral analysis techniques to identify and respond to non-human device behaviors. In the example illustrated in FIG. 2, the computing device is a mobile device 102 that includes a device processor (i.e., mobile device processor), configured with executable instruction modules that include a behavior observer module 212, a behavior extractor module 204, a behavior analyzer module 216, and an actuator module 208. Each of the modules 202-208 may be a thread, process, daemon, module, sub-system, or component that is implemented in software, hardware, or a combination thereof. In various aspects, the modules 202-210 may be implemented within parts of the operating system (e.g., within the kernel, in the kernel space, in the user space, etc.) or within separate programs or applications, in specialized hardware buffers or processors, or any combination thereof. In an aspect, one or more of the modules 202-208 may be

implemented as software instructions executing on one or more processors of the mobile device 102.

The device processor may also be configured to receive artificial attack software from a network server 116, install the artificial attack software, monitor the operations, accuracy, and performance of the various modules 202-208 of behavioral monitoring and analysis system 200, generate reports that characterize the operations the system 200 and/or the modules 202-208, and send the generated reports to the network server 116. The device processor may also be configured to set a timer, listen for “system-OK” signal after sending the reports to the network server 116, and perform dead-man operations if an “system-OK” signal is not received when the timer expires. In addition, the device processor may be configured listen for a “hand-man” signal indicating that there has been a disruption in the behavioral monitoring and analysis system, and to perform dead-man operations in response to receiving a “dead-man” signal.

The behavior observer module 202 may be configured to instrument application programming interfaces (APIs) at various levels/modules of the device, and monitor the activities, conditions, operations, and events (e.g., system events, state changes, etc.) at the various levels/module over a period of time via the in-strommed APIs. The behavior observer module 202 may collect behavior information pertaining to the monitored activities, conditions, operations, or events, and store the collected information in a memory (e.g., in a log file, etc.). The behavior observer module 202 may then communicate (e.g., via a memory write operation, function call, etc.) the collected behavior information to the behavior extractor module 204.

The behavior extractor module 204 may be configured to receive or refine the collected behavior information, and use this information to generate one or more behavior vectors. In an aspect, the behavior extractor module 204 may be configured to generate the behavior vectors to include a concise definition of the observed behaviors. For example, each behavior vector may succinctly describe observed behavior of the mobile device, software application, or process in a value or vector data-structure (e.g., in the form of a string of numbers, etc.). The behavior extractor module 204 may also be configured to generate the behavior vectors so that they function as an identifier that enables the mobile device ecosystem (e.g., the behavior analyzer module 206) to quickly recognize, identify, and/or analyze mobile device behaviors.

The behavior analyzer module 206 may be configured to apply the behavior vectors to classifier modules to determine if a device behavior is a non-benign behavior that is contributing to (or are likely to contribute to) the device’s degradation over time and/or which may otherwise cause problems on the device. The behavior analyzer module 206 may notify the actuator module 208 that actuator behavior is not benign. In response, the actuator module 208 may perform various actions or operations to heal, cure, isolate, or otherwise fix identified problems. For example, the actuator module 208 may be configured to terminate a software application or process when the result of applying the behavior vector to the classifier model (e.g., by the classifier module) indicates that a software application or process is not benign.

The behavior observer module 202 may be configured to monitor the activities of the mobile computing device 102. In various aspects, this may be accomplished by monitoring various software and hardware components of the mobile computing device 102 and collecting information pertaining to the communications, transactions, events, or operations of the monitored and measurable components that are associated with the activities of the mobile computing device 102.

Such activities include a software application’s performance of an operation or task, a software application’s execution in a processing core of the mobile computing device 102, the execution of process, the performance of a task or operation, a device behavior, the use of a hardware component, etc.

In various aspects, the behavior observer module 202 may be configured to monitor the activities of the mobile computing device 102 by collecting information pertaining to library API calls in an application framework or run-time libraries, system call, APIs, file-system and networking sub-system operations, device (including sensor devices) state changes, and other similar events. In addition, the behavior observer module 202 may monitor file system activity, which may include searching for filenames, categories of file accesses (operational information about files), creating or deleting files (e.g., type .exe, .zip, .cav, file read/write seek operations, changing file permissions, etc).

The behavior observer module 202 may also monitor the activities of the mobile computing device 102 by monitoring data network activity, which may include types of connections, protocols, port numbers, server check that the device is connected to, the number of connections, volume or frequency of communications, etc. The behavior observer module 202 may monitor phone network activity, which may include monitoring the type and number of calls or messages (e.g., SMS, etc.) sent out, received, or intercepted (e.g., the number of premium calls placed).

The behavior observer module 202 may also monitor the activities of the mobile computing device 102 by monitoring the system resource usage, which may include monitoring the number of forks, memory access operations, number of files open, etc. The behavior observer module 202 may monitor the state of the mobile computing device 102, which may include monitoring various factors, such as whether the display is on or off, whether the device is locked or unlocked, the amount of battery remaining, the state of the camera, etc. The behavior observer module 202 may also monitor inter-process communications (IPC) by, for example, monitoring attempts to connect services (browser, contacts-provider, etc.), the degree of inter-process communications, pop-up windows, etc.

The behavior observer module 202 may also monitor the activities of the mobile computing device 102 by monitoring driver statistics and/or the status of one or more hardware components, which may include cameras, sensors, electronic displays, WiFi communication components, data controllers, memory controllers, system controllers, access ports, timers, peripheral devices, wireless communication components, external memory chips, voltage regulators, oscillators, phases, locked loops, peripheral bridges, and other similar components used to support the processors and clients running on the mobile computing device 102.

The behavior observer module 202 may also monitor the activities of the mobile computing device 102 by monitoring one or more hardware counters that denote the state or status of the mobile computing device 102 and/or computing device subsystems. A hardware counter may include a special-purpose register of the processor cores that is configured to store a count value or state of hardware-related activities or events occurring in the mobile computing device 102.

The behavior observer module 202 may also monitor the activities of the mobile computing device 102 by monitoring the actions or operations of software applications, software downloads, from an application download server (e.g., Apple® App Store server), computing device information used by software applications, call information, text messaging information (e.g., SendSMS, BlockSMS, ReadSMS,

etc.), media messaging information (e.g., Receipts/MSNs), user account information, location information, camera information, accelerometer information, browser information, content of browser-based communications, content of voice-based communications, short range radio communications (e.g., Bluetooth, WiFi, etc.), content of text-based communications, content of recorded audio files, phonebook/contact information, contacts lists, etc.

The behavior observer module 202 may also monitor the activities of the mobile computing device 102 by monitoring transmissions or communications of the mobile computing device 102, including communications that include voice-over (VoiceOver/Comm), device identifiers (DeviceID/Comm), user account information (UserAccount/Comm), calendar information (Calendar/Comm), location information (Location/Comm), recorded audio information (Record Audio/Comm), accelerometer information (Accelerometer/Comm), etc.

The behavior observer module 202 may also monitor the activities of the mobile computing device 102 by monitoring the usage of, and updates changes to, compass information, computing device settings, battery life, gyroscope information, pressure sensors, magnet sensors, screen activity, etc. The behavior observer module 202 may monitor notifications communicated to and from a software application (AppNotifications), application updates, etc. The behavior observer module 202 may monitor conditions or events pertaining to a first software application requesting the download and/or install of a second software application. The behavior observer module 202 may monitor conditions or events pertaining to user verification, such as the entry of a password, etc.

The behavior observer module 202 may also monitor the activities of the mobile computing device 102 by monitoring conditions or events at multiple levels of the mobile computing device 102, including the application level, radio level, and sensor level. Application level observations may include observing the user via facial recognition software, observing social streams, observing names entered by the user, observing events pertaining to the use of PassBook®, Google® Wallet, Baypack, and other similar applications or services. Application level observations may also include observing events relating to the use of virtual private networks (VPNs) and events pertaining to synchronization, voice searches, voice control (e.g., lock unlock a phone by saying one word), language translators, the offloading of data for computations, video streaming, camera usage without user activity, microphone usage without user activity, etc.

Radio level observations may include determining the presence, existence or amount of any or more of user interaction with the mobile computing device 102 before establishing radio communication links or transmitting information, dual multiple subscriber identification module (SIM) cards, Internet radio, mobile phone tethering, offloading data for computations, device state communications, device as a game controller or home controller, vehicle communications, computing device synchronization, etc. Radio level observations may also include monitoring the use of radios (Wi-Fi, WiMax, Bluetooth, etc.) for positioning, peer-to-peer (p2p) communications, synchronization, vehicle to vehicle communications, and machine-to-machine (m2m). Radio level observations may further include monitoring network traffic usage, statistics, or profiles.

Sensor level observations may include monitoring a magnet sensor or other sensor to determine the usage and/or external environment of the mobile computing device 102 (for example, the computing device processor may be config-

ured to determine whether the device is in a holster (e.g., via a magnet sensor configured to sense a magnet within the holster or in the user's pocket (e.g., via the amount of light detected by a camera or light sensor)). Detecting that the mobile computing device 102 is in a holster may be relevant to recognizing suspicious behaviors, for example, because activities and motions related to active usage by a user (e.g., taking photographs or videos, sending messages, conducting a voice call, recording sounds, etc.) occurring while the mobile computing device 102 is holstered could be signs of nefarious processes executing on the device (e.g., to track or spy on the user).

Other examples of sensor level observations related to usage or external environments may include, detecting NFC signaling, collecting information from a credit card scanner, barcode scanner, or mobile tag reader, detecting the presence of a Universal Serial Bus (USB) power charging source, detecting that a keyboard or auxiliary device has been coupled to the mobile computing device 102, detecting that the mobile computing device 102 has been coupled to another computing device (e.g., via USB, etc.), determining whether an LED, flash, flashlight, or light source has been modified or disabled (e.g., inadvertently disabling an emergency signaling app, etc.), detecting that a speaker or microphone has been turned on or powered, detecting a charging or power event, detecting that the mobile computing device 102 is being used as a game controller, etc. Sensor level observations may also include collecting information from medical or healthcare sensors or from sensors on the user's body, collecting information from an external sensor plugged into the USB audio jack, collecting information from a tactile or haptic sensor (e.g., via a haptic interface, etc.), collecting information pertaining to the thermal state of the mobile computing device 102, etc.

To reduce the number of factors monitored to a manageable level, in an aspect, the behavior observer module 202 may be configured to perform coarse observations by monitoring an initial set of behaviors or factors that are a small subset of all factors that could contribute to the computing device's degradation. In an aspect, the behavior observer module 202 may receive the initial set of behaviors and/or factors from a server and/or a component in a cloud service. In an aspect, the initial set of behaviors/factors may be specified in machine learning classifier models.

Each classifier model may be a behavior model that includes data and/or information structures (e.g., feature vectors, behavior vectors, component lists, etc.) that may be used by a computing device processor to evaluate a specific feature or aspect of a computing device's behavior. Each classifier model may also include decision criteria for monitoring a number of features, factors, data points, entries, APIs, states, conditions, behaviors, applications, processes, operations, components, etc. (herein collectively "features") in the computing device. The classifier models may be pre-installed on the computing device, downloaded or received from a network server, generated in the computing device, or any combination thereof. The classifier models may be generated by using crowd sourcing solutions, behavior modeling techniques, machine learning algorithms, etc.

Each classifier model may be categorized as a full classifier model or a lean classifier model. A full classifier model may be a robust data model that is generated as a function of a large training data-set, which may include thousands of features and billions of entries. A lean classifier model may be a more focused data model that is generated from a reduced data-set that includes tests only the features/entries that are most rel-

except for determining whether a particular activity is an ongoing critical activity, and/or whether a particular computing device behavior is not benign.

A locally generated lean classifier model is a lean classifier model that is generated in the computing device. An application-specific classifier model is a classifier model that includes a focused data model that includes tests only the features entries that are most relevant for evaluating a particular software application. A device-specific classifier model is a classifier model that includes a focused data model that includes tests only computing device-specific features entries that are determined to be most relevant to classifying an activity or behavior in a specific computing device.

The behavior analyzer module 206 may be configured to apply the behavior vectors generated by the behavior extractor module 204 to a classifier model to determine whether a monitored activity (or behavior) is benign, suspicious, or non-benign. In an aspect, the behavior analyzer module 206 may classify a behavior as "suspicious" when the results of its behavioral analysis operations do not provide sufficient information to classify the behavior as either benign or non-benign.

The behavior analyzer module 206 may be configured to notify the behavior observer module 202 in response to determining that a monitored activity or behavior is suspicious. In response, the behavior observer module 202 may adjust the granularity of its observations (i.e., the level of detail at which computing device features are monitored) and/or change the features behaviors that are observed based on information received from the behavior analyzer module 206 (e.g., results of the real-time analysis operations), generate or collect new or additional behavior information, and send the new additional information to the behavior analyzer module 206 for further analysis/classification. Such feedback communications between the behavior observer module 202 and the behavior analyzer module 206 enable the mobile computing device 102 to recursively increase the granularity of the observations (i.e., make finer or more detailed observations) or change the features behaviors that are observed until an activity is classified, a source of a suspicious or performance-degrading computing device behavior is identified, until a processing or battery consumption threshold is reached, until the computing device processor determines that the source of the suspicious or performance-degrading computing device behavior cannot be identified from further increases in observation granularity. Such feedback communication also enable the mobile computing device 102 to adjust or modify the classifier models locally in the computing device without consuming an excessive amount of the computing device's processing, memory, or energy resources.

In an aspect, the behavior observer module 202 and the behavior analyzer module 206 may provide, either individually or collectively, real-time behavior analysis of the computing system's behaviors to identify suspicious behavior from limited and coarse observations, to dynamically determine behaviors to observe in greater detail, and to dynamically determine the level of detail required for the observations. This allows the mobile computing device 102 to efficiently identify and prevent problems without requiring a large amount of processor, memory, or battery resources on the device.

In various aspects, the device processor may be configured to monitor, analyze, and/or classify activities or behaviors by identifying a critical data resource that requires close monitoring, identifying an intermediate resource associated with the critical data resource, monitoring API calls made by a

software application when accessing the critical data resource and the intermediate resource, identifying computing device resources that are consumed or produced by the API calls, identifying a pattern of API calls as being indicative of non-benign activity by the software application, generating a light-weight behavior signature based on the identified pattern of API calls and the identified computing device resources, using the light weight behavior signature to perform behavior analysis operations, and determining whether the software application is benign or not benign based on the behavior analysis operations.

In various aspects, the device processor may be configured to monitor, analyze, and/or classify activities or behaviors by identifying APIs that are used most frequently by software applications executing on the computing device, storing information regarding usage of identified hot APIs in an API log in a memory of the computing device, and performing behavior analysis operations based on the information stored in the API log to identify behaviors that are inconsistent with normal operation patterns. In an aspect, the API log may be generated so that it is organized such that the values of generic fields that remain the same across invocations of an API are stored in a separate table, the values of specific fields that are specific to each invocation of the API. The API log may also be generated so that the values of the specific fields are stored in a table along with hash keys to the separate table that stores the values of the generic fields.

In various aspects, the device processor may be configured to monitor, analyze, and/or classify activities or behaviors by receiving from a server a full classifier model that includes a finite state machine that is suitable for conversion or expression as a plurality of boosted decision stumps, generating a lean classifier model in the computing device based on the full classifier, and using the lean classifier model in the computing device to classify the activities or behaviors as being either benign or not benign (i.e., malicious, performance degrading, etc.). In an aspect, generating the lean classifier model based on the full classifier model may include determining a number of unique test conditions that should be evaluated to classify an activity or behavior without consuming an excessive amount of processing, memory, or energy resources of the computing device, generating a list of test conditions by sequentially traversing the list of boosted decision stumps and inserting the test condition associated with each sequentially traversed boosted decision stump into the list of test conditions until the list of test conditions includes the determined number of unique test conditions, and generating the lean classifier model to include only those boosted decision stumps that test one of a plurality of test conditions included in the generated list of test conditions.

In various aspects, the device processor may be configured to monitor, analyze, and/or classify activities or behaviors by using device-specific information, such as capability and state information, of the computing device to identify device-specific test conditions in a plurality of test conditions that are relevant to classifying a behavior on the computing device, generating a lean classifier model that includes only the identified computing device-specific test conditions, and using the generated lean classifier model in the computing device to classify the behavior of the computing device. In an aspect, the lean classifier model may be generated to include only decision nodes that evaluate a computing device feature that is relevant to a current operating state or configuration of the computing device. In an aspect, generating the lean classifier model may include determining a number of unique test conditions that should be evaluated to classify the behavior without consuming an excessive amount of computing

device's resources, e.g., processing, memory, air, energy resources), generating a list of test conditions by sequentially traversing the plurality of test conditions in the full classifier model, inserting those test conditions that are relevant to classifying the behavior of the computing device into the list of test conditions until the list of test conditions includes the determined number of unique test conditions, and generating the lean classifier model to include decision nodes included in the full classifier model that test one of the conditions included in the generated list of test conditions.

In various aspects, the device processor may be configured to monitor, analyze, and/or classify activities or behaviors by monitoring an activity of a software application or process, determining an operating system execution state of the software application process, and determining whether the activity is a critical activity based on the operating system execution state of the software application or process during which the activity was monitored. In an further aspect, the device processor may determine whether the operating system execution state of the software application or process is relevant to the activity, generate a shadow feature vector that identifies the operating system execution state of the software application or process during which the activity was monitored, generate a behavior vector that associates the activity with the shadow feature value identifying the operating system execution state, and use the behavior vector to determine whether the activity is a critical activity or not benign.

In various aspects, the device processor may be configured to monitor, analyze, and/or classify activities or behaviors by monitoring an activity of a software application or process, determining an application-and-operating-system-agnostic execution state of the software application process, and determining whether the activity is a critical activity or not benign based on the activity and in the application-and-operating-system-agnostic execution state of the software application during which the activity was monitored. In an further aspect, the device processor may determine whether the application-and-operating-system-agnostic execution state of the software application is relevant to the activity, and generate a behavior vector that associates the activity with the application-and-operating-system-agnostic execution state, and use the behavior vector to determine whether the activity is a critical activity or not benign. The device processor may also use the application-and-operating-system-agnostic execution state to select a classifier model (e.g., application-specific classifier model), and apply the behavior vector to the selected classifier model to determine whether the activity is a critical activity or not benign.

In various aspects, the device processor may be configured to work in conjunction with a network server to intelligently and efficiently identify the features, factors, and data points that are most relevant to determining whether an activity is a critical activity and not benign. For example, the device processor may be configured to receive a full classifier model from the network server, and use the received full classifier model to generate lean classifier models (i.e., data behavior model(s) that are specific for the features and functionalities of the computing device or the software applications of the computing device). The device processor may use the full classifier model to generate a family of lean classifier models of varying levels of complexity (or "leanness"). The leanest family of lean classifier models (i.e., the lean classifier model based on the fewest number of test conditions) may be applied routinely until a behavior is encountered that the model cannot categorize as either benign or not benign and therefore is categorized by the model as suspicious, at which time a more robust (i.e., less lean) lean classifier model may

be applied in an attempt to categorize the behavior. The application of ever more robust lean classifier models within the family of generated lean classifier models may be applied until a definitive classification of the behavior is achieved. In this manner, the observer and/or analyzer modules can strike a balance between efficiency and accuracy by limiting the use of the most complex, but resource-intensive lean classifier models in those situations where a robust classifier model is needed to definitively classify a behavior.

In various aspects, the device processor may be configured to generate one or more lean classifier models by converting a finite state machine representation expression into boosted decision stumps, pruning or cutting the full set of boosted decision stumps based on computing device-specific states, features, behaviors, conditions, or configurations to include a subset or subsets of boosted decision stumps included in the full classifier model, and using the subset or subsets of boosted decision stumps to intelligently monitor, analyze and/or classify a computing device behavior.

Boosted decision stumps are one level decision trees that have exactly one node (and thus one test question or test condition) and a weight value, and thus are well suited for use in a binary classification of data behavior. That is, applying whether a vector to boosted decision stump results in binary answer (e.g., Yes or No). For example, if the question condition tested by a boosted decision stump is "Is the frequency of Short Message Service (SMS) transmission less than x per minute," applying a value of "3" to the boosted decision stump will result in either a "yes" answer (i.e., "less than 3" SMS transmissions) or a "no" answer for "3 or more" SMS transmissions).

Boosted decision stumps are efficient because they are very simple and primal (and thus do not require significant processing resources). Boosted decision stumps are also very parallelizable, and thus many stumps may be applied or tested in parallel at the same time (e.g., by multiple cores or processors in the computing device).

In an aspect, the device processor may be configured to generate a lean classifier model that includes a subset of classifier criteria included in the full classifier model and only those classifier criteria corresponding to the features relevant to the computing device configuration, functionality, and connected included hardware. The device processor may use this lean classifier model(s) to monitor only those features and functions present and relevant to the device. The device processor may then periodically modify or regenerate the lean classifier model(s) to include or remove various features and corresponding classifier criteria based on the computing device's current state and configuration.

As an example, the device processor may be configured to receive a large boosted-decision-stumps-classifier model that includes decision stumps associated with a full feature set of behavior models (e.g., classifiers), and derive one or more lean classifier models from the large classifier models by selecting only features from the large classifier models that are relevant to the computing device's current configuration, functionality, operating state and/or connected included hardware, and including in the lean classifier model a subset of boosted decision stumps that correspond to the selected features. In this aspect, the classifier criteria corresponding to features relevant to the computing device may be those boosted decision stumps included in the large classifier model that test at least one of the selected features. The device processor may then periodically modify or regenerate the boosted decision stumps-lean classifier model(s) to include or remove various features based on the computing device's current state and configuration so that the lean classifier

model continues to include application-specific or device-specific feature boosted decision stumps.

In addition, the device processor may also dynamically generate application-specific classifier models that identify correlations or features that are relevant to a specific software application (Google & wallet) and/or to a specific type of software application (e.g., games, navigation, financial, news, productivity, etc.). In an aspect, these classifier models may be generated to include a reduced and more focused subset of the decision nodes that are included in the full classifier model or those included in less classifier model generated from the received full classifier model.

In various aspects, the device processor may be configured to generate application-based classifier models for each software application in the system and/or for each type of software application in the system. The device processor may also be configured to dynamically identify the software applications and/or application types that are a high risk or susceptible to abuse (e.g., financial applications, point-of-sale applications, biometric sensor applications, etc.), and generate application-based classifier models for only the software applications and/or application types that are identified as being high risk or susceptible to abuse. In various aspects, device processor may be configured to generate the application-based classifier models dynamically, reactively, proactively, and/or every time a new application is installed or updated.

Each software application generally performs a number of tasks or activities on the computing device. The specific execution state in which certain tasks or activities are performed in the computing device may be a strong indicator of whether a behavior or activity merits additional or easier scrutiny, monitoring and/or analysis. As such, in the various aspects, the device processor may be configured to use information identifying the actual execution states in which certain tasks or activities are performed to focus its behavioral monitoring and analysis operations, and better determine whether an activity is a critical activity and/or whether the activity is not benign.

In various aspects, the device processor may be configured to associate the activities tasks performed by a software application with the execution states in which those activities tasks were performed. For example, the device processor may be configured to generate a behavior vector that includes the behavior information collected from monitoring the implemented components in a sub-vector or data-structure that lists the features, activities, or operations of the software for which the execution state is relevant (e.g., location access, SMS read operations, sensor access, etc.). In an aspect, this sub-vector data-structure may be stored in association with a shadow feature value sub-vector data-structure that identifies the execution state in which each feature activity operation was observed. As an example, the device processor may generate a behavior vector that includes a “location_background” data field whose value identifies the number or rate that the software application accessed location information when it was operating in a background state. This allows the device processor to analyze this execution state information independent of and/or in parallel with the other observed monitored activities of the computing device. Generating the behavior vector in this manner also allows the system to aggregate information (e.g., frequency or rate) over time.

In various aspects, the device processor may be configured to generate the behavior vectors to include information that may be input to a decision node in the machine learning classifier to generate an answer to a query regarding the monitored activity.

In various aspects, the device processor may be configured to generate the behavior vectors to include a concise definition of the observed monitored behaviors. The behavior vector may succinctly describe the observed behavior of the computing device, software application, or process in a value or vector data-structure (e.g., in the form of a string of numbers, etc.). The behavior vector may also function as an identifier that enables the computing device system to quickly recognize, identify, and/or analyze computing device behaviors.

In various aspects, the device processor may be configured to generate the behavior vectors to include a plurality of series of numbers, each of which signifies or characterizes a feature, activity, or a behavior of the mobile computing device 102. For example, numbers included in the behavior vector may signify whether a certain of the computing device is in use (e.g., as zero in one), how much network traffic has been transmitted from or generated by the computing device (e.g., 20 KB sec, etc.), how many internet messages have been communicated (e.g., number of SMS messages, etc.), etc. In an aspect, the behavior vector may encapsulate one or more “behavior features.” Each behavior feature may be an abstract number that represents all or a portion of an observed behavior or action. The behavior features may be agnostic to the hardware or software configuration of the computing device.

In various aspects, the device processor may be configured to generate the behavior vectors to include execution information. The execution information may be included in the behavior vector as part of a behavior (e.g., camera used 5 times in 3 second by a background process, camera used 3 times in 3 second by a foreground process, etc.) or as part of an independent feature. In an aspect, the execution state information may be included in the behavior vector as a shadow feature value sub-vector or data structure. In an aspect, the behavior vector may store the shadow feature value sub-vector data structure in association with the feature, activities, tasks for which the execution state is relevant.

FIG. 3 illustrates method 300 of analyzing a behavior-based security system of a computing device in accordance with an aspect. In an aspect, the behavior-based security system may be the comprehensive behavioral monitoring and analysis system discussed above (e.g., with reference to FIG. 2). In various aspects, method 300 may be performed by a processor or processing core of a computing device (e.g., network server, mobile device, etc.).

In block 302, a processing core of a computing device may generate artificial attack software. In block 304, the processing core may send the generated artificial attack software to a second computing device equipped with a behavior-based security system so as to simulate a cyberattack. In block 306, the processing core may set a timer. In block 308, the processing core may listen for a response from the second computing device. In determinations blocks 310 and 312, the processing core may determine whether a response has been received from the second computing device and whether the timer has expired. In the example illustrated in FIG. 3, the processing core determines whether the timer has expired in determination block 312 in response to determining that a response has not yet been received from the second computing device (i.e., determination block 310 “No”). In another aspect, the processing core may first determine that the timer has expired before determining whether a response has been received.

In response to determining that a response has not yet been received from the second computing device (i.e., determination block 310 “No”) and the timer has not yet expired (i.e., determination block 312 “No”), the processing core may continue listening for a response in block 308. In response to

determining that the timer has expired (i.e., determination block 312 “No”), the processing core may send a dead-man signal to the second computing device in block 314.

In response to determining that a response has been received from the second computing device (i.e., determination block 310 “Yes”), in determination block 316, the processing core may determine whether the response is adequate by determining whether the second computing device correctly or adequately identified and responded to the simulated cyberattack or non-benign behavior. In response to determining that the second computing device did not correctly adequately identify or respond to the simulated cyber attack (i.e., determination block 316 “No”), the processing core may send a dead-man signal to the second computing device in block 314. In response to determining that the second computing device correctly adequately identified and responded to the simulated cyber attack or non-benign behavior (i.e., determination block 316 “Yes”), in option block 318, the processing core may send a “system-ok” signal to the second computing device.

FIG. 4 illustrates a method 400 of evaluating a behavior-based security system of a computing device in accordance with an aspect. The behavior-based security system may be a comprehensive behavioral monitoring and analysis system of a mobile or resource constrained computing device. In an aspect, method 400 may be performed by a processing core of a mobile computing device.

In block 402, the processing core may receive artificial attack software from another computing device (e.g., network server, another mobile device, etc.). In block 404, the processing core may use the received artificial attack software to simulate a cyber attack or non-benign behavior in the mobile device. In block 406, the processing core may monitor the operations of a behavior-based security system of the mobile device to collect behavior information. In block 408, the processing core may send the collected behavior information to the computing device that sent the artificial attack software. In block 410, the processing core may set a timer and listen for a response from the computing device that sent the artificial attack software. In block 412, the processing core may perform dead-man operations in response to determining that the timer has expired and the response has not yet been received from the computing device that sent the artificial attack software.

FIG. 5 illustrates an aspect method 500 of using a lean classifier model to classify a behavior of the mobile device. Method 500 may be performed by a processing core in a mobile device.

In block 502, the processing core may perform observations to collect behavior information from various components that are instrumented at various levels of the mobile device system. In an aspect, this may be accomplished via the behavior observer module 202 discussed above with reference to FIG. 2. In block 504, the processing core may generate a behavior vector characterizing the observations, the collected behavior information, and/or a mobile device behavior. Also in block 504, the processing core may use a full classifier model received from a network server to generate a lean classifier model or a family of lean classifier models of varying levels of complexity (or “leaness”). To accomplish this, the processing core may end a family of boosted decision stumps included in the full classifier model to generate lean classifier models that include a reduced number of boosted decision stumps and/or evaluate a limited number of test conditions.

In block 506, the processing core may select the leanest classifier in the family of lean classifier models (i.e., the model based on the fewest number of different mobile device

states, features, behaviors, or conditions) that has not yet been evaluated or applied by the mobile device. In an aspect, this may be accomplished by the processing core selecting the first classifier model in an ordered list of classifier models.

In block 508, the processing core may apply collected behavior information to behavior vectors to each boosted decision stump in the selected lean classifier model. Because boosted decision stumps are binary decisions, and the lean classifier model is generated by selecting many binary decisions that are based on the same test condition, the process of applying a behavior vector to the boosted decision stumps in the lean classifier model may be performed in a parallel operation. Alternatively, the behavior vector applied in block 508 may be iterated or filtered to just include the limited number of test condition parameters included in the lean classifier model, thereby further reducing the computational effort in applying the model.

In block 510, the processing core may compute or determine a weighted average of the results of applying the collected behavior information to each boosted decision stump in the lean classifier model. In block 512, the processing core may compare the computed weighted average to a threshold value. In determination block 514, the processing core may determine whether the results of this comparison and/or the results generated by applying the selected lean classifier model are suspicious. For example, the processing core may determine whether these results may be used to classify a behavior as either malicious or benign with a high degree of confidence, and if not treat the behavior as suspicious.

If the processing core determines that the results are suspicious (e.g., determination block 514 “Yes”), the processing core may repeat the operations in blocks 506-512 to select and apply a stronger (i.e., less lean) classifier model that evaluates more device states, features, behaviors, or conditions until the behavior is classified as malicious or benign with a high degree of confidence. If the processing core determines that the results are not suspicious (e.g., determination block 514 “No”), such as by determining that the behavior can be classified as either benign or non-benign with a high degree of confidence, in block 516, the processing core may use the result of the comparison generated in block 512 to classify a behavior of the mobile device as benign, non-benign or potentially malicious.

In an alternative aspect method, the operations described above may be accomplished by sequentially selecting a boosted decision stump that is not already in the lean classifier model; identifying all other boosted decision stumps that depend upon the same mobile device state, feature, behavior, or condition as the selected decision stump (and thus can be applied based upon one determination result); including in the lean classifier model the selected and all identified other boosted decision stumps that depend upon the same mobile device state, feature, behavior, or condition; and repeating the process for a number of times equal to the determined number of test conditions. Because all boosted decision stumps that depend on the same test condition as the selected boosted decision stump are added to the lean classifier model each time, limiting the number of times this process is performed will limit the number of test conditions included in the lean classifier model.

FIG. 6 illustrates an example method 600 for performing dynamic and adaptive observations in accordance with an aspect. In block 602, the device processor may perform coarse observations by monitoring/observing a subset of a large number factors/behaviors that could contribute to the mobile device’s degradation. In block 603, the device processor may generate a behavior vector characterizing the coarse

observations and/or the mobile device behavior based on the coarse observations. In block 604, the device processor may identify subsystems, processes, and/or applications associated with the coarse observations that may potentially contribute to the mobile device's degradation. This may be achieved, for example, by comparing information received from multiple sources with contextual information received from sensors of the mobile device. In block 606, the device processor may perform behavioral analysis operations based on the coarse observations. In an aspect, as part of blocks 603 and 604, the device processor may perform one or more of the operations discussed above with reference to FIGS. 2-10.

In determination block 608, the device processor may determine whether suspicious behaviors or potential problems can be identified and corrected based on the results of the behavioral analysis. When the device processor determines that the suspicious behaviors or potential problems can be identified and corrected based on the results of the behavioral analysis (i.e., determination block 608 “Yes”), in block 618, the processor may initiate a process to correct the behavior and return to block 602 to perform additional coarse observations.

When the device processor determines that the suspicious behaviors or potential problems cannot be identified and/or corrected based on the results of the behavioral analysis (i.e., determination block 608 “No”), in determination block 609, the device processor may determine whether there is a likelihood of a problem. In an aspect, the device processor may determine that there is a likelihood of a problem by computing a probability of the mobile device encountering potential problems and/or engaging in suspicious behaviors, and determining whether the computed probability is greater than a predetermined threshold. When the device processor determines that the computed probability is not greater than the predetermined threshold and/or there is not a likelihood that suspicious behaviors or potential problems exist and/or are detectable (i.e., determination block 609 “No”), the processor may return to block 602 to perform additional coarse observations.

When the device processor determines that there is a likelihood that suspicious behaviors or potential problems exist and/or are detectable (i.e., determination block 609 “Yes”), the device processor may perform deeper logging observations or final logging on the identified subsystems, processes, or applications in block 610. In block 612, the device processor may perform deeper and more detailed observations on the identified subsystems, processes, or applications. In block 614, the device processor may perform further and/or deeper behavioral analysis based on the deeper and more detailed observations.

In determination block 608, the device processor may again determine whether the suspicious behaviors or potential problems can be identified and corrected based on the results of the deeper behavioral analysis. When the device processor determines that the suspicious behaviors or potential problems cannot be identified and corrected based on the results of the deeper behavioral analysis (i.e., determination block 608 “No”), the processor may repeat the operations in blocks 610-614 until the level of detail is fine enough to identify the problem or until it is determined that the problem cannot be identified with additional detail or that no problem exists.

When the device processor determines that the suspicious behaviors or potential problems can be identified and corrected based on the results of the deeper behavioral analysis (i.e., determination block 608 “Yes”), in block 618, the device processor may perform operations to correct the problem

behavior, and the processor may return to block 602 to perform additional operations.

In an aspect, as part of blocks 602-618 of method 600, the device processor may perform real-time behavior analysis of the system's behavior to identify suspicious behaviors from limited and coarse observations, to dynamically determine the behavior to observe in greater detail, and to dynamically determine the precise level of detail required for the observations. This enables the device processor to efficiently identify and prevent problems from occurring without requiring the use of a large amount of processor, memory, or battery resources on the device.

In various aspects, a processor or processing core (collectively “processor”) may be configured to perform operations for analyzing a behavior-based security system of a target computing device. The behavior-based security system may be a comprehensive behavioral monitoring and analysis system such as the system 200 discussed above with reference to FIG. 2. In various aspects, the processor may be included in the target computing device or in a different computing device as the target computing device.

In an aspect, the operations may be performed by an attack simulation module operating on the processor. The attack simulation module may be configured to run on the same operating system as the behavior-based security system or on a hypervisor of the target computing device. In an aspect, the target computing device may include or configured in software instructions to execute a hypervisor that is configured to monitor, control, or evaluate the operations of the attack simulation module. This allows the attack simulation module to operate on the same operating system as the behavior-based security system because the hypervisor helps ensure that the attack simulation module does not become vulnerable to the same types of attacks or problems as the behavior-based security system.

As mentioned above, a processor may be configured to perform operations for analyzing a behavior-based security system of a target computing device. As part of these operations, the processor may generate artificial attack software. The artificial attack software may include executable code, scripts, or other information suitable for simulating a non-benign behavior in the target computing device (i.e., the same or different computing device that includes the behavior-based security system being evaluated). That is, the processor may be configured to generate the artificial attack software to include information that is suitable for simulating a non-benign behavior in the target computing device.

For example, in an aspect, the processor may be configured to receive a corpus of behavior information from many computing devices (e.g., mobile devices, etc.) and/or the corpus of behavior information to identify a non-benign behavior, and generating the artificial attack software to include information suitable for causing a processor to perform operations associated with the identified non-benign behavior. As another example, the processor may generate the artificial attack software to include information suitable for causing a processor to perform operations that are associated with a known cyber attack.

The processor may be configured to send the generated artificial attack software to a target computing device via communication messages, function calls, memory read/write operations, etc. to cause the target computing device to perform operations associated with a known identified non-benign behavior. In response, the processor may receive behavior information from the target computing device, generate a behavior vector based on the received behavior information, apply the generated behavior vector to a classifier

model to generate a result, and use the generated result to determine whether the behavior-based security system responded adequately to the simulated non-benign behavior. In addition, the processor may be configured to determine whether the behavior-based security system responded adequately to the simulated non-benign behavior by determining whether a response was received within a defined time period, such as before expiration of a timer.

The processor may be configured to send a dead-man signal to the target computing device in response to determining that the behavior-based security system did not respond adequately to the simulated non-benign behavior. Sending a dead-man signal may include sending a communication message that includes information suitable for causing the mobile device to perform one or more dead-man operations, such as reboots, for preventing the completion of an NFC-based transaction by the target computing device.

In an aspect, the target computing device may be configured to receive artificial attack software, install the received artificial attack software, monitor operations of the behavior-based security system to collect behavior information, and send the collected behavior information to the processor. The target computing device may set a timer in response to sending the collected behavior information, and perform dead-man operations in response to determining that a system-on-chip was not received by the mobile device before an expiration of a timer.

The various aspects may be implemented on a variety of computing devices, an example of which is illustrated in FIG. 7. Specifically, FIG. 7 is a system block diagram of a mobile computing device in the form of a smartphone cell phone 700 suitable for use with any of the aspects. The cell phone 700 may include a processor 702 coupled to internal memory 704, a display 706, and to a speaker 708. Additionally, the cell phone 700 may include an antenna 710 for sending and receiving electromagnetic radiation that may be connected to a wireless data link and/or cellular telephone transceiver 712 coupled to the processor 702. Cell phones 700 typically also include menu selection buttons or rocker switches 714 for receiving user inputs.

A typical cell phone 700 also includes a sound encoding/decoding circuit 716 that digitizes sound received from a microphone into data packets suitable for wireless transmission, and decodes received sound data packets to generate analog signals that are provided to the speaker 708 to generate sound. Also, one or more of the processor 702, wireless transceiver 712, and CODEC 716 may include a digital signal processor (DSP) circuit (not shown separately). The cell phone 700 may further include a ZigBee transceiver (i.e., an IEEE 802.15.4 transceiver) for low power short-range communications between wireless devices, or other similar communication circuitry (e.g., circuitry implementing the Bluetooth® or Wi-Fi protocols, etc.).

The aspects and network servers described above may be implemented in variety of commercially available server devices, such as the server 800 illustrated in FIG. 8. Such a server 800 typically includes a processor 801 coupled to volatile memory 802 and a large capacity non-volatile memory, such as a disk drive 803. The server 800 may also include a floppy disc drive, compact disc (CD) or digital disc drive 804 coupled to the processor 801. The server 800 may also include network access ports 806 coupled to the processor 801 for establishing data connections with a network 805, such as a local area network coupled to other communication system computers and servers.

The processors 702, 801, may be any programmable microprocessor, microcomputer or multiple processor chip or

chips that can be configured by software instructions (applications) to perform a variety of functions, including the functions of the various aspects described below. In some mobile devices, multiple processors 702 may be provided, such as one processor dedicated to wireless-communication functions and one processor dedicated to running other applications. Typically, software applications may be stored in the internal memory 704, 802, before they are accessed and loaded into the processor 902, 801. The processor 702, 801 may include internal memory sufficient to store the application software instructions. In some servers, the processor 801 may include internal memory sufficient to store the application software instructions. In some receiver devices, the secure memory may be in a separate memory chip coupled to the processor 801. The internal memory 704, 802 may be a volatile or nonvolatile memory, such as flash memory, or a mixture of both. For the purposes of this description, a general reference to memory refers to all memory accessible by the processor 702, 801, including internal memory 704, 802, removable memory plugged into the device, and memory within the processor 702, 801 itself.

Many modern computing are resource constrained systems that have relatively limited processing, memory, and energy resources. For example, a mobile device is a complex and resource constrained computing device that includes many features or factors that could contribute to its degradation in performance and power utilization levels over time. Examples of factors that may contribute to performance degradation include poorly designed software applications, malware, viruses, fragmented memory, and background processes. Due to the number, variety, and complexity of these factors, it is often not feasible to evaluate all of the various competitive behaviors, processes, operations, conditions, states, or feature combinations (hereof) that may degrade performance and/or power utilization levels of these complex yet resource-constrained systems. As such, it is difficult for users, operating systems, or application programs (e.g., anti-virus software, etc.) to accurately and efficiently identify the sources of such problems. As a result, mobile device users currently have few remedies for preventing the degradation in performance and power utilization levels of a mobile device over time, or for restoring an aging mobile device to its original performance and power utilization levels.

The various aspects discussed in this application are especially well suited for use in resource constrained computing devices, such as mobile devices, because they do not require evaluating a very large corpus of behavior information, generate classifier behavior models dynamically to account for device-specific or application-specific features of the computing device, intelligently prioritize the features that are tested evaluated by the classifier behavior models, are not limited to evaluating an individual application program or process, intelligently identify the factors or behaviors that are to be monitored by the computing device, accurately and efficiently classify the monitored behaviors, and/or do not require the execution of computationally-intensive processes. For all these reasons, the various aspects may be implemented or performed in a resource-constrained computing device without having a significant negative and/or user-perceivable impact on the responsiveness, performance, or power consumption characteristics of the device.

For example, modern mobile devices are highly configurable and complex systems. As such, the factors or features that are most important for determining whether a particular device behavior is benign or not benign (e.g., malicious or performance-degrading) may be different in each mobile device. Further, a different combination of factors/features

may require monitoring and/or analysis in each mobile device in order for that device to quickly and efficiently determine whether a particular behavior is benign or not benign. Yet, the precise combination of factor/features that require monitoring and analysis, and the relative priority or importance of each feature or feature evaluation can often only be determined using device-specific information obtained from the specific computing device in which the behavior is to be monitored or analyzed. In these and other scenarios, classifier models generated in any computing device other than the specific device in which they are used cannot include information that identifies the precise combination of factor/features that are most important to classifying a software application or device behavior in that specific device. That is, by generating classifier models in the specific computing device in which the models are used, the various aspects generate improved models that better identify and prioritize the factors/features that are most important for determining whether a software application, process, activity or device behavior is benign or non-benign.

As used in this application, the terms "component," "module," "intended," "system," "engine," "generator," "manager," and the like are intended to include a computer-related entity, such as, but not limited to, hardware, firmware, a combination of hardware and software, software, or software in execution, which are configured to perform particular operations or functions. For example, a component may be, but is not limited to, a process running on a processor, a processor, an object, an executable, a thread of execution, a program, and/or a computer. By way of illustration, both an application running on a computing device and the computing device may be referred to as a component. One or more components may reside within a process and/or thread of execution and a component may be located on one processor or core and/or distributed between two or more processors or cores. In addition, these components may execute from various non-transitory computer readable media having various instructions and/or data structures stored thereon. Components that communicate by way of local and/or remote processes, function as preexisting calls, electronic signals, data packets, memory read/writes, and other known network, computer, processor, and/or process related communication methodologies.

The foregoing method descriptions and the process flow diagrams are provided merely as illustrative examples and are not intended to require or imply that the steps of the various aspects must be performed in the order presented. As will be appreciated by one of skill in the art the order of steps in the foregoing aspects may be performed in any order. Words such as "herein," "then," "next," etc. are not intended to limit the order of the steps; these words are simply used to guide the reader through the description of the methods. Further, any reference to claim elements in the singular, for example, using the articles "a," "an" or "the" is not to be construed as limiting the element to the singular.

The various illustrative logical blocks, modules, circuits, and algorithm steps described in connection with the aspects disclosed herein may be implemented as electronic hardware, computer software, or combinations of both. To clearly illustrate this interchangeability of hardware and software, various illustrative components, blocks, modules, circuits, and steps have been described above generally in terms of their functionality. Whether such functionality is implemented as hardware or software depends upon the particular application and design constraints imposed on the overall system. Skilled artisans may implement the described functionality in varying ways for each particular application, but such implemen-

tation decisions should not be interpreted as causing a departure from the scope of the present invention.

The hardware used to implement the various illustrative logics, logical blocks, modules, and circuits described in connection with the aspects disclosed herein may be implemented or performed with a general purpose processor, a digital signal processor (DSP), an application specific integrated circuit (ASIC), a field programmable gate array (FPGA) or other programmable logic device, discrete gate or transistor logic, discrete hardware components, or any combination thereof designed to perform the functions described herein. A general-purpose processor may be a microprocessor, but in the alternative, the processor may be any conventional processor, controller, microcontroller, or state machine. A processor may also be implemented as a combination of computing devices, e.g., a combination of a DSP and a microprocessor, a plurality of microprocessors, one or more microprocessors in conjunction with a DSP core, or any other such configuration. Alternatively, some steps or methods may be performed by circuitry that is specific to a given function.

In one or more exemplary aspects, the functions described may be implemented in hardware, software, firmware, or any combination thereof. If implemented in software, the functions may be stored as one or more instructions or code on a non-transitory computer-readable medium or non-transitory processor-readable medium. The steps of a method or algorithm disclosed herein may be embodied in a processor-executable software module which may reside on a non-transitory computer-readable or processor-readable storage medium. Non-transitory computer-readable or processor-readable storage media may be any storage media that may be accessed by a computer or a processor. By way of example but not limitation, such non-transitory computer-readable or processor-readable media may include: XML, ROM, EPROM, EEPROM, ASII memory, CD-ROM or other optical disk storage, magnetic disk storage or other magnetic storage devices, or any other medium that may be used to store desired program code in the form of instructions or data structures and that may be accessed by a computer. Disk and disc, as used herein, includes compact disc (CD), laser disc, optical disc, digital versatile disc (DVD), floppy disk, and blue-ray disc where disks generally reproduce data magnetically, while discs reproduce data optically with lasers. Combinations of the above are also included within the scope of non-transitory computer-readable and processor-readable media. Additionally, the operations of a method or algorithm may reside as one or any combination or set of codes and/or instructions on a non-transitory processor-readable medium and/or computer-readable medium, which may be incorporated into a computer program product.

The preceding description of the disclosed aspects is provided to enable any person skilled in the art to make or use the present invention. Various modifications to these aspects will be readily apparent to those skilled in the art, and the generic principles defined herein may be applied to other aspects without departing from the spirit or scope of the invention. Thus, the present invention is not intended to be limited to the aspects shown herein but is to be accorded the widest scope consistent with the following claims and the principles and novel features disclosed herein.

What is claimed is:

1. A method of analyzing a behavior-based security system of a mobile device, comprising:
generating by a processor artificial attack software configured to simulate a non-benign behavior in the mobile device;

sending the generated artificial attack software to the mobile device so as to simulate the non-benign behavior in the mobile device;

receiving behavior information from the mobile device in response to sending the generated artificial attack software to the mobile device;

generating a behavior vector based on the received behavior information;

applying the generated behavior vector to a classifier model to generate a result;

using the generated result to determine whether the behavior-based security system of the mobile device responded adequately to the simulated non-benign behavior; and

sending a dead-man signal to the mobile device in response to determining that the behavior-based security system of the mobile device did not respond adequately to the simulated non-benign behavior.

2. The method of claim 1, further comprising setting a timer by the processor, wherein determining whether the behavior-based security system of the mobile device responded adequately to the simulated non-benign behavior comprises determining whether a response was received from the mobile device before expiration of the timer.

3. The method of claim 1, further comprising sending a system-ok signal to the mobile device in response to determining that the behavior-based security system of the mobile device responded adequately to the simulated non-benign behavior.

4. The method of claim 1, wherein sending the dead-man signal to the mobile device comprises sending a communication message that includes information suitable for causing the mobile device to perform dead-man operations.

5. The method of claim 1, wherein generating the artificial attack software comprises:

- receiving a corpus of behavior information from many mobile devices;
- analyzing the corpus of behavior information to identify the non-benign behavior; and
- generating the artificial attack software to include information suitable for causing a mobile device processor of the mobile device to perform the identified non-benign behavior.

6. The method of claim 1, wherein generating the artificial attack software comprises generating the artificial attack software to include information suitable for causing a mobile device processor of the mobile device to perform operations associated with a known cyber attack.

7. The method of claim 1, further comprising:

- receiving the artificial attack software in a mobile device processor of the mobile device;
- installing the received artificial attack software in the mobile device;
- monitoring operations of the behavior-based security system to collect behavior information; and
- sending collected behavior information to the processor.

8. The method of claim 7, further comprising:

- setting by the mobile device processor a timer in response to sending the collected behavior information; and
- performing by the mobile device processor dead-man operations in response to determining that a system-ok signal has not been received by the mobile device and that the timer has expired.

9. The method of claim 1, wherein the simulated non-benign behavior includes simulated behaviors of a near-field communication (NFC) transaction, and wherein the dead-

man signal includes information suitable for causing the mobile device to prevent completion of an near field communication (NFC) transaction.

10. A computing device, comprising:

- a processor configured with processor-executable instructions to perform operations comprising:
- generating artificial attack software configured to stimulate a non-benign behavior in a mobile device;
- sending the generated artificial attack software to the mobile device so as to simulate the non-benign behavior in the mobile device;
- receiving behavior information from the mobile device in response to sending the generated artificial attack software to the mobile device;
- generating a behavior vector based on the received behavior information;
- applying the generated behavior vector to a classifier model to generate a result;
- using the generated result to determine whether a behavior-based security system of the mobile device responded adequately to the simulated non-benign behavior; and
- sending a dead-man signal to the mobile device in response to determining that the behavior-based security system of the mobile device did not respond adequately to the simulated non-benign behavior.

11. The computing device of claim 10, wherein:

- the processor is configured with processor-executable instructions to perform operations further comprising setting a timer by the processor, and
- the processor is configured with processor-executable instructions to perform operations such that determining whether the behavior-based security system of the mobile device responded adequately to the simulated non-benign behavior comprises determining whether a response was received from the mobile device before expiration of the timer.

12. The computing device of claim 10, wherein the processor is configured with processor-executable instructions to perform operations further comprising sending a system-ok signal to the mobile device in response to determining that the behavior-based security system of the mobile device responded adequately to the simulated non-benign behavior.

13. The computing device of claim 10, wherein the processor is configured with processor-executable instructions to perform operations such that sending the dead-man signal to the mobile device comprises sending a communication message that includes information suitable for causing the mobile device to perform dead-man operations.

14. The computing device of claim 10, wherein the processor is configured with processor-executable instructions to perform operations such that generating artificial attack software comprises:

- receiving a corpus of behavior information from many mobile devices;
- analyzing the corpus of behavior information to identify the non-benign behavior; and
- generating artificial attack software to include information suitable for causing a mobile device processor of the mobile device to perform the identified non-benign behavior.

15. The computing device of claim 10, wherein the processor is configured with processor-executable instructions to perform operations such that generating artificial attack software comprises generating artificial attack software to

include information suitable for causing a mobile device processor of the mobile device to perform operations associated with a known cyber attack.

16. A non-transitory computer readable storage medium having stored thereon processor-executable software instructions configured to cause a processor to perform operations for analyzing a behavior-based security system of a mobile device, the operations comprising:

- generating artificial attack software configured to simulate a non-benign behavior in the mobile device;
- sending the generated artificial attack software to the mobile device so as to simulate the non-benign behavior in the mobile device;
- receiving behavior information from the mobile device in response to sending the generated artificial attack software to the mobile device;
- generating a behavior vector based on the received behavior information;
- applying the generated behavior vector to a classifier model to generate a result;
- using the generated result to determine whether the behavior-based security system of the mobile device responded adequately to the simulated non-benign behavior; and
- sending a dead-man signal to the mobile device in response to determining that the behavior-based security system of the mobile device did not respond adequately to the simulated non-benign behavior.

17. The non-transitory computer readable storage medium of claim 16, wherein the stored processor-executable software instructions are configured to cause a processor to perform operations further comprising setting a timer by the processor, wherein determining whether the behavior-based security system of the mobile device responded adequately to the simulated non-benign behavior comprises determining whether a response was received from the mobile device before expiration of the timer.

18. The non-transitory computer readable storage medium of claim 16, wherein the stored processor-executable software instructions are configured to cause a processor to perform operations further comprising sending a system-ok signal to the mobile device in response to determining that the behavior-based security system of the mobile device responded adequately to the simulated non-benign behavior.

19. The non-transitory computer readable storage medium of claim 16, wherein the stored processor-executable software instructions are configured to cause a processor to perform operations such that sending the dead-man signal to the mobile device comprises sending a communication message that includes information suitable for causing the mobile device to perform dead-man operations.

20. The non-transitory computer readable storage medium of claim 16, wherein the stored processor-executable software instructions are configured to cause a processor to perform operations such that generating artificial attack software comprises:

- receiving a corpus of behavior information from many mobile devices;
- analyzing the corpus of behavior information to identify the non-benign behavior; and

generating artificial attack software to include information suitable for causing a mobile device processor of the mobile device to perform the identified non-benign behavior.

21. The non-transitory computer readable storage medium of claim 16, wherein the stored processor-executable software instructions are configured to cause a processor to perform operations such that generating artificial attack software comprises generating artificial attack software to include information suitable for causing a mobile device processor of the mobile device to perform operations associated with a known cyber attack.

22. A computing device, comprising:

- means for generating artificial attack software configured to simulate a non-benign behavior in a mobile device;
- means for sending the generated artificial attack software to the mobile device so as to simulate the non-benign behavior in the mobile device;
- means for receiving behavior information from the mobile device in response to sending the generated artificial attack software to the mobile device;
- means for generating a behavior vector based on the received behavior information;
- means for applying the generated behavior vector to a classifier model to generate a result;
- means for using the generated result to determine whether a behavior-based security system of the mobile device responded adequately to the simulated non-benign behavior; and
- means for sending a dead-man signal to the mobile device in response to determining that the behavior-based security system of the mobile device did not respond adequately to the simulated non-benign behavior.

23. The computing device of claim 22, further comprising means for setting a timer, wherein means for determining whether the behavior-based security system of the mobile device responded adequately to the simulated non-benign behavior comprises means for determining whether a response was received from the mobile device before expiration of the timer.

24. The computing device of claim 22, further comprising means for sending a system-ok signal to the mobile device in response to determining that the behavior-based security system of the mobile device responded adequately to the simulated non-benign behavior.

25. The computing device of claim 22, wherein means for sending the dead-man signal to the mobile device comprises sending a communication message that includes information suitable for causing the mobile device to perform dead-man operations.

26. The computing device of claim 22, wherein means for generating artificial attack software comprises

- means for receiving a corpus of behavior information from many mobile devices;
- means for analyzing the corpus of behavior information to identify the non-benign behavior; and
- means for generating artificial attack software to include information suitable for causing a mobile device processor of the mobile device to perform the identified non-benign behavior.

* * * *



US09501328B2

(12) **United States Patent**
Robatmili et al.

(10) **Patent No.:** US 9,501,328 B2
(11) **Date of Patent:** Nov. 22, 2016

(54) **METHOD FOR EXPLOITING PARALLELISM IN TASK-BASED SYSTEMS USING AN ITERATION SPACE SPLITTER**

(71) **Applicant:** QUALCOMM Incorporated, San Diego, CA (US)

(72) **Inventors:** Behman Ruhatlilli, San Jose, CA (US); Shaizeen Dilawarhun Aga, Ann Arbor, MI (US); Dario Suarez Guefa, Santa Clara, CA (US); Arun Ramam, Santa Clara, CA (US); Aravind Narayanan, Sunnyvale, CA (US); Gheorghe Calin Cascaval, Palo Alto, CA (US); Pablo Montesinos Ortega, Fremont, CA (US); Han Zhao, Santa Clara, CA (US)

(73) **Assignee:** QUALCOMM Incorporated, San Diego, CA (US)

(*) **Notice:** Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days

(21) **Appl. No.:** 14/673,857

(22) **Filed:** Mar. 30, 2015

(65) **Prior Publication Data**

US 2016/0293012 A1 Oct. 6, 2016

(51) **Int. Cl.**

G06F 9/46 (2006.01)

G06F 9/50 (2006.01)

(52) **U.S. Cl.**

CPC G06F 9/5066 (2013.01); G06F 9/5027 (2013.01)

(58) **Field of Classification Search**

CPC G06F 9/5066; G06F 9/5027

USPC 718/104

See application file for complete search history

(56) **References Cited**

U.S. PATENT DOCUMENTS

8,178,967 B2	7,3914	Bordelon et al
8,215,291 B2	8,3914	Masson et al
2005/0254742 A1	9,3558	Vasapati et al
2009/0406333 A1	12,3529	Harper, III
2011/0319187 A1	7,3913	Vorbach
2014/0268893 A1*	9,3515	Monclinos
		Oriago
		0,063-9,3881
		"18,106

OTHER PUBLICATIONS

- Lam D., "Sharing Information in Parallel Search with Search Space Partitioning," Apr. 3, 2013, 69 Pages
Raman R., et al., "Compiler Support for Work-Stealing Parallel Runtime Systems," 2010, 20 pages

(continued)

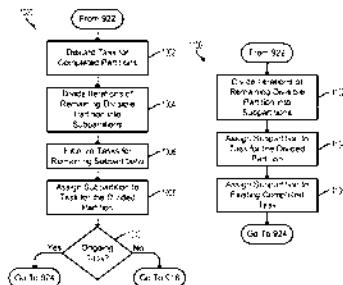
Primary Examiner: Caetano Trinca

(74) **Attorney, Agent, or Firm:** The Marbury Law Group, PLLC

(57) **ABSTRACT**

This embodiment includes computing devices, systems, and methods for task-based handling of repetitive processes in parallel. At least one processor of the computing device, or a specialized hardware controller, may be configured to partition iterations of a repetitive process and assign the partitions to initialized tasks to be executed in parallel by a plurality of processor cores. Upon completing a task, remaining divisible portions of the repetitive process of ongoing tasks may be subpartitioned and assigned to the ongoing task and the completed task or a newly initialized task. Information about the iteration space for a repetitive task may be stored in a descriptor table, and status information for all partitions of a repetitive process stored in a status table. Each processor core may have an associated local table that tracks iteration execution of each task and is synchronized with the status table.

18 Claims 12 Drawing Sheets



US 9,501,328 B2

Page 2

(56)

References Cited

OIFIR PUBLICATIONS

- International Search Report and Written Opinion - PCT/US2016/018681 - ISA (PPO) - May 24, 2016
Kimura, S. et al., "Calculus of CM", Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA '97), Apr. 26, 2007 (Apr. 26, 2007 - Apr. 29, 2007) (Apr. 26, 2007), p. 102, XP055269691, San Diego, California DOI: 10.1145/1259862.1259883 [REPA, 97-X-1-99981-706-3]
Sanchez, D., et al., "Flexible Architectural Support for Fine-Grain Scheduling", Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating

- Systems, ASPLOS '19, Mar. 13, 2019 (Mar. 13, 2019-Mar. 17, 2019) (Mar. 17, 2019), p. 311, XP055366013, New York, New York, USA DOI: 10.1145/3291202.3291345 ISBN: 978-1450358339-1
Lamport, L. et al., "Lazy Bus Splitting", ACM SIGPLAN Notices, vol. 35, No. 5, May 1, 2010 (May 1, 2010), p. 179, XP055205327, 2 Paper Place, Suite 701 New York, NY 10121-4701 USA ISSN: 0362-1318, DOI: 10.1145/1857853.1853270
Yigitcan, W., et al., "A Hierarchical WorkStealing Framework for Multi-core Clusters", 2012 15th International Conference on Parallel and Distributed Computing, Applications and Technologies, II/1 - Dec. 13, 2012 (Dec. 14, 2012), pp. 356-358, XP032475790 DOI: 10.1109/PDCAT.2012.627 ISBN: 978-0-7695-4879-1 [retrieved on Sep. 5, 2017]

* cited by examiner

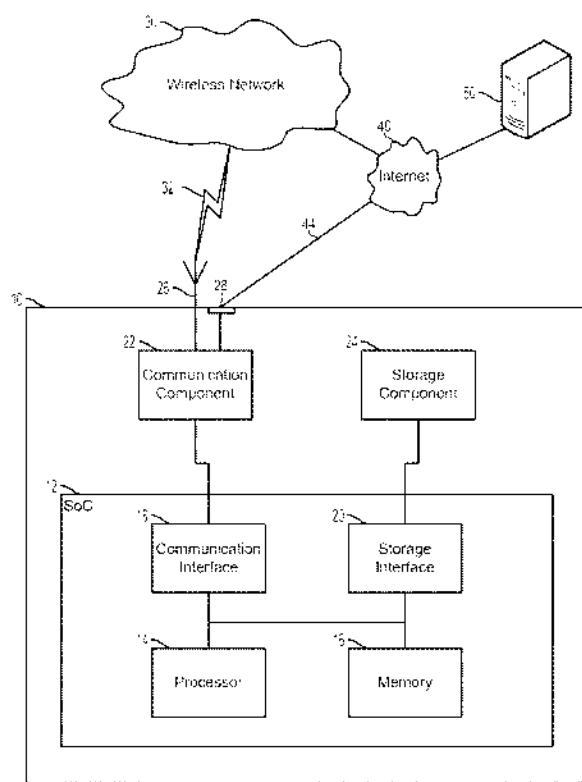


FIG. 1

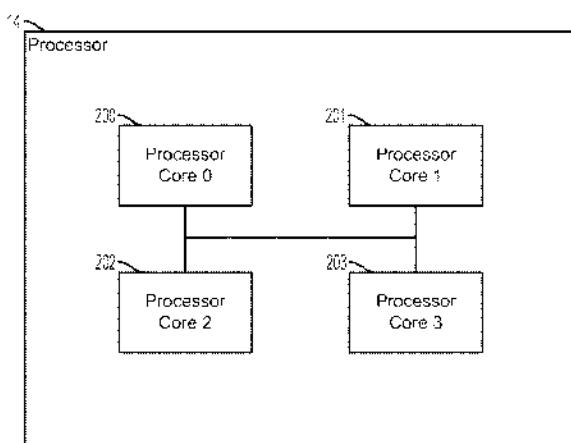


FIG. 2

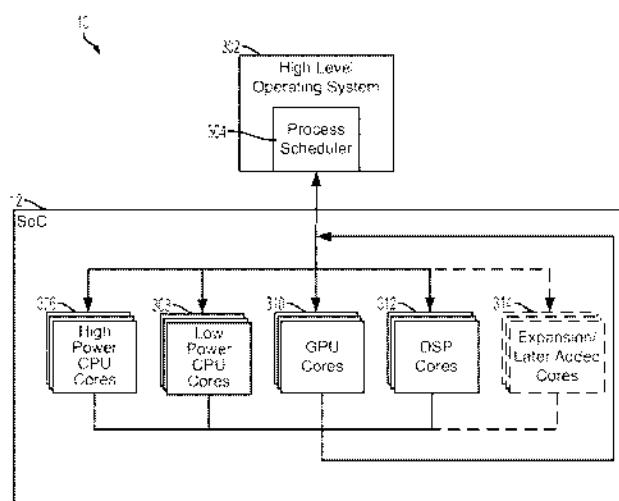


FIG. 3

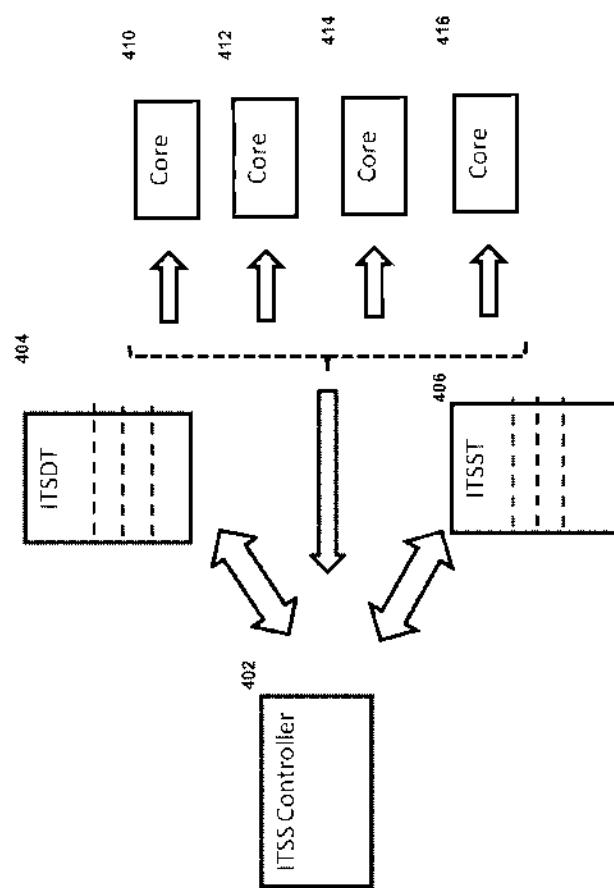


FIG. 4

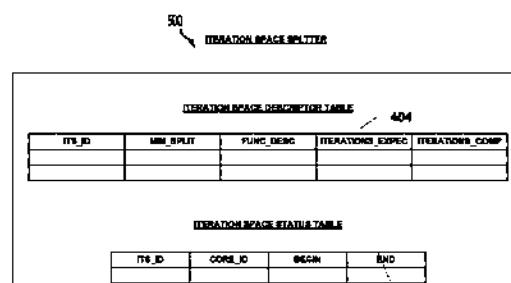


FIG. 5A



FIG. 5B

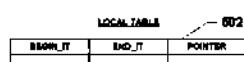


FIG. 5C

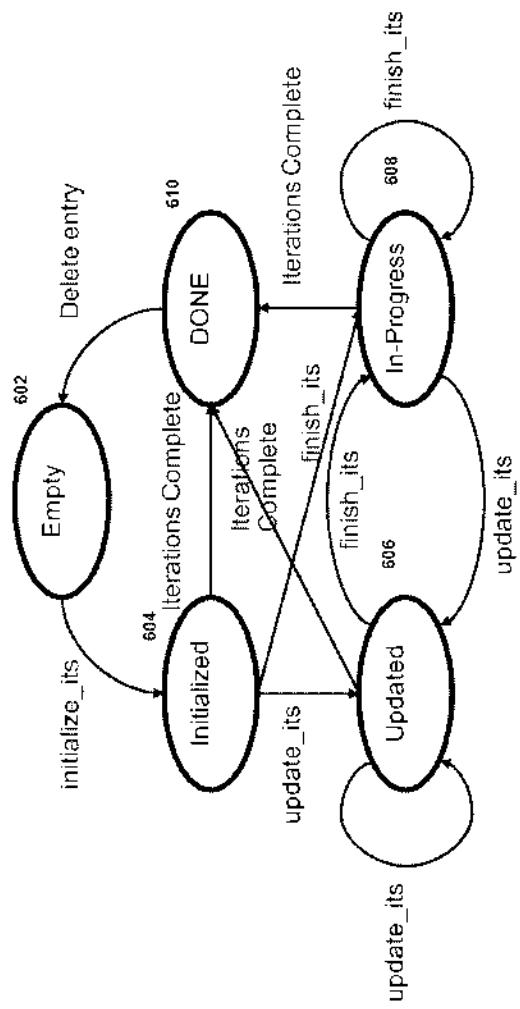


FIG. 6

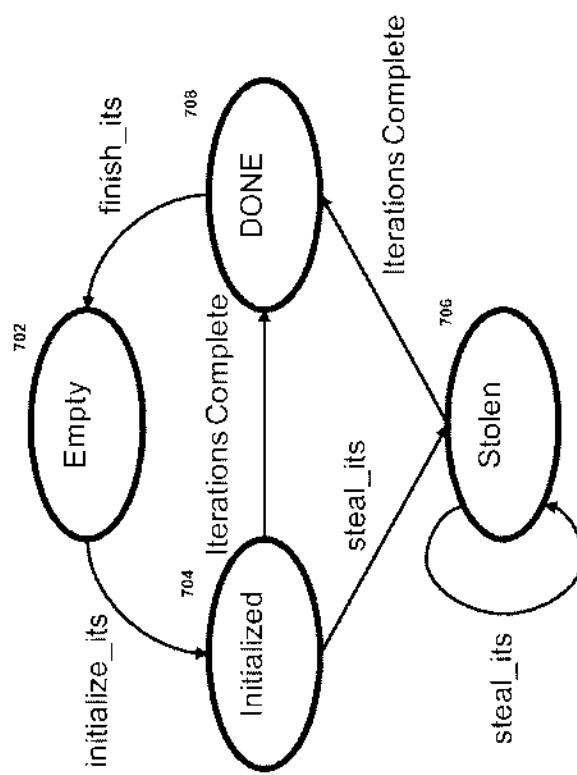


FIG. 7

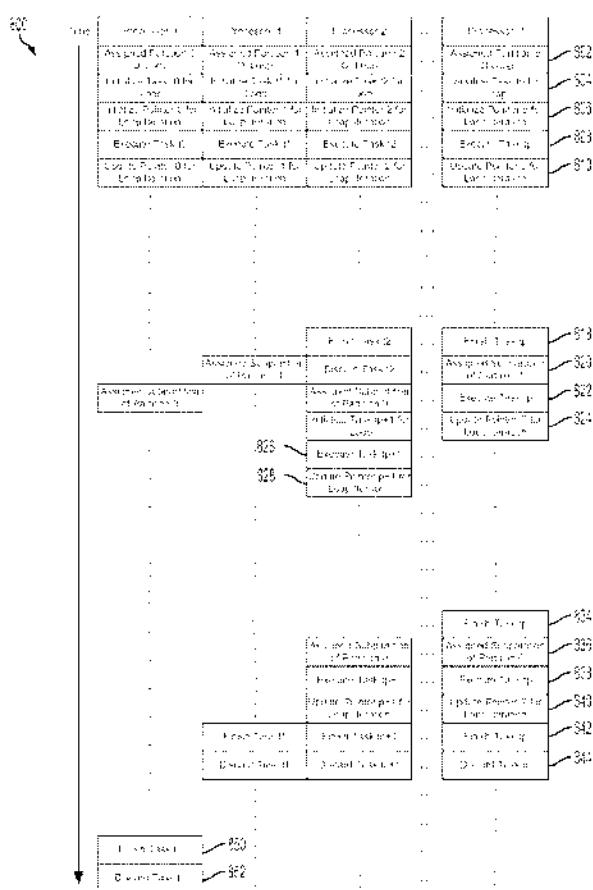


FIG. 8

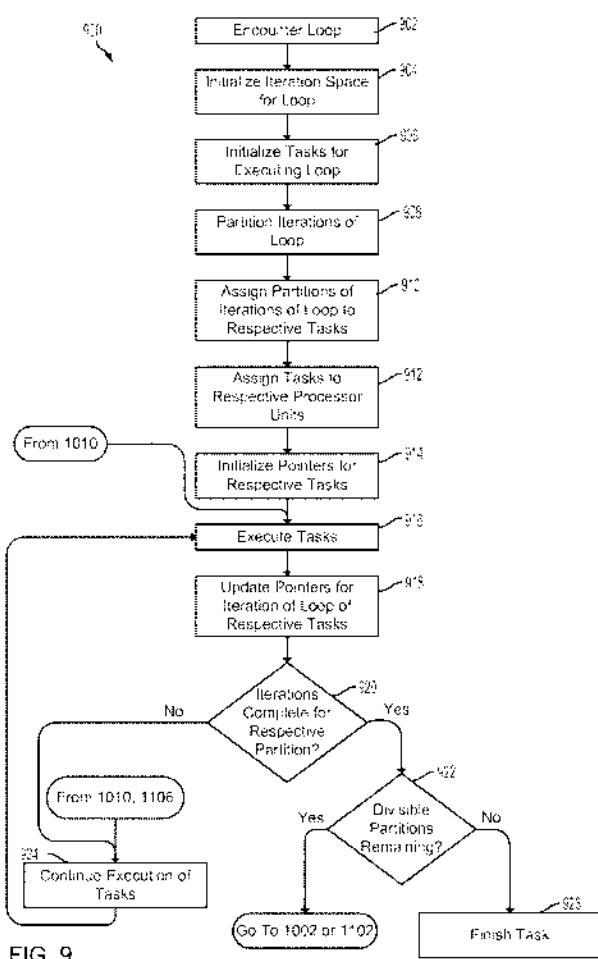


FIG. 9

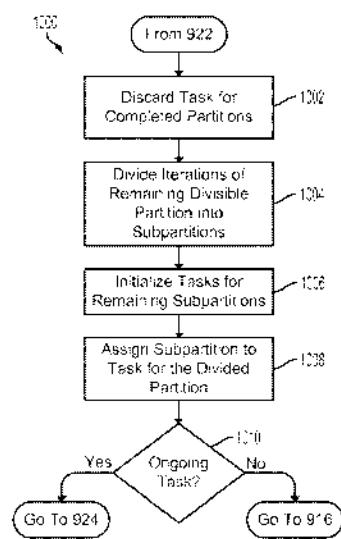


FIG. 10

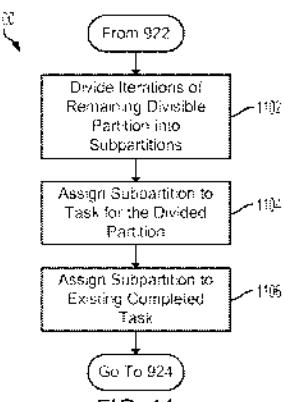


FIG. 11

U.S. Patent

Nov. 22, 2016

Sheet 11 of 12

US 9,501,328 B2

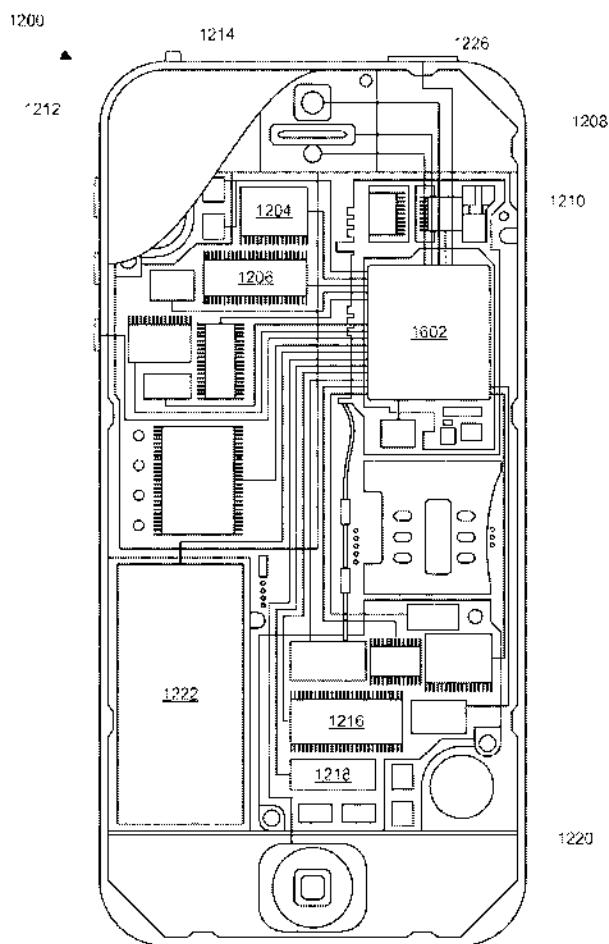


FIG. 12

U.S. Patent

Nov. 22, 2016

Sheet 12 of 12

US 9,501,328 B2

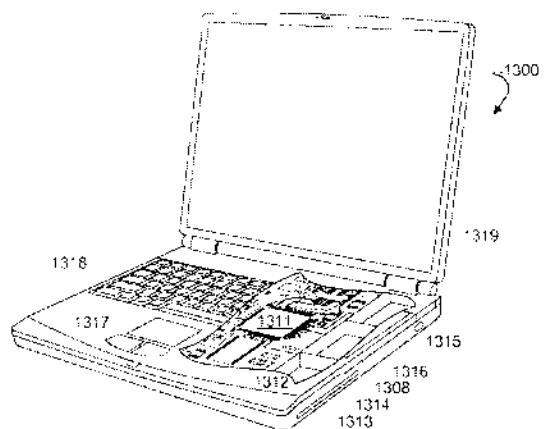


FIG. 13

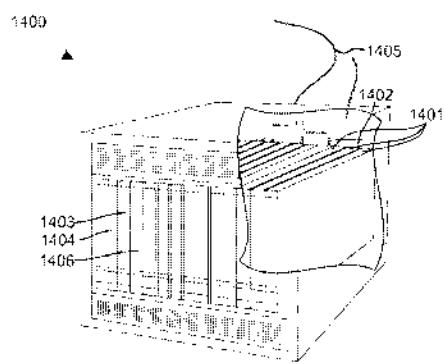


FIG. 14

1

**METHOD FOR EXPLOITING PARALLELISM
IN TASK-BASED SYSTEMS USING AN
ITERATION SPACE SPLITTER**

BACKGROUND

A common concept in computer programming is the execution of one or more instructions repetitively according to a given criterion. This repetitive execution can be accomplished by programming using recursion, fixed point iteration, or looping constructs. In various instances computer programs can include repetitions of processes in which a repetitive process may execute a certain number of times according to a criterion. For instance, if the repetitive process criterion directs the repetitive process to execute "n" number of times, the total number of executions of the repetitive process can be as great as " n^c " executions.

In some computer systems with multiple processors or multi-core processors, execution of processes can be run in parallel with each other on the multiple processors or cores. Such parallel execution of repetitive processes can improve the performance of the computer system. For example, in a computer system with four or more processors or processor cores, if the repetitive process criterion directs the repetitive process to execute n number of times, it can be split into p divisions, for example $n_0, n_1, n_2, \dots, n_p$. The p divisions of n can each represent a subset of the number of times to execute the repetitive process. The repetitive process can be assigned to execute on respective processors or processor cores for one of the subsets $n_0, n_1, n_2, \dots, n_p$.

However, in many computer systems, this does not alleviate an issue with the overall overhead involved in executing repetitive processes. In a task-based multi-time system, a separate task can be created for each execution of the p divisions of each repetitive process. The greater the number of tasks, the greater an amount of overhead is created for managing all of the tasks.

SUMMARY

The methods and apparatuses of various embodiments provide circuits and methods for task-based handling of repetitive processes in multi-technology communication devices. Embodiment methods may include detecting, by a processor or processor core of a multi-technology communications device, a repetitive process ready for execution; initializing, by a central controller of the processor or processor core, a set of descriptor table entries for iteration space information of the repetitive process; initializing, by the central controller of the processor or processor core, a set of status table entries for iteration space information of the repetitive process; partitioning, by the central controller of the processor or processor core, iterations of the repetitive process into a first plurality of partitions; initializing, by the central controller of the processor or processor core, a first task for executing iterations of a first partition; initializing, by the central controller of the processor or processor core, a second task for executing iterations of a second partition; assigning, by the central controller of the processor or processor core, the first task to execute by a first processor or processor core and the second task to execute by a second processor or processor core in parallel; updating, by multiple local controllers each associated with one of the processors or processor cores, multiple local tables to include partition execution status information for a task assigned to execute on the respective processor or processor core, wherein each table is associated with one of the processors or processor

2

cores; and executing the first task, by the first processor or processor core, and executing the second task by the second processor or processor core in parallel.

Some embodiments may include completing, by the second processor or processor core, execution of the second task; updating, by the local controller associated with the second processor or processor core, the local table associated with the second processor or processor core to indicate that the second task has finished execution; updating, by the central controller of the processor, the status table to indicate that the second task has finished completion and the second processor or processor core is available; determining, by the central controller of the processor, whether the first partition is divisible; and partitioning, by the central controller of the processor, the first partition of the first task into a second plurality of partitions in response to determining that the first partition is divisible. Such embodiments may further include assigning, by the central controller of the processor, a third partition of the second plurality of partitions to the second task for execution, and a fourth partition of the second plurality of partitions to the first task for execution; updating, by the central controller of a processor, the status table with at least the additional number of partitions, task assignment information, and task iteration status; and updating, by multiple local controllers, each local controller associated with one of the processors or processor cores, multiple local tables, each table associated with one of the processors or processor cores, to include updated partition execution information for a task assigned to execute on the respective processor core. Alternatively, some of such embodiments, each of the partitions of the second plurality of partitions may be of any size. Alternatively, in some of such embodiments, determining whether the first partition is divisible further comprises determining, by the central controller of the processor, whether the remaining iterations of partitions assigned to the first task exceeds a minimum threshold.

Some embodiments may include partitioning, by the central controller of the processor, the iterations of the repetitive process by a number of partitions equivalent to a number of available processors or processor cores.

Some embodiments may include initializing, by a local controller associated with the first processor or processor core, a first pointer in the local table associated with the first processor or processor core for the first task; updating, by the local controller of the first processor or processor core, the first pointer to indicate execution of the iterations of the repetitive process of the first partition; and updating, by the central controller of the processor, an execution status of the first partition in the status table.

Some embodiments may include determining, by the local controller of the first processor or processor core, that an end iteration criterion for the first partition is invalid; updating, by the local controller of the first processor or processor core, a local table associated with the first processor or processor core to reflect the new end iteration criterion for the first partition; and updating, by the central controller, the status table with the new end iteration criterion for the first partition.

In some embodiments, updating any of the local tables further comprises requesting, by the central controller of the processor, partition information contained in a local table.

In some embodiments, updating any of the local tables further comprises passing, by the local controller of a respective processor or processor core, information updated in a local table to the status table at predetermined intervals or in response to an event.

US 9,501,328 B2

3

In some embodiments, the states table or any of the local tables may be updated without interrupting execution of the tasks by the processors or processor cores.

Some embodiments may further include completing, by the processors or processor cores, execution of all partitions of a repetitive process, deleting by the local controllers of the multiple processor cores, the partition entries associated with the repetitive process from each of the local tables, and deleting, by the central controller, memory space information from the status table and descriptor table.

Embodiments include a multi-technology communication device having multiple processors or processor cores configured with processor executable instructions to perform operations of one or more of the embodiment methods described above.

Embodiments include a multi-technology communication device having means for performing functions of one or more of the embodiment methods described above.

Embodiments include a non-transitory processor-readable medium having stored thereon processor-executable software instructions to cause a processor to perform operations of one or more of the embodiment methods described above.

BRIEF DESCRIPTION OF THE DRAWINGS

The accompanying drawings, which are incorporated herein and constitute part of this specification, illustrate example embodiments of the invention, and together with the general description given above and the detailed description given below, serve to explain the features of the invention.

FIG. 1 is a component block diagram of an example computing device suitable for implementing an embodiment.

FIG. 2 is a component block diagram of an example multi-core processor suitable for implementing an embodiment.

FIG. 3 is a functional and component block diagram of a system-on-chip suitable for implementing an embodiment.

FIG. 4 is a diagram of task-based handling of repetitive processes in accordance with an embodiment.

FIGS. 5A-C are diagrams of task-based handling of repetitive processes in accordance with an embodiment.

FIG. 6 is a state diagram of task-based handling of repetitive processes in accordance with an embodiment.

FIG. 7 is a state diagram of task-based handling of repetitive processes in accordance with an embodiment.

FIG. 8 is a chart diagram of task-based handling of repetitive processes in accordance with an embodiment.

FIG. 9 is a process flow diagram illustrating an embodiment method for task-based handling of repetitive processes.

FIG. 10 is a process flow diagram illustrating an embodiment method for dividing a partition of repetitive process iterations into subpartitions in task-based handling of repetitive processes.

FIG. 11 is a process flow diagram illustrating an embodiment method for dividing a partition of repetitive process iterations into subpartitions in task-based handling of repetitive processes.

FIG. 12 is a component block diagram illustrating an example of a computing device suitable for use with the various embodiments.

FIG. 13 is a component block diagram illustrating another example computing device suitable for use with the various embodiments.

4

FIG. 14 is a component block diagram illustrating an example server device suitable for use with the various embodiments.

DETAIL DESCRIPTION

The various embodiments will be described in detail with reference to the accompanying drawings. Wherever possible, the same reference numbers will be used throughout the drawings to refer to the same or like parts. References made to particular examples and implementations are for illustrative purposes, and are not intended to limit the scope of the invention or the claims.

The term "computing device" is used herein to refer to any one or all of cellular telephones, smartphones, personal or mobile multi-media players, personal data assistants (PDAs), personal computers, laptop computers, tablet computers, smartbooks, ultrabooks, palm-top computers, wireless electronic mail receivers, multimedia Internet enabled cellular telephones, wireless gaming controllers, desktop computers, computer servers, data servers, telecommunication infrastructure rack servers, video distribution servers, application specific servers, and similar personal or commercial electronic devices which include a memory, and one or more programmable multi-core processors.

The terms "system-on-chip" (SoC) and "integrated circuit" are used interchangeably herein to refer to a set of interconnected electronic circuits typically, but not exclusively, including multiple hardware cores, a memory, and a communication interface. The hardware cores may include a variety of different types of processors, such as a general purpose multi-core processor, a multi-core central processing unit (CPU), a multi-core digital signal processor (DSP), a multi-core graphics processing unit (GPU), a multi-core accelerated processing unit (APU), and a multi-core auxiliary processor. A hardware core may further embody other hardware and hardware combinations, such as a field programmable gate array (FPGA), an application-specific integrated circuit (ASIC), other programmable logic device, discrete gate logic, transistor logic, performance monitoring hardware, watchdog hardware, and timer references. Integrated circuits may be configured such that the components of the integrated circuit reside on a single piece of semiconductor material, such as silicon. Such a configuration may also be referred to as the SoC components being on a single chip.

In an embodiment, a process executing in a scheduler, within or separate from an operating system, for a multi-processor or multi-core processor system may reduce the overhead of repetitive processes (e.g., loops) in task-based run-time systems employing parallel processing across multiple processors or processor cores, or load balancing one execution of one or more repetitive processes across the multiple processors or processor cores. In an embodiment, the iteration space of a repetitive process may be stored in a descriptor table in a memory location, along with information regarding the function to be executed at each iteration of the repetitive process. In an embodiment, the repetitive process may have a criterion to execute until a relationship to a value it is realized. The relationship between the repetition value and the value it may be any arithmetic or logical relationship. To employ parallel processing of the repetitive process, tasks may be initialized for subsets, or partitions, of the criterion. For example, if the criterion is to repeat the repetitive process for each value between a starting value and the value it by incrementing the

US 9,501,328 B2

5

repetition value until it equals n , then the task may be assigned a subset of the repetitions between the starting value and the value n .

The number of tasks, represented here by p , and how they are assigned their respective subsets may vary. In at least one embodiment, the number of tasks may be equal to the number of available processors or processor cores. For example, with four available processors or processor cores (i.e., p=4), four subsets may be initialized, represented here by n0, n1, n2, and n3, and four tasks t may be initialized, represented here by t0, t1, t2, and t3. Each subset may be associated with a task t, such as n0 with t0, n1 with t1, n2 with t2, and n3 with t3. In an embodiment, information regarding the assignment of tasks, along with information about respective task criterion, may be stored in a master status table in a memory location. Information may be passed from the descriptor table to the status table to enable assignment iterations of the repetitive process to the one or more processors or processor cores.

During execution of the tasks, the computer system may store a pointer, or other type of reference, for each task to a local table in a memory location accessible by processor or processor core executing the task and indicating the progress of the task. With each iteration of the repetitive processes of the tasks, the respective pointers may be updated. In at least one embodiment, the local tables may be synched with the master status table to ensure that the master table reflects progress through the iteration space across all processors or processor cores.

In an embodiment in which the task completes its iterations, i.e., the repetition value for the task equals a final repetition value for the task's subset of n, the processor may discard the task. While at least one task may complete, one or more of the other tasks may continue to execute. Discarding the completed task may make the respective processor or processor core that executed the completed task available for other work. While at least one task is still executing, the scheduler may further divide the subset of the executing task into one or more new subsets, or subpartitions, and initialize one or more tasks to execute from the new subsets on the now available processor(s) or processor core(s). In an embodiment, rather than discarding completed tasks, while other tasks continue to execute, the scheduler may reassign the completed task to a new subset of the further divided subset.

FIG. 1 illustrates a system that may implement various embodiments. With respect to FIG. 1, a computing device 10 that may include an SoC 12 with a processor 14, a memory 16, a communication interface 18, and a storage interface 20. The computing device may further include a communication component 22 such as a wired or wireless modem, a storage component 24, an antenna 26 for establishing a wireless connection 32 to a wireless network 30, and/or the network interface 28 for connecting to a wired connection 44 to the Internet 40. The computing device 10 may communicate with a remote computing device 50 over the wireless connection 32 and/or the wired connection 44. The processor 14 may comprise any of a variety of hardware cores as described above. The SoC 12 may include one or more processors 14. The computing device 10 may include one or more SoC's 12, thereby increasing the number of processors 14. The computing device 10 may also include processors 14 that are not associated with an SoC 12. The processors 14 may each be configured for specific purposes that may be the same or different from other processors 14 of the computing device 10. Further, individual processors 14 may be multi-core processors as described below with reference to FIG. 2.

6

The computing device 10 and/or SoC 12 may include one or more memories 16 configured for various purposes. The memory 16 may be a volatile or non-volatile memory configured for storing data and processor-executable code for access by the processor 14. In an embodiment, the memory 16 may be configured to, at least temporarily, store data related to tasks of repetitive processes as described herein. In some embodiments, the iteration space splitter may be implemented with multiported SRAM cells to provide simultaneous memory access. As discussed in further detail below, each of the processor cores of the processor 14 may be assigned a task comprising a subset, or partition, of the iterations of the repetitive process by a scheduler of a high level operating system running on the computing device 10.

The communication interface 18, communication component 22, antenna 26 and/or network interface 28, may work in unison to enable the computing device 10 to communicate over a wireless network 30 via a wireless connection 32, and/or a wired connection 44 with the remote computing device 50. The wireless network 30 may be implemented using a variety of wireless communication technologies, including, for example, radio frequency spectrum used for wireless communications, to provide the computing device 10 with a connection to the Internet 40 by which it may exchange data with the remote computing device 50.

The storage interface 20 and the storage component 24 may work in unison to allow the computing device 10 to store data on a non-volatile storage medium. The storage component 24 may be configured much like an embodiment of the memory 16 in which the storage component 24 may store the data related to tasks of repetitive processes, such that the data may be accessed by one or more processors 14. The storage interface 20 may control access the storage device 24 and allow the processor 14 to read data from and write data to the storage device 24.

It should be noted that some or all of the components of the computing device 10 may be differently arranged and/or combined while still serving the necessary functions. Moreover, the computing device 10 may not be limited to use of each of the components, and multiple instances of each component, in various configurations, may be included in the computing device 10.

FIG. 2 illustrates a multi-core processor 14 suitable for implementing various embodiments. With respect to FIG. 2, the multi-core processor 14 may have a plurality of processor cores 200, 201, 202, 203. In an embodiment, the processor cores 200, 201, 202, 203 may be equivalent processor cores in that processor cores 200, 201, 202, 203 of a single processor 14 may be configured for the same purpose and to have the same performance characteristics. For example, the processor 14 may be a general purpose processor, and the processor cores 200, 201, 202, 203 may be equivalent general purpose processor cores. Alternatively, the processor 14 may be a graphics processing unit or a digital signal processor, and the processor cores 200, 201, 202, 203 may be equivalent graphics processor cores or digital signal processor cores, respectively. Through variations in the manufacturing process and materials, it may result that the performance characteristics of the processor cores 200, 201, 202, 203 may differ from processor core to processor core within the same multi-core processor 14 or another multi-core processor 14 using the same designed processor cores. In an embodiment, the processor cores 200, 201, 202, 203 may include a variety of processor cores that are non-equivalent. For example, some of the processor cores 200, 201, 202, 203 may be configured for the same or different purposes and to have the same or different perfor-

mance characteristics. In an embodiment, the processor cores 200, 201, 202, 203 may include a combination of equivalent and nonequivalent processor cores.

In the example illustrated in FIG. 2, the multi-core processor 14 includes four processor cores 200, 201, 202, 203 (i.e., processor core 0, processor core 1, processor core 2, and processor core 3); for ease of explanation, the examples herein may refer to the four processor cores 200, 201, 202, 203 illustrated in FIG. 2. However, it should be noted that FIG. 2 and the four processor cores 200, 201, 202, 203 illustrated and described herein are in no way meant to be limiting. The computing device 10, the SoC 12, or the multi-core processor 14 may individually or in combination include fewer or more than the four processor cores 200, 201, 202, 203.

FIG. 3 illustrates a computing device 10 having an SoC 12 including multiple processor cores 306, 308, 310, 312, 314 suitable for implementing various embodiments. With respect to FIGS. 1-3, the computing device 10 may also include a high level operating system 302, which may be configured to communicate with the components of the SoC 12 and operate a process or task scheduler 304 for managing the processes or tasks assigned to the various processor cores 306, 308, 310, 312, 314. In various embodiments, the task scheduler 304 may be a part of or separate from the high level operating system 302.

Different types of multi-core processors are illustrated in FIG. 3, including a high performance high leakage multi-core general purpose central processing unit (CPU); 306 referred to as a "high power CPU" core" in the figure; a low performance low leakage multi-core general purpose central processing unit (CPU); 308 referred to as a "low power CPU core" in the figure; a multi-core graphics processing unit (GPU) 310; a multi-core digital signal processor (DSP); 312, and other processor cores 314.

FIG. 3 also illustrates that processor cores 314 may be installed in the computing device after it is sold, such as in expansion or enhancement of processing capability or as an update to the computing device. After-market expansions of processing capabilities are not limited to central processor cores, and may be any type of computing module that may be added to or replaced in a computing system, including for example, additional, upgraded or replacement module processors, additional or replacement graphics processors (GPUs), additional or replacement audio processors, and additional or replacement DSPs, any of which may be installed as single-chip-multi-core modules or clusters of processors (e.g., one on Sat 1). Also, servers, such added or replaced processor components may be installed as processing modules (or blades) that plug into a receptacle and wiring harness interface.

Each of the groups of processor cores illustrated in FIG. 3 may be part of a multi-core processor 14 as described above. Moreover, these five example multi-core processors (i.e., groups of processor cores) are not meant to be limiting, and the computing device 10 or the SoC 12 may individually or in combination include fewer or more than the five multi-core processors 306, 308, 310, 312, 314 (or groups of processor cores), including types not displayed in FIG. 3.

Various embodiments may include methods, systems, and devices for allowing applications to implement iteration space splitters in hardware. The execution of partitions of repetitive processes across multiple processor cores may be assisted or controlled through the use of instruction set architecture (ISA) extensions, special register read/write instruc-

tions, designated memory addresses, via memory mapped input/output (MMIO) accelerator devices and/or by using one or more coprocessors.

FIG. 4 illustrates an embodiment workflow of task-based handling of repetitive processes. An embodiment of an iteration space splitter implemented in hardware may include an iteration space splitter controller 402 ("central controller"), which may control initialization, updating, and deletion of entries of an iteration space descriptor table 404 ("descriptor table") in a memory location. An iteration space splitter implemented in hardware may also include an iteration space status table 406 ("status table") in a memory location responsible for tracking execution of iterations of one or more repetitive processes on multiple processor cores 410, 412, 414, 416. A local controller may be associated with each of the processor cores 410, 412, 414, 416 to control tracking of repetitive process iterations assigned in a respective processor core and relay status information to the descriptor table 404 and status table 406. Local tables assigned to memory locations may correspond to each of the processor cores 410, 412, 414, 416 and may hold information regarding progress of individual iterations of partitions of one or more repetitive processes.

In various embodiments, software may require the execution of multiple repetitive processes (e.g., loops) during regular operation. To improve speed of execution completion, modern processors may enable load balancing across multiple processor cores by splitting the iteration space and assigning portions of the repetitive process execution to different processor cores. Because the number of iterations required to complete execution of a repetitive process may vary throughout execution, iteration space splitting techniques must be dynamic. Software based dynamic partitioning schemes require the storage of the iteration space in shared memory, which must be inspected by processor cores to determine if partitions may be subdivided further. Such approaches may lead to cache pollution resulting from numerous processor core inspection and synchronization efforts, as well as coherence traffic.

Various embodiments may include methods, systems, and devices for implementing the dynamic iteration space splitting scheme in hardware using a descriptor table 404, a status table 406, and multiple local tables, all located in cache memory. The central controller 402 may partition the iterations of the repetitive process and assign each partition to a different processor or processor core 410, 412, 414, 416. In doing so, the computing device may process the partitions in parallel. Unlike software based solutions, a hardware based approach enables parallel messaging between the central controller and the processor or processor cores, efficient iteration status lookup, and parallel work stealing. For example, multiple processor or processor cores may request and be assigned subpartitions of a partition being executed on another processor or processor core, at the same time and without interrupting execution. Thus, the implementation of an iteration space splitter in hardware may enable repetitive process scheduling, assignment and execution without requiring intervention from the run-time or operating systems, and consequently may improve the function of a computing device implementing various embodiments.

The central controller may determine the size of various partitions, and to which processor or processor core to assign each partition based on various criteria. For example, the central controller may size and allocate partitions based on the type, the performance characteristics, and/or the availability of the processor or processor core, and/or the type,

the resource requirements, and/or the latency tolerance of the execution of repetitive processes. Granularity of partition size may be any size, including very small numbers of iterations, thereby enabling precision load balancing across multiple processor cores. Further, partition granularity may be dynamic and may be adjusted by the central controller as needed during execution without interrupting already executing partitions of the repetitive process.

FIGS. 5A-C illustrate an embodiment of an iteration space splitter for task-handling of execution of repetitive processes. Various embodiments of an iteration space splitter for load-balancing of execution of repetitive processes across multiple processor cores may include a descriptor table 404 and a status table 406 implemented in memory. In some embodiments, the descriptor table 404 may be primarily purposed with maintaining metadata for one or more iteration spaces being executed or queued for execution by the multiple processor cores. In some embodiments, the status table 406 may be responsible for maintaining status information for partitions of one or more iteration spaces that are being executed or are queued for execution of the multiple processor cores. Respective assignments of multiple partitions of the iteration space of a repetitive process may be maintained in the fields of the status table 406 until all iterations of all partitions are completed.

In various embodiments, the descriptor table 404 may have multiple fields for maintaining metadata for multiple iteration spaces. A non-limiting example of an iteration space splitter 500 including a descriptor table 404 and a status table 406 is shown in FIG. 5A. A first field of the descriptor table 404 may be designated for maintaining iteration space identifiers. At the time of initialization, each iteration space may be assigned an identifier, which may be used to track execution progress across the multiple processor cores. A second field may be designated for minimum partition size, the minimum partition size or minimum split size may be an optional parameter that indicates the smallest number of iterations that may be assigned to any partition. Thus, a partition of minimum size may not be subdivided further. A third field may be designated for iteration space function descriptors. In some embodiments, function descriptors may be the instructions to be executed in each iteration of the iteration space of the repetitive process. In some embodiments, the function descriptor may be a stub describing the instructions to be executed, or a pointer to the instructions. In heterogeneous devices, the function descriptor may be a table of pointers to different instructions. A fourth field may be designated for the expected number of iterations, which may be calculated from beginning and ending iterations input. A fifth field may be designated for the number of iterations completed, and may be updated as execution of repetitive process partitions progresses.

In various embodiments, the status table 406 may have several fields maintaining information about repetitive process partition assignments. Some embodiments may include a first field designated for iteration space identifiers. Iteration space identifiers may be generated during initialization of the iteration space, and may be used by the central controller to cross-correlate one or more partitions of a repetitive process. A second field may designate the processor core identifier, which may be an identifier from a group of predetermined identifiers, each assigned to a different processor core. A third field may designate the beginning iteration count, and a fourth field may designate the end iteration count. Thus, a processor core assigned iterations 1-40 of a repetitive process may have a different set of table

entries than a processor core assigned to execution iterations 41-80, even though their overall partition size may be the same.

Multiple atomic operations may be included to enable insertion, removal, and editing of table entries. As shown in FIG. 5B, an initializing operation (e.g., an initialize_its operation) may provide the iteration space splitter with parameters, such as beginning of iterations, end of iterations, function to be executed, and an optional minimum split size. This information may be added to the descriptor table 404, the expected iterations calculated, and an iteration space identifier generated. The iteration space identifier, beginning and end information, and function to be executed may be used by the central controller to split the iteration space into multiple partitions. The central controller may assign the multiple partitions to one or more processor cores depending upon the size of the iteration space partitions and processor core availability. Assignment information may be maintained in the status table 406 and individual partition information passed in local tables 506 associated with each of the processors or processor cores. As shown in FIG. 5C, each local table 506 may include entries for beginning and end criteria of a partition as well as a pointer for tracking the current iteration or previous iteration of the task executing iterations of the partition.

Subpartitions of active iteration space partitions may be created by the central controller and implemented using a "steal_its" operation. A processor core that completes all its assigned iterations may be available for additional work. The central controller executing a steal_its operation may "steal" a portion of an active partition from another processor core and reassign it to the available processor core, thereby reducing the load on the active core and increasing the speed of overall execution. An "update_its" operation may be used to update the information in the iteration space status table 406 and descriptor table 404 when work is stolen, completed, or new iterations are found. Updates to the start and end iterations of a partition may be passed to the local table 506 to enable the local controller to track progress of task execution through the partition.

Partitions that the central controller deems completed may be checked against local tables of the processor cores using a "finish_its" operation. Completed partitions may optionally remain in the status table until all partitions are complete and the iteration space is ready for deletion from the descriptor table 404 and the status table 406. In this manner, the central controller may manipulate partition sizes, update iteration fields, and delete entries associated with completed partitions of iteration spaces.

FIG. 6 illustrates an embodiment of workflow states of a task-based handling of repetitive processes. With reference to FIGS. 1-6, the descriptor table 404 may be implemented in a cache memory of the computing device 10 and may have elements that transition through multiple states during operation. The descriptor table may be empty 602 prior to detection by executing software of any repetitive processes. Once one or more repetitive processes are discovered and deemed ready for execution, an iteration space may be initialized, such as by the initialize_its operation. Information about the repetitive process may be maintained in the table as discussed with respect to FIG. 5. After entries for a particular iteration space are added to the descriptor table 404, the iteration space may be referred to as initialized 604. Information associated with the initialized iteration space may be passed in the status table 406 for partitioning of the execution assignments.

US 9,501,328 B2

11

During execution of tasks handling iteration executions, the processor cores in the requesting software program may discover additional iterations of the repetitive process. For example, if an end executive criterion is met, the repetitive process may repeat, such as in a 'while' loop with an unsatisfied condition. In some embodiments, one or more local controllers associated with each of the multiple processors may pass a message to the central controller indicating that additional iterations have been discovered and a local table has been updated to reflect the new end execution criterion. The central controller 402 may update the descriptor table 404 to reflect the additional iterations, thereby modifying the expected number of iterations. Modifications to the descriptor table 404 entries may transition the respective iteration space into an updated state 606. In some embodiments, the iteration space may enter the updated state 616 several times during execution of the iteration space. In some embodiments, the iteration space may never enter an updated state 606 and may proceed directly from an initialized state 604 to an in-progress state 608, or a done state 610.

In various embodiments, the iteration spaces maintained in the descriptor table 404 may enter an in-progress state 608 after being initialized or updated. In some embodiments, a finish operation may indicate that execution of a partition should commence or continue, and may place an iteration space in an in-progress state 608. An in-progress state may indicate that at least one partition of the iteration space has not completed executing all iterations of a repetitive process. Conversely, if all iterations of all partitions of a repetitive process have finished executing then the iteration space is done 610 and may be deleted from the descriptor table 404.

FIG. 7 illustrates an embodiment of workflow states of a task-based handling of repetitive processes. The status table 406 may be implemented in a cache memory of the computing device 10 and may have elements that transition through multiple states during operation. Prior to partitioning of iterations of a repetitive process, the status table elements may be in an empty state 702. In various embodiments, when an iteration space is initialized 604, entries may be made in the descriptor table 404 and an iteration space identifier may be generated. A portion of the information entered into the descriptor table 404 along with the iteration space identifier may be added to the status table 406, thereby placing the entry in the status table into an initialized state 704. The central controller 402 may determine whether to partition the iterations of the iteration space based on a variety of factors. If no partitioning is needed to complete all of the iterations of the repetitive process, the central controller may initialize a single task for executing the repetitive process and assign the task to a single processor core. Once the task has finished executing and all iterations are complete, the iteration space may transition to a done state 708.

The central controller may determine that the best execution of the iteration space requires dividing the workload of executing across multiple processor cores. In various embodiments, the central controller may compare the iteration space size against the minimum split size stored in the descriptor table 404 and may generate multiple tasks and portions having one number of iterations (e.g., any number greater than the minimum split size). In an embodiment, the number of tasks may be equal to the number of partitions as described above, or to the number available processors or processor cores. For example, with four available processors or processor cores (see FIG. 2) (i.e., p=4), four tasks may be initialized. Each task may be associated with a processor or processor core. For example, task 0 may be associated with processor core 0, task 1 with processor core 1, task 12 with

12

processor core 1, and task 16 with processor core 3. Some embodiments may include consideration of each processor core's current workload prior to partitioning.

As processor cores become available to receive new tasks, they may request work from the central controller, which may subdivide existing partitions to reduce load on an active processor core and provide work for the available processor core. Upon completion of a task, or completion of the iterations assigned to the task, the processor or processor core that executed the task may be available for executing further tasks. The remaining tasks having more than one iteration of the respective partition of the repetitive process to execute may be re-signed, a subpartition of the respective partition, and the completed task may be assigned another subpartition of the same partition. Thus, in an embodiment, a partition assigned to a task, where the task has yet to begin executing, at least the last iteration of the partition, may be split into subpartitions so that one or more of the yet to be executed iterations of the partition may be reassigned to an available processor or processor core to increase the speed of executing the iterations of a repetitive process. This reallocation is referred to herein as "work stealing" and may be initiated using a steal_its_operation.

With partitioning and subpartitioning of the iteration space may transition the partition into a stolen state 706. Stolen partitions may be continuously subpartitioned so long as the minimum split size is honored, if one is specified during initialization. Once all tasks are finished executing their respective partitions, the partition may enter the done state 708. Completion of partitions may be passed in a message to the descriptor table, which may be updated to indicate that an iteration space is in progress 608 or done 610.

FIG. 8 illustrates an example chart 800 of task-based handling of repetitive processes. The chart 800 illustrates an example time progression of the states of four processors or processor cores (processor 0, processor 1, processor 2, and processor 3) implementing task-based handling of repetitive processes. The use of four processors or processor cores in this example is not meant to be limiting, and similar task-based handling of repetitive processes may be implemented using more or fewer than four processors or processor cores. In the example illustrated in row 802 of the chart 800, each of the processors or processor cores may be assigned a respective partition of iterations of a repetitive process. Processor 0 may be assigned partition 0, processor 1 may be assigned partition 1, processor 2 may be assigned partition 2, and processor 3 may be assigned partition 3. In row 804, tasks may be initialized for executing the iterations of the respective partitions of the repetitive process assigned to each processor or processor core. In this example, task 0 may be initialized for partition 0 and processor 0, task 1 may be initialized for partition 1 and processor 1, task 12 may be initialized for partition 2 and processor 2, and task 16 may be initialized for partition 3 and processor 3.

Also in row 806, a pointer or other reference type may be initialized for the respective tasks. In various embodiments, the pointer may be maintained in a local table associated with each of the processors or processor cores. Each local table may have entries for beginning and ending criteria of the iteration space partition, and may maintain a pointer to the most recently executed iteration. In this example, pointer 0 may be initialized for task 0, pointer 1 may be initialized for task 12, pointer 2 may be initialized for task 16, and pointer 3 may be initialized for task 16. The pointers may be used to track the progress of the execution of the repetitive processes for their respective tasks, and the pointers may be

US 9,501,328 B2

13

cessible by the central controller for use in determining which partitions have sufficient remaining iterations to permit subpartitioning. In an embodiment, a pointer may be initialized for each of one or more repetitive processes for each task. The task may access the pointer of the respective task to identify the repetitive process iteration of the task when instructed to execute.

In row 808, each of the processors or processor cores may begin to execute their respective tasks. Executing the tasks may include executing the assigned partitions of the iterations of the repetitive processes.

In row 810, the processors or processor cores may update the respective pointers to indicate the start or completion of execution of the repetitive processes of the respective tasks. Throughout the execution of the tasks, the pointers may be repeatedly updated to indicate the iteration of the repetitive processes for being executed. In some embodiments, the central controller may query the local table for pointer position and use this information to update the descriptor table and status table. In some embodiments, a local controller associated with each of the processors or processor cores, may push pointer position in the central controller at regular intervals. The central controller may use the received information to update the descriptor table and status table. In some embodiments, a local controller associated with each of the processors or processor cores may use the pointer positions to calculate iteration completed, and update the descriptor table and status table directly.

In this example, processor 2 and processor p may implement different schemes. The scheme for processor 2 may include discarding the completed task 12 in row 820 after the task finished in row 816. In row 822, processor 2 may be assigned a subpartition of one of the ongoing tasks being executed by another of the processors or processor cores. The subpartition may be one or more iterations of the repetitive process that has yet to be executed by one of the ongoing tasks. The partition of the remaining iterations of the ongoing task may be divided into two or more subpartitions, and the subpartitions may be assigned to tasks. One of the subpartitions may be assigned to the original task of the partition, and the other subpartition(s) may be assigned to other new or existing but completed tasks. In this example, partition II of ongoing task 10 being executed on processor II may include unexecuted iterations of the repetitive process. Partition II may be divided into two subpartitions, one of which may be assigned to processor 0 and task 10, and the other may be assigned to processor 2 and a newly initialized task tp+1 in rows 822 and 824. Much like above, in row 824 processor 2 may initialize a pointer for other reference types for the respective task, and in row 826 processor 2 may begin executing task tp+1. In an embodiment, initializing the pointer may involve initializing a new pointer for the task, or updating the existing pointer. Also as described above, in row 828 during the execution of task tp+1, the respective pointer for task tp+1 may be updated for the current or last executed iteration of the repetitive process.

The scheme for processor p differs from the scheme for processor 2 described above in that rather than discarding the completed task and initializing a new task to execute a subpartition of the iterations of the repetitive process, processor p uses the existing completed task. In this example, Partition I of ongoing task 10 being executed on processor I may include unexecuted iterations of the repetitive process. Partition I may be divided into two subpartitions, one of which may be assigned to processor 1 and task t1, and the other of which may be assigned to processor p and existing

14

completed task tp in row 820. Much like above, in row 822 processor p may begin executing task tp for the subpartition, and update the respective pointer for the iteration of the repetitive process for task tp in row 824. In this example scheme, there is no need to initialize a new pointer as they both may exist from the previous execution of task tp; however, a new pointer may be initialized if so desired.

For the respective scheme implemented to engage the available processor or processor core with further task execution, several of the states in the above described rows 818, 820, 822, 824, 826 and 828 may be repeated to complete execution of the tasks for all of the iterations of the respective subpartitions of the repetitive process, each of the processors or processor cores. Depending on various factors such as the times described above, one or more of the tasks may complete executing at the same or different times. For example, in row 834, task tp may finish executing, while tasks 0, 1, and tp+1 may continue to execute, but in row 842 tasks 0, tp+1, and tp may finish at the same time, while task 10 may continue to execute. In an embodiment, where only one ongoing task remains, and the ongoing task is executing the final iteration of its partition of the iterations of the repetitive process, the partition cannot be subpartitioned to assign iterations of the repetitive process to the available processors or processor cores.

In an embodiment, the central controller may assign further subpartitions of an ongoing repetitive process as processors or processor cores become available. For example, in row 834 task tp may finish executing and processor tp may become available for execution of other iterations of the repetitive process. Partition 0 may be the largest remaining partition despite previous subpartitioning in row 822. As such, task tp+1's subpartition may be further subpartitioned and assigned a partition re-assigned to task tp+1 and the remaining partition assigned to task tp in row 836. In an embodiment, tasks tp+1 and tp may execute their respective iterations of partition II in row 838. In row 840 the tasks may update respective pointers to keep track of the started or completed iterations of the repetitive process. Upon completion of all iterations of assigned partitions of a repetitive process, the completed tasks from row 842, task 10, task tp+1 and task tp, may be discarded in row 844.

While the final ongoing task continues to execute its last iteration, several of the states in the above described rows 45 may be repeated to aid in executing the iterations of the repetitive process when necessary. In row 850 the final ongoing task, task 10 in this example, may complete its execution. With no remaining repetitive process iterations, task 10 may be discarded in row 852.

It should be noted that the various described states of the processors or processor cores may occur in a different order than in the examples described herein. The descriptions of FIGS. 4-8 are not meant to be limiting as to the order or number of processors or processor cores, states, tasks, shadow tasks, partitions, subpartitions, pointers or other relevance types, iterations, processes, or any other element described herein.

FIG. 9 illustrates an embodiment method 900 for task-based handling of repetitive processes. The method 900 may be executed by one or more processors or processor cores of the computing device. While running programs in a task-based run-time system, in block 902 the processor or processor core may encounter an repetitive process, or loop, in a program. In block 904, an iteration space for the repetitive process may be initialized with respect to a descriptor table and a status table. Iteration space initialization may include generation by a central controller (i.e., the iteration space

Splitter controller of FIG. 4, which may also be called an "ISS controller" of an iteration space identifier, and information regarding the beginning and end criteria of the repetitive process, and a descriptor for a function to be executed at each iteration of the repetitive process. Initialization of descriptor table entries may further include enumeration of the number of expected iterations. In some embodiments, a minimum split size may be defined by a processor or programmer, and may indicate the smallest number of iterations contained in a partition of the repetitive process. The generated information may be stored in the descriptor table and status table as described with reference to FIGS. 5A-5C.

In block 906, one or more tasks may be initialized for executing the repetitive process in parallel across multiple processors or processor cores. The number of tasks initialized to execute the repetitive process may vary. In an embodiment, the number of tasks initialized may be equal to a number of available processors or processor cores to which the tasks may be assigned as further described below. In other embodiments, the number of tasks may be determined by one or more factors including characteristics of the processors or processor cores, characteristics of the program and/or the repetitive process, and states of the computing device, including temperature and power availability.

In block 908, the iterations of the repetitive process may be divided by the central controller into partitions for execution as part of the initialized tasks in parallel on the multiple processors or processor cores. In an embodiment, the number of partitions may be determined based on the number of initialized tasks, or available processors or processor cores. The makeup of each partition may be determined by various factors including characteristics of the processors or processor cores, characteristics of the program and/or the repetitive process, and states of the computing device, including temperature and power availability. The partitions may equally as possible divide the number of iterations of the repetitive process, or the partitions may be unequal in number of iterations of the repetitive process. In various embodiments, the number of iterations in each partition may be bounded by a minimum split size, which may be optionally defined during iteration space initialization. If no minimum split size is defined for an iteration space, then the number of iterations for each partition may have significant granularity, such as a partition size of one or more iterations.

In block 910, the partitions of the repetitive process may be assigned to respective tasks. In block 912, the initialized tasks, and thereby the respective partitioned iterations of the repetitive process, may be assigned to respective processors or processor cores. Assignment of tasks to processors or processor cores may include the central controller updating the status table to indicate which partitions are executing on which processor or processor cores. In some embodiments, assignment of tasks to processors or processor cores may further include updating by the central controller or by local controllers associated with each of the processors or processor cores, a local table associated with each of the processors or processor cores to include start and end iteration criteria for the partition.

Such like initializing the tasks and partitioning the iterations, assignments to particular processors or processor cores may be determined by various factors including characteristics of the processors or processor cores, characteristics of the program and/or the repetitive process, and states of the computing device, including temperature and power availability.

In block 914, the processor or processor core may initialize a pointer (or other type of reference) for each task. The pointer may be accessible by a respective local controller and maintained in a respective local table. The pointer may be used to track the iterations of the repetitive processes so that the respective tasks know which iterations of the repetitive process to execute. In block 912, the assigned tasks may begin executing in parallel on the respective processors or processor cores to which the tasks are assigned. In block 918, the respective pointers for the tasks may be updated to reflect changes in the iterations of the repetitive processes of the executing tasks, such as completion or starting of an iteration of the repetitive processes.

In determining block 920, the processor or processor core may determine whether the iterations of the repetitive process for a respective task are complete, or that there are no remaining iterations for execution (i.e., determination block 920 "Yes"), the processor or processor core may continue to execute the respective task in block 924.

In response to determining that the iterations of the repetitive process for a respective task are complete, or that there are no remaining iterations for execution (i.e., determination block 920 "No"), the processor or processor core may determine whether the remaining iterations for another respective task are divisible in determination block 922. In determining whether the remaining iterations for the other respective task are divisible, the remaining iterations may be divisible when more than the executing iteration remain to be executed. Alternatively, the partition of the other respective task may be divisible if the remaining iterations number more than a defined minimum split size. The remaining iterations may be indivisible when only the executing iteration for the other respective task remains, or if the remaining iterations number fewer than a defined minimum split size.

In response to determining that the remaining iterations for the other respective task are divisible (i.e., determination block 922 "Yes"), depending on the implemented scheme the processor or processor core may divide the remaining iterations of the repetitive process into subpartitions, as described below in either method 1000 (see FIG. 10) or method 1100 (see FIG. 11). In response to determining that the remaining iterations for the respective task are indivisible (i.e., determination block 922 "No"), the processor or processor core may flush the task in block 926, and may discard the task and await new work. In some embodiments, the central controller or relevant local controller may update the status table to indicate that the task has finished execution of all assigned iterations. In some embodiments, the central controller may remove the partition information and associated entries from the status table once the task has finished executing and no ongoing tasks have divisible partitions.

FIG. 10 illustrates an embodiment method 1000 for dividing a partition of repetitive process iterations into subpartitions in task-based handling of repetitive processes. The method 1000 may be executed by one or more processors or processor cores of the computing device. As described above with reference to FIG. 9, the method 1000 may be invoked in response to the processor determining that the iterations of the repetitive process for a respective task are complete (i.e., determination block 920 "Yes") and that the remaining iterations for another respective task are divisible (i.e., determination block 922 "Yes"). In other words, method 1000 may be invoked when a task running on a processor or processor core completes its execution and

another task running on another processor or processor core is ongoing and has more iterations than just the executing iteration, or a minimum split size remaining.

In block 1002, the completed task may be discarded. In block 1004, the iterations of the ongoing task may be divided into subpartitions of the partition of iterations assigned to the ongoing task. The central controller may perform a lookup in the status table to determine the processors or processor cores that are still executing tasks associated with the iteration identifier linked to the completed task. The central controller may then subpartition one or more of the partitions executing on other processors or processor cores and assign the one or more new partitions to the available processor core. The status table may be updated by the central controller, and a local table associated with the available processor or processor core may be updated to include either new start and end execution criterion for the newly assigned partition. Similarly, the local table pointer may be reset so as to track the execution progress of the task through the iterations of the new partition. For example, a partition of iterations of a repetitive process assigned to a task may include 500 iterations. In such an example, the ongoing task may have executed 174 iterations, and the task may be executing the 175th iteration, leaving 325 iterations yet to be executed.

With resources such as processor and processor cores being available to aid in executing these remaining iterations of the task, the remaining 325 iterations may be divided into subpartitions of the original 500 iteration partition or what is now the 325 remaining iterations partition. In this example, one or more processors or processor cores may be available, and the remaining 325 iterations may be divided up in any manner over any number of the available processors or processor cores. For instance, the remaining iterations may be divided equally or unequally over the available processors or processor cores, and it is possible that at least one available processor or processor core is not assigned a subpartition of the remaining iterations. Further, the processor or processor core executing the task with the remaining iterations may be assigned at least the executing iteration of the task at the time the remaining iterations are divided. How the remaining iterations are divided into subpartitions may depend on a variety of factors including whether a minimum split size was defined at the time of iteration space execution characteristics of the processors or processor cores (e.g., relative processing speed, relative power efficiency, current leakage, etc.), characteristics of the program and/or the repetitive process, and states of the computing device, including temperature and power availability (e.g., un-battery or charging).

In block 1006, tasks may be initialized for the remaining unassigned subpartitions. In block 1008, one subpartition may be assigned to the ongoing task for which the iterations are being divided. Thus, all of the subpartitions get assigned to either the existing ongoing task or a newly initialized task for executing on the available processor(s) or processor core(s).

In determination block 1010, the processor or processor core may determine whether the task is an ongoing task or a new task. In response to determining that the task is an ongoing task (i.e., determination block 1010 “Yes”), the processor or processor core executing the ongoing task may continue executing the task in block 924 (see FIG. 9). In response to determining that the task is not an ongoing task (i.e., determination block 1010 “No”), and thus is a new task, the processor or processor core assigned to execute the

new task may execute the task in block 916 as described above with reference to FIG. 9.

FIG. 11 illustrates an embodiment method 1100 for dividing a partition of repetitive process iterations into subpartitions for task-based handling of repetitive processes. With reference to FIGS. 1-11, the method 1100 may be executed by one or more processors or processor cores of the computing device. As described above with reference to FIG. 9, the method 1100 may be invoked in response to determining that the iterations of the repetitive process for a respective task are complete (i.e., determination block 920 “Yes”) and that the remaining iterations for another respective task are divisible (i.e., determination block 922 “Yes”). In other words, method 1100 may be invoked when a task running on a processor or processor core completes its execution, and another task running on another processor or processor core is ongoing and has more iterations than just the executing iteration or a minimum split size remaining. This is similar to the method 1000 described with reference to FIG. 10; however, rather than discard the completed task, as in block 1002 (see FIG. 10), the respective processor or processor cores may retain the completed tasks to execute the remaining iterations of the repetitive process.

In block 1102, the remaining iterations of an ongoing task may be divided into subpartitions much like in block 1004 described above with reference to FIG. 10. In block 1104, one of the subpartitions containing portions of the remaining iterations of the ongoing task may be assigned to the ongoing task to complete executing a reduced portion of its original partition of the iterations of the repetitive process. In block 1106, the remaining unassigned subpartitions may be assigned to the existing completed tasks. Thus, all of the subpartitions get assigned to either the existing ongoing task or an existing completed task for executing on the available processor(s) or processor core(s). The processor or processor core for executing each task may proceed to continue executing the task in block 924 as described above with reference to FIG. 9.

FIG. 12 illustrates an example of a computing device suitable for implementing the various embodiments, for instance, some or all of the methods illustrated in FIGS. 9-11. A smartphone computing device 1200 may include a multi-core processor 1202 coupled to a touchscreen controller 1204 and an internal memory 1206. The multi-core processor 1202 may be one or more multi-core integrated circuits designated for general or specific processing tasks. The internal memory 1206 may be volatile or non-volatile memory, and may also be secure and/or encrypted memory, or insecure and/or unencrypted memory, or any combination thereof. The touchscreen controller 1204 and the multi-core processor 1202 may also be coupled to a touchscreen panel 1202, such as a resistive-sensing touchscreen, capacitive-sensing touchscreen, infrared sensing touchscreen, etc. Additionally, the display of the computing device 1200 need not have touch screen capability.

The smartphone computing device 1200 may have one or more radio signal transceivers 1208 (e.g., Peanut, Bluetooth, Zigbee, Wi-Fi, RF radio) and antenna 1210, for sending and receiving communications, coupled to each other and/or to the multi-core processor 1202. The transceivers 1208 and antenna 1210 may be used with the above-mentioned circuitry to implement the various wireless transmission protocol stacks and interfaces. The smartphone computing device 1200 may include a cellular network wireless modem chip 1216 that enables communication via a cellular network and is coupled to the processor.

The smartphone computing device 1200 may include a peripheral device connection interface 1218 coupled to the multi-core processor 1202. The peripheral device connection interface 1218 may be singularly configured to accept one type of connection, or may be configured to accept various types of physical and communication connections, creation or proprietary, such as USB, FireWire, Thunderbolt, or PCIe. The peripheral device connection interface 1218 may also be coupled to a similarly configured peripheral device connection port (not shown).

The smartphone computing device 1200 may also include speakers 1214 for providing audio outputs. The smartphone computing device 1200 may also include a housing 1220, constructed of a plastic, metal, or a combination of materials, for containing all or some of the components disclosed herein. The smartphone computing device 1200 may include a power source 1222 coupled to the multi-core processor 1202, such as a disposable or rechargeable battery. The rechargeable battery may also be coupled to the peripheral device connection port to receive a charging current from a source external to the smartphone computing device 1200. The smartphone computing device 1200 may also include a physical button 1224 for receiving user inputs. The smartphone computing device 1200 may also include a power button 1226 for turning the smartphone computing device 1200 on and off.

The various embodiments described above, for instance, some or all of the methods illustrated in FIGS. 9-11, may also be implemented within a variety of other computing devices, such as a laptop computer 1300 illustrated in FIG. 13. With reference to FIGS. 1-13, a laptop computer may include a touchpad touch surface 1317 that serves as the computer's pointing device, and thus may receive drag, scroll, and flick gestures similar to those implemented on computing devices equipped with a touch screen display and described above. A laptop computer 1300 will typically include a multi-core processor 1311 coupled to volatile memory 1312 and a large capacity nonvolatile memory, such as a disk drive 1313 or flash memory. Additionally, the computer 1300 may have one or more antenna 1308 for sending and receiving electromagnetic radiation that may be connected to a wireless data link and/or cellular telephone transceiver 1316 coupled to the multi-core processor 1311. The computer 1300 may also include a floppy disc drive 1314 and a compact disc (CD) drive 1315 coupled to the multi-core processor 1311. In a notebook configuration, the computer housing includes the touchpad 1317, the keyboard 1318, and the display 1319 all coupled to the multi-core processor 1311. Other configurations of the computing device may include a computer mouse or trackball coupled to the processor (e.g., via a USB input) as are well known, which may also be used in conjunction with the various embodiments. A desktop computer may similarly include these computing device components in various configurations, including separating and combining the components at one or more separate but connectable parts.

The various embodiments, for instance, some or all of the methods illustrated in FIGS. 9-11, may also be implemented on any of a variety of commercially available server devices, such as the server 1400 illustrated in FIG. 14. With reference to FIGS. 1-14, a server 1400 typically includes one or more multi-core processor assemblies 1401 coupled to volatile memory 1402 and a large capacity nonvolatile memory, such as a disk drive 1404. As illustrated in FIG. 14, multi-core processor assemblies 1401 may be added to the server 1400 by inserting them into the racks of the assembly. The server 1400 may also include a floppy disc drive, compact disc

(CD or DVD) disc drive 1406 coupled to the processor 1401. The server 1400 may also include network access ports 1403 coupled to the multi-core processor assemblies 1401 for establishing network interface connections with a network 1405, such as a local area network coupled to other broadcast system endpoints and servers, the Internet, the public switched telephone network, and/or a cellular data network (e.g., CDMA, TDMA, GSM, PCS, 3G, 4G, LTE, or any other type of cellular data network).

Computer program code or "program code" for executing on a programmable processor for carrying out operations of the various embodiments may be written in a high level programming language such as C, C++, C#, Smalltalk, Java, JavaScript, Visual Basic, a Structured Query Language (e.g., Transact-SQL), Perl, in various other programming languages. Program code or programs stored on a computer readable storage medium as used in this application may refer to machine language code (such as object code) whose format is understandable by a processor.

Many computing devices operating system kernels are organized into a user space (in which non-privileged code runs) and a kernel space (in which privileged code runs). This separation is of particular importance in Android and other general purpose license (GPL) compliant environments where code that is part of the kernel space must be GPL licensed, while code running in the user-space may not be GPL licensed. It should be understood that the various software components/modules discussed here may be implemented in either the kernel space or the user space, unless expressly stated otherwise.

The foregoing method descriptions and the process flow diagrams are provided merely as illustrative examples and are not intended to require or imply that the operations of the various embodiments must be performed in the order presented. As will be appreciated by one of skill in the art, the order of operations in the foregoing embodiments may be performed in any order. Words such as "herein," "then," "next," etc., are not intended to limit the order of the operations, these words are simply used to guide the reader through the description of the methods. Further, any reference to claim elements in the singular, for example, using the articles "a," "an" or "the" is not to be construed as limiting the element to the singular.

The various illustrative logical blocks, modules, circuitry, and algorithm operations described in connection with the various embodiments may be implemented as electronic hardware, computer software, or combinations of both. To clearly illustrate this interchangeability of hardware and software, various illustrative components, blocks, modules, circuitry, and operations have been described above generally in terms of their functionality. Whether such functionality is implemented as hardware or software depends upon the particular application and design constraints imposed on the overall system. Skilled artisans may implement the described functionality in varying ways for each particular application, but such implementation decisions should not be interpreted as causing a departure from the scope of the present invention.

The hardware need to implement the various illustrative logics, logical blocks, modules, and circuitry described in connection with the embodiments disclosed herein may be implemented or performed with a general purpose processor, a digital signal processor (DSP), an application specific integrated circuit (ASIC), a field programmable gate array (FPGA), or other programmable logic device, discrete gate or transistor logic, discrete hardware components, or any combination thereof designed to perform the functions described

herein. A general-purpose processor may be a microprocessor, but, in the alternative, the processor may be any conventional processor, controller, microcontroller, or state machine. A processor may also be implemented as a combination of computing devices, e.g., a combination of a DSP and a microprocessor, a plurality of microprocessors, one or more microprocessors in conjunction with a DSP core, or any other such configuration. Alternatively, some operations or methods may be performed by circuitry that is specific to a given function.

In one or more embodiments, the functions described may be implemented in hardware, software, firmware, or any combination thereof. If implemented in software, the functions may be stored as one or more instructions or code on a non-transitory computer-readable medium or a non-transitory processor-readable medium. The operations of a method or algorithm disclosed herein may be embodied in a processor-executable software module that may reside on a non-transitory computer-readable or processor-readable storage medium. Non-transitory computer-readable or processor-readable storage media may be any storage media that may be accessed by a computer or a processor. By way of example but not limitation, such non-transitory computer-readable or processor-readable media may include RAM, ROM, EEPROM, FLASH memory, CD-ROM or other optical disk storage, magnetic disk storage or other magnetic storage devices, or any other medium that may be used to store desired program code in the form of instructions or data structures and that may be accessed by a computer disk and disc, as used herein, includes compact disc (CD), laser disc, optical disc, digital versatile disc (DVD), floppy disk, and Blu-ray disc, wherein disks usually reproduce data magnetically, while discs reproduce data optically with lasers. Combinations of the above are also included within the scope of non-transitory computer-readable and processor-readable media. Additionally, the operations of a method or algorithm may reside as one or any combination or set of codes and/or instructions on a non-transitory processor-readable medium and/or computer-readable medium, which may be incorporated into a computer program product.

The preceding description of the disclosed embodiments is provided to enable any person skilled in the art to make or use the present invention. Various modifications to these embodiments will be readily apparent to those skilled in the art, and the generic principles defined herein may be applied to other embodiments without departing from the spirit or scope of the invention. Thus, the present invention is not intended to be limited to the embodiments shown herein but is to be accorded the widest scope consistent with the following claims and the principles and novel features disclosed herein.

What is claimed is:

1. A method of task-based handling of repetitive processes, comprising:
initializing, by a central controller, a set of descriptor table entries for maintaining in a descriptor table, wherein the descriptor table entries are for iteration space information of a repetitive process ready for execution;
initializing, by the central controller, a set of status table entries for maintaining in a status table, wherein the status table entries are for iteration space information of the repetitive process;
partitioning, by the central controller, iterations of the repetitive process into a first plurality of partitions;
initializing, by the central controller, a first task for executing iterations of a first partition;
- 22
initializing, by the central controller, a second task for executing iterations of a second partition;
assigning, by the central controller, the first task to execute by a first processor or processor core and the second task to execute by a second processor or processor core in parallel;
updating, by multiple local controllers each associated with one of the first and second processors or processor cores, multiple local tables to include partition execution status information for a task assigned to execute at the respective processor or processor core, wherein each table is associated with one of the first and second processors or processor cores;
executing the first task by the first processor or processor core and executing the second task by the second processor or processor core in parallel;
completing, by the second processor or processor core, execution of the second task;
updating, by the local controller associated with the second processor or processor core, the local table associated with the second processor or processor core to indicate that the second task has finished execution;
updating, by the central controller, the status table to indicate that the second task has finished completion and the second processor or processor core is available;
determining, by the central controller, whether the first partition is divisible; and
partitioning, by the central controller, the first partition of the first task into a second plurality of partitions in response to determining that the first partition is divisible;
2. The method of claim 1, further comprising:
assigning, by the central controller, a third partition of the second plurality of partitions to the second task for execution, and a fourth partition of the second plurality of partitions to the first task for execution;
updating, by the central controller, the status table with at least additional number of partitions, task assignment information, and task iteration status; and
updating, by multiple local controllers, each local controller associated with one of the first and second processors or processor cores, multiple local tables, each table associated with one of the first and second processors or processor cores, to include updated partition execution information for a task assigned to execute on the respective processor core;
3. The method of claim 1, wherein each of the partitions of the second plurality of partitions may be of any size;
4. The method of claim 1, wherein determining whether the first partition is divisible further comprises determining, by the central controller, whether the remaining iterations of partition assigned to the first task exceeds a minimum threshold;
5. The method of claim 1, further comprising partitioning, by the central controller of the processor, the iterations of the repetitive process by a number of partitions equivalent to a number of available processors or processor cores;
6. The method of claim 1, further comprising:
initializing, by the local controller associated with the first processor or processor core, a first pointer in the local table associated with the first processor or processor core for the first task;
updating, by the local controller of the first processor or processor core, the first pointer to indicate execution of the iterations of the repetitive process of the first partition; and

- updating, by the central controller, an execution status of the first partition in the status table;
7. The method of claim 1, further comprising:
- determining, by the local controller of the first processor or processor core, that an end iteration criterion for the first partition is invalid;
 - updating, by the local controller of the first processor or processor core, a local table associated with the first processor or processor core to reflect the new end iteration criterion for the first partition; and
 - updating, by the central controller, the status table with the new end iteration criterion for the first partition;
8. The method of claim 1, wherein updating any of the local tables further comprises requesting, by the central controller, partition information contained in a local table;
9. The method of claim 1, wherein updating any of the local tables further comprises passing, by the local controller of a respective processor or processor core, information updated in a local table to the status table at predetermined intervals or in response to an event;
10. The method of claim 1, wherein the status table or any of the local tables may be updated without interrupting execution of tasks by the first and second processors or processor cores;
11. The method of claim 1, further comprising:
- completing, by the first and second processors or processor cores, execution of all partitions of the repetitive process;
 - deleting, by the local controllers of the multiple processor cores, partition entries associated with the repetitive process from each of the local tables; and
 - deleting, by the central controller, iteration space information from the status table and the descriptor table;
12. A multi-technology communication device, comprising:
- multiple processors or processor cores configured with processor-executable instructions to perform operations comprising:
 - initializing in a descriptor table a set of descriptor table entries for iteration space information of a repetitive process ready for execution;
 - initializing in a status table a set of status table entries for iteration space information of the repetitive process;
 - partitioning iterations of the repetitive process into a first plurality of partitions;
 - initializing a first task for executing iterations of a first partition;
 - initializing a second task for executing iterations of a second partition;
 - assigning the first task to execute by a first processor or processor core and the second task to execute by a second processor or processor core in parallel;
 - updating multiple local tables to include partitioned execution status information for a task assigned to execute on the respective processor or processor core, wherein each table is associated with one of the first and second processors or processor cores;
 - executing the first task by the first processor or processor core and executing the second task by the second processor or processor core in parallel;
 - wherein at least one of the first and second processors or processor cores is further configured with processor-executable instructions to perform operations comprising:
13. The multi-technology communication device of claim 12, wherein at least one of the first and second processors or processor cores is further configured with processor-executable instructions to perform operations comprising:
- completing execution of the second task;
 - updating the local table associated with the second processor or processor core to indicate that the second task has finished execution;
 - updating the status table to indicate that the second task has finished completion and the second processor or processor core is available;
 - determining whether the first partition is divisible; and
 - partitioning the first partition of the first task into a second plurality of partitions in response to determining that the first partition is divisible;
14. The multi-technology communication device of claim 12, wherein at least one of the first and second processors or processor cores is further configured with processor-executable instructions to perform operations comprising:
- assigning a third partition of the second plurality of partitions to the second task for execution; and a fourth partition of the second plurality of partitions to the first task for execution;
 - updating the status table with at least additional number of partitions, task assignment information and task iteration status; and
 - updating each local controller associated with one of the first and second processors or processor cores, multiple local tables, each table associated with one of the first and second processors or processor cores, to include updated partition execution information for a task assigned to execute on the respective processor core;
15. The multi-technology communication device of claim 12, wherein each partition of the second plurality of partitions may be of any size;
16. The multi-technology communication device of claim 12, wherein at least one of the first and second processors or processor cores is further configured with processor-executable instructions to perform operations comprising:
- initializing a first pointer in the local table associated with the first processor or processor core for the first task;
 - updating the first pointer to indicate execution of the iterations of the repetitive process of the first partition; and
 - updating an execution status of the first partition in the status table;
17. The multi-technology communication device of claim 12, wherein at least one of the first and second processors or processor cores is further configured with processor-executable instructions to perform operations comprising:
- determining that an end iteration criterion for the first partition is invalid;
 - updating a local table associated with the first processor or processor core to reflect the new end iteration criterion for the first partition; and
 - updating the status table with the new end iteration criterion for the first partition;
18. A non-transitory processor-readable medium having stored thereon processor-executable software instructions to cause a processor of a multi-technology communication device to perform operations comprising:
- initializing a set of descriptor table entries for iteration space information of a repetitive process ready for execution;
 - initializing a set of status table entries for iteration space information of the repetitive process;

partitioning iterations of the repetitive process into a first plurality of partitions;
initializing a first task for executing iterations of a first partition;
initializing a second task for executing iterations of a second partition;
assigning the first task to execute by a first processor or processor core and the second task to execute by a second processor or processor core in parallel;
updating multiple local tables to include partition execution status information for a task assigned to execute on the respective processor or processor core, wherein each table is associated with one of the first and second processors or processor cores;
executing the first task by the first processor or processor core and executing the second task by the second processor or processor core in parallel;
completing, by the second processor or processor core, execution of the second task;
updating the local table associated with the second processor or processor core to indicate that the second task has finished execution;
updating the status table to indicate that the second task has finished completion and the second processor or processor core is available;

26

determining whether the first partition is divisible; and
partitioning the first partition of the first task into a second plurality of partitions in response to determining that the first partition is divisible.



US09710588B2

(12) **United States Patent**
Robatmili et al.(10) **Patent No.:** US 9,710,388 B2
(11) **Date of Patent:** Jul. 18, 2017(54) **HARDWARE ACCELERATION FOR INLINE CACHES IN DYNAMIC LANGUAGES**(71) **Applicant:** QUALCOMM Incorporated, San Diego, CA (US)(72) **Inventors:** Behnam Robatmili, San Jose, CA (US); Gheorghe Calin Cascaval, Palo Alto, CA (US); Moulakar Nagaraja Reddy, Santa Clara, CA (US); Darin Sanchez Gracia, Santa Clara, CA (US)(73) **Assignee:** QUALCOMM Incorporated, San Diego, CA (US)(1*) **Notice:** Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 412 days.(21) **Appl. No.:** 14/262,852(22) **Filed:** Apr. 28, 2014(65) **Prior Publication Data**

US 2015/0205726 A1 Jul. 23, 2015

Related U.S. Application Data

(60) Provisional application No. 61/930,808, filed on Jun. 23, 2014.

(51) **Int. Cl.**

G06F 12/08 (2016.01)

G06F 9/44 (2006.01)

(Continued)

(52) **U.S. Cl.**

CPC G06F 12/4875 (2015.01); G06F 9/3045 (2015.01); G06F 9/3067 (2015.01)

(Continued)

(58) **Field of Classification Search**

CPC G06F 12/0875; G06F 9/4431; G06F 12/0892; G06F 9/3045; G06F 9/3867;

G06F 2212/452

See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

7,153,664 B1 4,352,8 Chong et al. 2002/0087598 A1 * 7,395,2 Perinchery 6,004,177 3048 711,131 (Continued)

CITATION PUBLICATIONS

Car S., et al., "IMM Virtual Method Invoking Optimization Based on CMM Table," 16th IEEE International Conference on Networking, Architecture and Storage (NASC), Jul. 28, 2011 (Jul. 28, 2011) pp. 122-129, XPD-3938442, DOI: 10.1109/NASC.2011.13, ISBN: 978-1-4577-1172-5

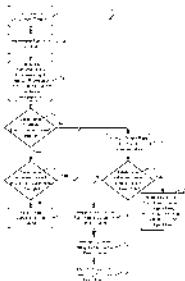
(Continued)

Primary Examiner: Michael Alsip

(74) **Attorney, Agent, or Firm:** The Marbury Law Group, PLLC(57) **ABSTRACT**

Aspects include a computing device, systems, and methods for hardware acceleration for inline caches in dynamic languages. An inline cache may be utilized for an instance of a dynamic software operation. A call of an initialized instance of the dynamic software operation may be executed by an inline cache hardware accelerator. The inline cache may be checked to determine that its data is current. When the data is current, the initialized instance of the dynamic software operation may be executed using the related inline cache data. When the data is not current, a new inline cache may be initialized for the instance of the dynamic software operation, including the not current data of a previously initialized instance of the dynamic software operation. The inline cache hardware accelerator may include an inline cache memory, a coprocessor, and/or a functional unit in an inline cache pipeline connected to a processor pipeline.

30 Claims, 8 Drawing Sheets



US 9,710,388 B2

Page 2

(51) Int. Cl.	G06F 9/00	(2006.01)
	G06F 9/18	(2006.01)
	G06F 12/0875	(2016.01)
	G06F 12/0892	(2016.01)
(52) U.S. Cl.		
CPC	G06F 9/4451 (2013.01); G06F 12/0892 (2013.01); G06F 2212/352 (2013.01)	
(56) References Cited		
		U.S. PATENT DOCUMENTS
200900610196 AL * 1.2009 Pearson		
20120231159 AL * 11.2012 Gao		G06F 9/11
		717-149
20130255286 AL 8.2013 Baracough et al		
20150067267 AL 3.2015 Pizio		
20150067658 AL * 3.2015 Balanenburg		G06F 12/0815
		717-149
20150255750 AL 7.2015 Revoirth et al		

OTTER PUBLICATIONS

- International Search Report and Written Opinion - PCT/US2015/012527; ISA/PPD; Mar. 23, 2015
Vigakrishnan, S., et al., "Object-Oriented Architectural Support for a Java Processor," In: Lecture Notes in Computer Science, Jan. 1 1998 (Jan. 1, 1998); Springer Berlin Heidelberg, Berlin, Heidelberg, NL0355176046, ISBN: 978-3-540-49523-8, Vol. 1445, pp. 336-354, DOI: 10.1007/BFb0025499
Advances in JavaScript Performance on Windows 8, 2012, Retrieved on May 7, 2014, Retrieved from the Internet : URL <http://blogs.msdn.com/b/javascript/2012/09/23/advances-in-javascript-performance-on-windows-8.aspx>, 17 Pages
Li, S., et al., "TypeCast: Demystifying Dynamic Typing of JavaScript Applications," High Performance Embedded Architectures and Compilers, 2011, 11 pages
Zakirov, S., "A study of superimpositions and dynamic mass optimizations," Feb. 2011, 101 Pages

* cited by examiner

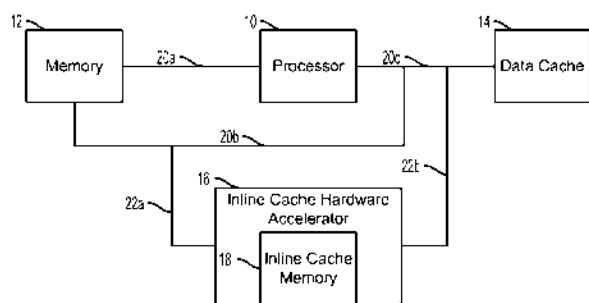


FIG. 1

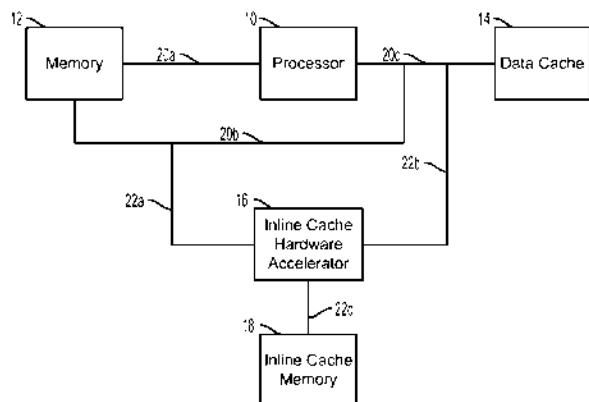


FIG. 2

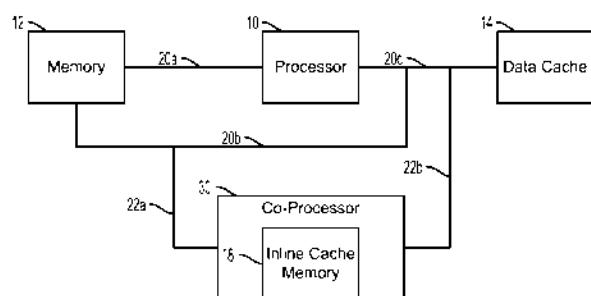


FIG. 3

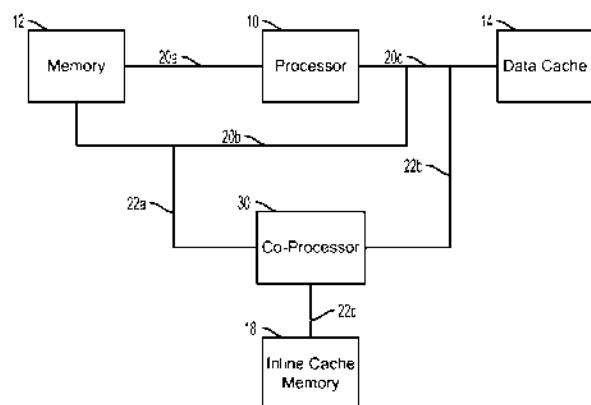


FIG. 4

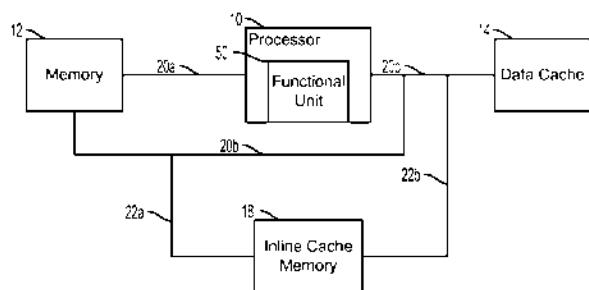


FIG. 5

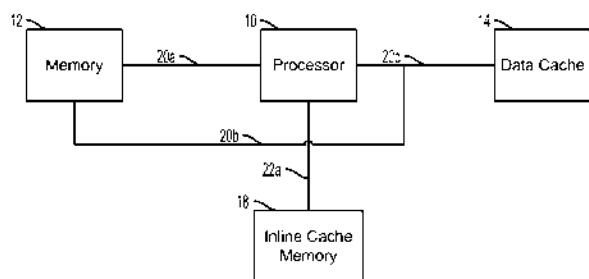


FIG. 6

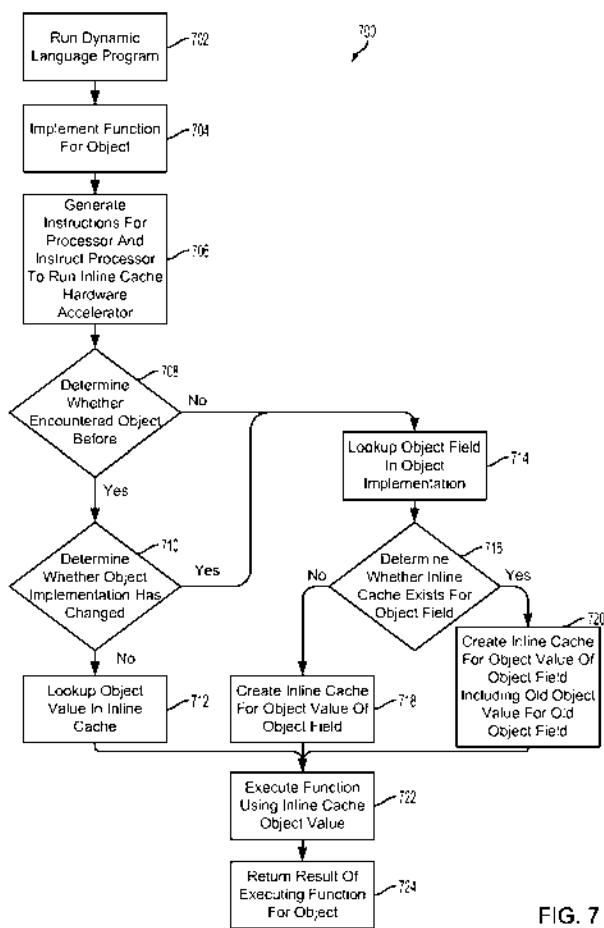


FIG. 7

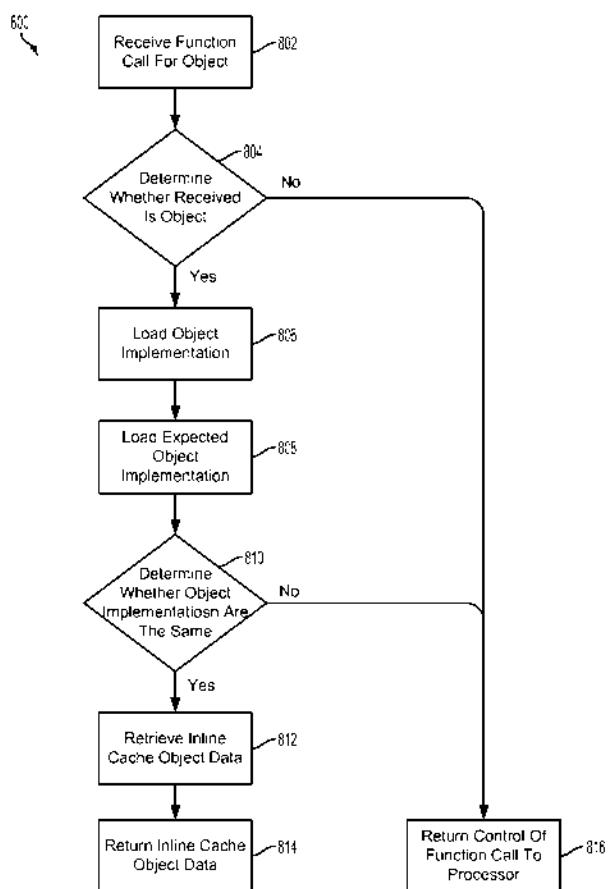


FIG. 8

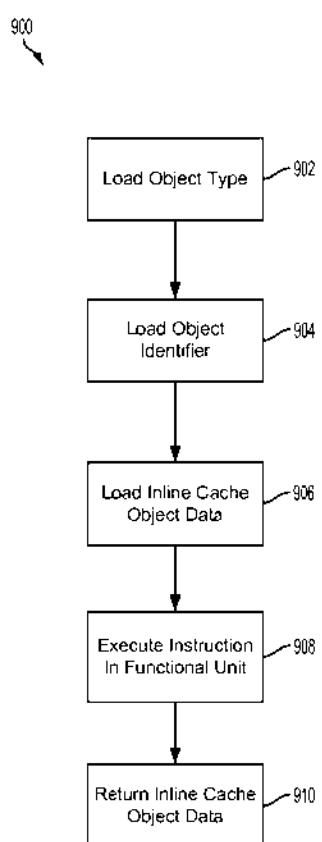


FIG. 9

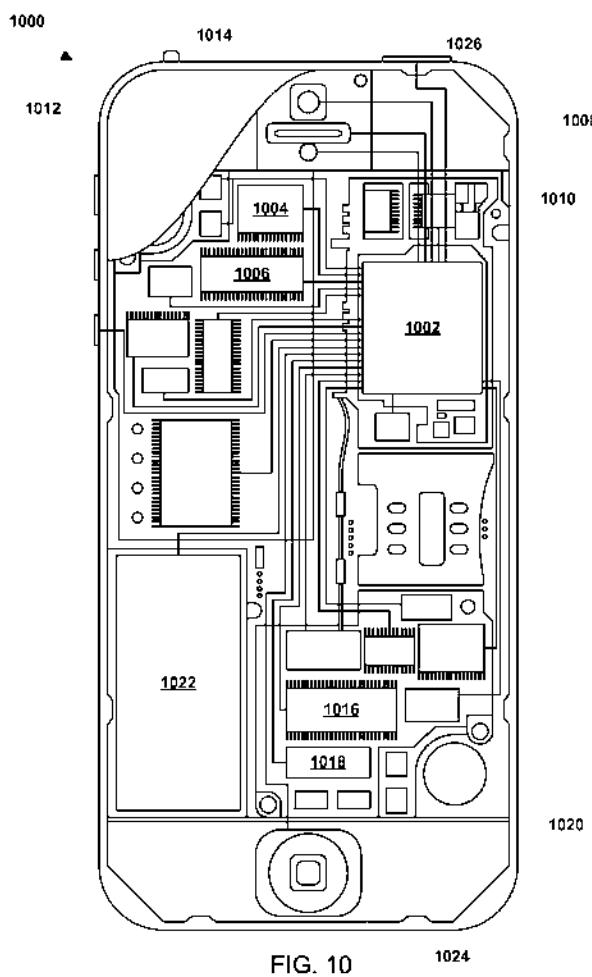


FIG. 10

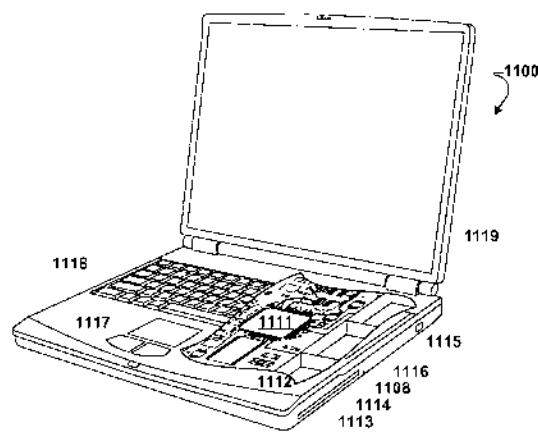


FIG. 11

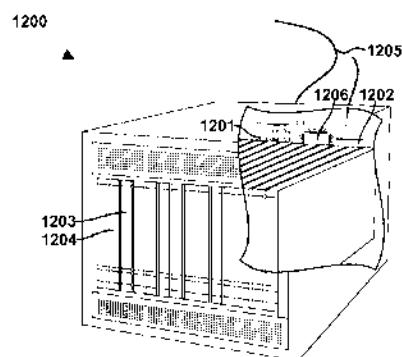


FIG. 12

1

2

HARDWARE ACCELERATION FOR INLINE CACHES IN DYNAMIC LANGUAGES

RELATED APPLICATIONS

This application claims the benefit of priority to U.S. Provisional Application No. 61,930,808 entitled "Hardware Acceleration For Inline Caches In Dynamic Languages" filed Jan. 23, 2014, the entire contents of which are hereby incorporated by reference.

BACKGROUND

Dynamic programming languages, such as JavaScript, Python and Ruby, are often used to execute common behaviors at runtime that other languages may execute while compiling the code. Dynamic programming languages increase the flexibility of a software program, often slowing down execution due to additional runtime compilation. Inline caches are a technique frequently used to reduce code execution overhead for dynamic languages by generating "fast paths" from common templates for the common behaviors. However, inline caches increase the memory usage of the program by storing additional inline cached code and constant values. In particular for mobile devices, memory is a constrained resource.

SUMMARY

The various aspects focus on methods and apparatuses for increasing the processing speed of dynamic language software on a computing device. Aspects methods may include initializing a first inline cache for a first instance of a dynamic software operation by a processor, storing the first inline cache in memory configured to provide fast access for storing and retrieving the first inline cache, receiving a second instance of the dynamic software operation in a coprocessor, determining whether the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same by the coprocessor, executing the second instance of the dynamic software operation by the coprocessor using the first inline cache from the memory in response to determining that the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same, and returning a result of executing the second instance of the dynamic software operation by the coprocessor to a processor.

In an aspect, determining whether the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same may include comparing a first object implementation related to the first instance of the dynamic software operation with a second object implementation related to the second instance of the dynamic software operation, and determining whether the first object implementation and the second object implementation are the same.

An aspect method may further include initializing a second inline cache for the second instance of the dynamic software operation, including the first inline cache configured to replace the initialized first inline cache in response to determining that the first instance of the dynamic software operation and the second instance of the dynamic software operation are different, storing the second inline cache in the memory configured to provide fast access for storing and retrieving the second inline cache, and executing the second instance of the dynamic software operation by the copro-

cessor using the second inline cache from the memory in response to determining that the first instance of the dynamic software operation and the second instance of the dynamic software operation are different.

An aspect method may further include determining whether the first inline cache exists for the first instance of the dynamic software operation, in which initializing the first inline cache for the first instance of the dynamic software operation by the processor may include initializing the first inline cache for the first instance of the dynamic software operation by the processor in response to determining that the first inline cache for the first instance of the dynamic software operation does not exist.

In an aspect, initializing the first inline cache for the first instance of the dynamic software operation by the processor may include traversing an object implementation for the dynamic software operation until identifying a data of the object implementation relating to the first instance of the dynamic software operation, executing the dynamic software operation of the first instance of the dynamic software operation, and returning a result of the first instance of the dynamic software operation.

In an aspect, returning the result of executing the second instance of the dynamic software operation by the coprocessor to the processor may include returning the result directly to the processor.

In an aspect, returning the result of executing the second instance of the dynamic software operation by the coprocessor may include returning the result to the processor indirectly through a data cache accessible to the processor and the coprocessor.

In an aspect, storing the first inline cache in the memory configured to provide fast access for storing and retrieving the first inline cache may include receiving the first inline cache from the processor disposed on a processor pipeline at the memory disposed on an inline cache pipeline connected to the processor pipeline, receiving the second instance of the dynamic software operation at the coprocessor may include receiving the second instance of the dynamic software operation from the processor disposed on the processor pipeline at the coprocessor disposed on the inline cache pipeline connected to the processor pipeline, and returning the result of executing the second instance of the dynamic software operation by the coprocessor may include sending the result of executing the second instance of the dynamic software operation from the coprocessor disposed on the inline cache pipeline to the processor disposed on the processor pipeline connected to the inline cache pipeline.

An aspect method may include generating executable operations for the coprocessor by a compiler, and instructing the processor to cause the coprocessor to execute the generated executable operations to perform operations which may include initializing the first inline cache for the first instance of the dynamic software operation by a processor, storing the first inline cache in the memory configured to provide fast access for storing and retrieving the first inline cache, receiving the second instance of the dynamic software operation in a coprocessor, determining whether the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same, executing the second instance of the dynamic software operation by the coprocessor using the first inline cache from the memory in response to determining that the first instance of the dynamic software operation and the second instance of the dynamic software operation are the

Some, and returning the result of executing the second instance of the dynamic software operation by the coprocessor.

An aspect includes a computing device having a processor, a memory, and a coprocessor communicatively connected to each other and the processor and coprocessor configured with processor-executable instructions to perform operations of one or more of the aspect methods described above.

An aspect includes a non-transitory processor-readable medium having stored thereon processor-executable software instructions to cause a processor and a coprocessor to perform operations of one or more of the aspect methods described above.

An aspect includes a computing device having means for performing functions of one or more of the aspect methods described above.

BRITISH DESCRIPTION OF THE DRAWINGS

The accompanying drawings, which are incorporated herein and constitute part of this specification, illustrate exemplary aspects of the invention, and together with the general description given above and the detailed description given below serve to explain the features of the invention.

FIG. 1 is a component block diagram illustrating a computing device having an inline cache hardware accelerator and an inline cache memory attached to a processor pipeline, for hardware acceleration for inline caches in dynamic languages, in accordance with an aspect.

FIG. 2 is a component block diagram illustrating a computing device having an inline cache hardware accelerator and an inline cache memory attached to a processor pipeline, for hardware acceleration for inline caches in dynamic languages, in accordance with another aspect.

FIG. 3 is a component block diagram illustrating a computing device having a coprocessor and an inline cache memory attached to a processor pipeline, for hardware acceleration for inline caches in dynamic languages, in accordance with another aspect.

FIG. 4 is a component block diagram illustrating a computing device having a coprocessor and an inline cache memory attached to a processor pipeline, for hardware acceleration for inline caches in dynamic languages, in accordance with another aspect.

FIG. 5 is a component block diagram illustrating a computing device having a functional unit and an inline cache memory attached to a processor pipeline, for hardware acceleration for inline caches in dynamic languages, in accordance with another aspect.

FIG. 6 is a component block diagram illustrating a computing device having an inline cache memory attached to a processor pipeline, for hardware acceleration for inline caches in dynamic languages, in accordance with an aspect.

FIG. 7 is a process flow diagram illustrating an aspect method for utilizing inline cache code and constant initialization using hardware acceleration for inline caches in dynamic languages.

FIG. 8 is a process flow diagram illustrating an aspect method for use of inline cache code and constants by a processor for inline caches in dynamic languages.

FIG. 9 is a process flow diagram illustrating an aspect method for use of inline cache code and constants by a functional unit for inline caches in dynamic languages.

FIG. 10 is a component block diagram illustrating an exemplary mobile computing device suitable for use with the various aspects.

FIG. 11 is a component block diagram illustrating an exemplary computing device suitable for use with the various aspects.

FIG. 12 is a component block diagram illustrating an exemplary server device suitable for use with the various aspects.

DETAILED DESCRIPTION

The various aspects will be described in detail with reference to the accompanying drawings. Whenever possible, the same reference numbers will be used throughout the drawings to refer to the same or like parts. References made to particular examples and implementations are for illustrative purposes, and are not intended to limit the scope of the invention or the claims.

The word "exemplary" is used herein to mean "serving as an example, instance, or illustration." Any implementation described herein as "exemplary" is not necessarily to be construed as preferred or advantageous over other implementations.

The term "computing device" is used interchangeably herein to refer to any one or all of cellular telephones, smartphones, personal or mobile multimedia players, personal data assistants (PDAs), laptop computers, tablet computers, phablets, notebooks, ultrabooks, palm-top computers, wireless electronic mail receivers, multimedia Internet enabled cellular telephones, wireless gaming controllers, desktop computers, servers, and similar personal or commercial electronic devices which include a memory, and a programmable processor.

The terms "system-on-chip" (SoC) and "integrated circuit" (IC) are used interchangeably herein to refer to a set of interconnected electronic circuits typically, but not exclusively, including one or more hardware cores, memory units, and communication interfaces. A hardware core may include a variety of different types of processors, such as a general purpose processor, a central processing unit (CPU), a digital signal processor (DSP), a graphics processing unit (GPU), an accelerated processing unit (APU), an auxiliary processor, a single-core processor, and a multi-core processor. A hardware core may further embody other hardware and hardware combinations, such as a field programmable gate array (FPGA), an application-specific integrated circuit (ASIC), other programmable logic device, discrete gate logic, transistor logic, performance monitoring hardware, logic, threshold logic, and time references. Integrated circuits may be configured such that the components of the integrated circuit reside on a single piece of semiconductor material, such as silicon. Such a configuration may also be referred to as the IC components being on a single chip.

Inline caching is a technique used to speedup dynamic compilation for languages such as JavaScript, PHP, Python, and Ruby. A compiler (e.g., a static compiler, a runtime compiler, or a dynamic compiler) may identify patterns of bytecode that exhibit common behaviors and may use code templates to generate executable code. The generated code may be parameterized with some object information and stored into an inline cache. The compiler may place guards to check whether the object matches the generated code, retrieve the code from the inline cache, and call it. The inline cache code may obviate the need for generating the same sequence repeatedly.

However, inline caching introduces a different set of performance issues. In a typical computing device (or computing system), inline cache items are still executed by the processor and stored in the memory. Also, many of the common tasks may generate an inline cache item for various

objects when the same inline cache item could provide the current result when using the proper parameters. Thus, inline caching techniques may clutter up the processor, memory and/or pipeline, and use up computing and power resources.

The various aspects include methods, devices, and non-transitory processor-readable storage media for increasing the processing speed of dynamic language software on a computing device. Hardware acceleration for inline caches in dynamic languages may utilize dedicated resources to manage the inline cache items for common tasks and take the burden off of the typical components of a computing device, such as an application processor, pipeline, and memory. In particular, an inline cache hardware accelerator may include at least some dedicated memory and some dedicated processing. The dedicated memory may be implemented as a separate memory, as a part of a data cache, or as part of a system memory. The dedicated processing may be implemented as extensions to an existing processor execution pipeline, as a processor functional unit, as a coprocessor communicating with a core processor through existing coprocessor interfaces, or as a separate and communicating with the processor through a proprietary interface. Hardware acceleration for inline caches in dynamic languages may be implemented along with the common inline caching, such that certain tasks are dedicated for hardware acceleration while others, likely the less common tasks, may continue to be managed by the typical components of the computing device.

The components for hardware acceleration for inline caches in dynamic languages may be separate but connected to the typical components of the computing device. For example, the computing device may contain a processor pipeline for transmitting signals between the various other components, such as the processor and a data cache. The components for hardware acceleration for inline caches in dynamic languages may be connected to each other and to the processor pipeline by an inline cache pipeline that may allow the components for hardware acceleration for inline caches in dynamic languages to transmit signals to each other and to the typical components of the computing device. This arrangement may allow the hardware acceleration for inline caches in dynamic languages to operate without impeding the operation of the typical components until communication with the typical components is necessary. The components for hardware acceleration for inline caches in dynamic languages may communicate with certain of the typical components either directly via the connected pipelines, or indirectly through the other typical components.

[FIG. 1] illustrates a computing device having an inline cache hardware accelerator and an inline cache memory attached to a processor pipeline, for hardware acceleration for inline caches in dynamic languages, in accordance with an aspect. A computing device may include a processor 10, registers or a memory 12, such as an accumulator or for storing stacks or register files, a data cache 14, an inline cache hardware accelerator 16, an inline cache memory 18, a processor pipeline 20a, 20b, 20c, and an inline cache pipeline 22a, 22b. The processor 10 may be one or more of a variety of different types of processors as described above. The processor 10 may be configured to execute coupled and/or interpreted executable operations from software programs. In an aspect, the software programs may be written in a dynamic programming language and the executable operations may be compiled or interpreted at runtime. The processor 10 may also be configured to execute sets of instructions to manage and interact with other components

of the computing device to execute portions of the dynamic programming language software programs.

The memory 12 may be configured to store state values for various states of the computing device. The state values stored in the memory 12 may be accessible for read and write operations by the processor 10. The data cache 14 may be configured to store data related to the executable operations executed by the processor 10. The data stored in the data cache 14 may be input to the processor 10 to execute the executable operations, or output by the processor 10 resulting from the executed operations and stored for later use or for access by other components of the computing device. The processor 10, the memory 12, and the data cache 14 may be connected to each other by the processor pipeline 20a, 20b, 20c. The processor pipeline 20a, 20b, 20c may be configured to transmit signals representing the executable operations, the computing device states, and the software program data between the components connected to it.

The inline cache hardware accelerator 16 which may be configured to increase the speed of processing executable operations from dynamic programming language software programs ("dynamic software operations"). The configurations of the inline cache hardware accelerator 16 are discussed in further detail below. The inline cache memory 18 may be configured to provide fast access for storing and retrieving data, and may be configured to store constant values associated with instances of the dynamic software operations executed by the processor 10, at least initially. The constant values may be associated with a particular instance of an object used to execute a dynamic software operation, such that the constant value may be recalled for future execution of the same instance of the object for the dynamic software operation. In an aspect, the dynamic software operation may include one operation or a set of operations, and therefore references to singular operation and plural operations are not intended to limit the scope of the claims in terms of the number of executable operations unless explicitly recited in a claim.

The inline cache memory 18 may be dedicated for this specific purpose, and therefore may be relatively small and fast memory because the storing of constants may not require a lot of space; and providing quick access to the inline cache memory 18 for read and write operations may facilitate increasing the processing speed of the dynamic software operations. Access to the inline cache memory 18 may be limited to the inline cache hardware accelerator 16. In an aspect, the inline cache memory 18 may be included as part of the inline cache hardware accelerator 16. An inline cache pipeline 22a, 22b may connect the inline cache hardware accelerator 16 and the inline cache memory 18 to the processor 10, the memory 12, and the data cache 14. The inline cache pipeline 22a, 22b may connect directly to the other components of the computing device, or may connect to the processor pipeline 20a, 20b, 20c. The inline cache pipeline 22a, 22b may be configured to transmit signals representing data used to execute dynamic software operations, the computing device states, and/or data resulting from the execution of the dynamic software, including the constant values stored on the inline cache memory 18.

[FIG. 2] illustrates a computing device having an inline cache hardware accelerator and an inline cache memory attached to a processor pipeline, for hardware acceleration for inline caches in dynamic languages, in accordance with an aspect. Similar to the computing device illustrated in FIG. 1, the computing device may include the processor 10, the registers or the memory 12, the data cache 14, the inline cache hardware accelerator 16, the inline cache memory 18,

the processor pipeline 20a, 20b, 20c, and the inline cache pipeline 22a, 22b. In an aspect the inline cache memory 18 may be separate from the inline cache hardware accelerator 16. In various aspects, the inline cache memory 18 may be a standalone memory, memory integrated into the cache hierarchy, or a portion of the system memory of the computing device. The computing device may also include int inline cache pipeline 22c, further configured to connect the inline cache hardware accelerator 16 and the inline cache memory 18. The inline cache memory 18 may be accessible by the inline cache hardware accelerator 16, and by the other components, such as the processor 10, through the inline cache hardware accelerator 16.

FIG. 3 illustrates a computing device having a coprocessor 30 and an inline cache memory attached to a processor pipeline, for hardware acceleration for inline caches in dynamic languages, in accordance with an aspect. Similar to the computing device illustrated in FIG. 1, the computing device may include the processor 10, the registers or the memory 12, the data cache 14, the inline cache memory 18, the processor pipeline 20a, 20b, 20c, and the inline cache pipeline 22a, 22b. In an aspect the inline cache hardware accelerator may be a coprocessor 30 configured to execute the dynamic software operations. The coprocessor 30, like the processor 10, may be one or more of a variety of different types of processors as described above. In an aspect, the coprocessor 30 may be a programmable logic device programmed to execute one or more dynamic software operations. In an aspect, the programmable coprocessor 30 may be reprogrammable to execute a different dynamic software operation. In another aspect, the computing device may include multiple coprocessors 30, each configured to execute a specific dynamic software operation. The dynamic software operations may be typical executable operations and the use of more coprocessors 30 may be provided for inclusion in the computing device on the basis of the typical dynamic software operations for the computing device. In an aspect, the inline cache memory 18 may be included as part of and/or dedicated to the coprocessor 30. In another aspect, the inline cache memory 18 may be one or more inline cache memories 18 shared across multiple coprocessors 30. In an aspect, the coprocessor 30 may be implemented as a functional extension of the processor pipeline 20a, 20b, 20c, and configured with instructions to initialize the inline cache memory 18, to execute the code stored in the inline cache memory 18, and to query the inline cache memory 18 for the presence of code for a first instance of a dynamic software operation. In an aspect, the coprocessor 30 may be implemented as a standalone unit communicating with the processor 10 using standard coprocessor interfaces. Such communications may include transferring initial values, initiating computation, and transferring output results. In another aspect, the coprocessor 30 may be implemented as a standalone unit communicating with the processor 10 using an enhanced interface, allowing the processor 10 to provide more information to the coprocessor 30 (e.g., an object map).

As described in further detail below, the processor 10 may pass data to the coprocessor 30 relating to an instance of a dynamic software operation. The coprocessor 30 may determine whether there are any inline cache data relating to the instance of the dynamic software operation. When the coprocessor 30 determines that the instance of the dynamic software operation is not yet initialized (i.e., the dynamic software operation is “uninitialized”) or that it is the first instance of the dynamic software operation, the coprocessor 30 may signal the processor 10 indicating that it cannot

process the instance of the dynamic software operation. The coprocessor 30 may determine that the instance of the dynamic software operation is uninitialized by comparing data stored in the inline cache memory 18 with the data received from the processor 10. When there is no data in the inline cache memory 18 for the instance of the dynamic software operation, the coprocessor 30 may determine that the instance is uninitialized. The processor 10 may initialize the instance of the dynamic software operation and store the results of the instance as a constant value to the inline cache memory 18.

When there is data in the inline cache memory 18 for the instance of the dynamic software operation, the coprocessor 30 may determine that the instance is initialized. The coprocessor 30 may then determine whether the data stored in the inline cache memory 18 for the instance of the dynamic software operation is the correct data for the instance. The coprocessor 30 may determine that the stored data is not the correct data for the instance of the dynamic software operation by comparing the data stored in the inline cache memory 18 with the data received from the processor 10. When the data stored for the instance of the dynamic software operation does not match the data provided by the processor 10, the coprocessor 30 may determine that the data stored for the instance of the dynamic software operation is not the correct data, and as a result may signal the processor 10 indicating that the coprocessor 30 cannot process the instance of the dynamic software operation. The processor 10 may then initialize this instance of the dynamic software operation and store the results of the instance as another constant value in the inline cache memory 18.

Stored data for instances of a dynamic software operation may not match data received from the processor 10 due to a change in an implementation for the objects, such as object maps, for the dynamic software operations. Such a change may associate instances of the objects with different values causing different results for functions for the instances of the objects, or instances of the dynamic software operations. Similar to the uninitialized instance above, an instance of a dynamic software operation may be uninitialized due to such a change.

When there is data in the inline cache memory 18 for the instance of the dynamic software operation and the data is correct data, the coprocessor 30 may return the constant value for the instance of the dynamic software operation to the processor 10. In an aspect, the above initialization of the instance for the dynamic software operation may include the processor 10 running the instance for the dynamic software operation as normal to implement its result. Some dynamic software operations are common and various instances of the dynamic software operations are called repeatedly. In many cases, the same instance of the dynamic software operation is called repeatedly. The coprocessor 30 may be configured to increase the speed of the dynamic software operations by providing precomputed or precalculated results to the initialized instances of the dynamic software operation stored in a memory dedicated to storing the results (i.e., the inline cache memory 18).

FIG. 4 illustrates a computing device having a coprocessor 30 and an inline cache memory attached to a processor pipeline, for hardware acceleration for inline caches in dynamic languages, in accordance with an aspect. Similar to the computing device illustrated in FIG. 3, the computing device may include the processor 10, the registers or the memory 12, the data cache 14, the inline cache memory 18, the processor pipeline 20a, 20b, 20c, the inline cache pipeline 22a, 22b, and the coprocessor 30, in an aspect the

inline cache memory **18** may be separate from the coprocessor **30**. The computing device may also include the inline cache pipeline **22c**, further configured to connect the coprocessor **30** and the inline cache memory **18**, like in FIG. 2. The inline cache memory **18** may be accessible by the coprocessor **30**, and by the other components, such as the processor **10**, through the coprocessor **30**. In an aspect, the inline cache memory **18** may be dedicated to the coprocessor **30**. In another aspect, the inline cache memory **18** may be one or more inline cache memories **18** shared across multiple coprocessors **30**. Regardless of the difference in structures of the computing device illustrated in FIGS. 4 and the computing device illustrated in FIG. 3, the operation of the computing devices and their components are substantially similar.

In aspects, including the coprocessor **30**, processors **10** may be configured to communicate with the coprocessor **30** and may not require alterations to the processors' instruction set architecture. In an aspect, data associated with the instances of the dynamic software operations may be explicitly passed from the processor **10** to the coprocessor **30**. In another aspect, the coprocessor **30** may be able to communicate directly with the memory **12** and the data cache **14**. This direct communication may allow for the coprocessor **30** to implicitly receive data associated with the instances of the dynamic software operations from the processor **10** by way of the memory **12** and the data cache **14**. Function calls to the coprocessor **30** by the processor **10** may include blocking type calls, where only one call may be handled at a time. Function calls to the coprocessor **30** by the processor **10** may include non-blocking or asynchronous type calls, which may allow for multiple calls concurrently in the inline cache pipeline **22a**, **22b**, **22c**. Similarly, the function calls to the coprocessor **30** by the processor **10** may parallel type calls, which may allow for multiple calls simultaneously in the inline cache pipeline **22a**, **22b**, **22c**.

FIG. 5 illustrates computing device having a functional unit and an inline cache memory attached to a processor pipeline, for hardware acceleration for inline caches at dynamic languages, in accordance with an aspect. Similar to FIG. 2, the computing device may include the processor **10**, the registers or the memory **12**, the data cache **14**, the inline cache memory **18**, the processor pipeline **20a**, **20b**, **20c**, and the inline cache pipeline **22a**, **22b**. In an aspect the inline cache hardware accelerator may be a functional unit **50** configured to execute the dynamic software operations. The functional unit **50** may be a circuit configured to perform a specific function or calculation. For example, the functional unit **50** may be an adder or a multiplier. One or more functional units **50** may be combined to achieve the specific function or calculation. Multiple functional units **50** or groups of functional units **50** may be configured to implement a variety of different dynamic software operations. The functional unit **50** may be an integrated component of the processor **10**. In an aspect the inline cache memory **18** may be separate from the functional unit **50**. The computing device may also include an inline cache pipeline, further configured to connect the functional unit **50** and the inline cache memory **18**. The inline cache memory **18** may be accessible by the functional unit **50** and the other components, such as the processor **10**, through the functional unit **50**. In an aspect, the inline cache memory **18** may be dedicated to functional unit **50**. In another aspect, the inline cache memory **18** may be one or more inline cache memories **18** shared across multiple functional units **50**.

As described in further detail below, the processor **10** may determine whether the instance of the dynamic software

operation is initialized or uninitialized similar to how the coprocessor makes the determination in the description above. When the instance of the dynamic software operation is uninitialized, the processor **10** may execute the instance of the dynamic software operation to determine the result. The processor **10** may store the result of the executed instance of the dynamic software operation on the inline cache memory **18**, such that the instance of the dynamic software operation may be initialized. When the processor **10** determines that the instance of the dynamic software operation is initialized, the processor **10** may pass data to one or more selected functional units **50** relating to the initialized instance of the dynamic software operation. The functional unit **50** selected by the processor **10** may be configured specifically to execute the dynamic software operation received by the processor **10**. The processor **10** may match the dynamic software operation with the appropriate functional unit **50**, and pass the appropriate data to the selected functional unit **50**. The functional unit **50** may operate using the data from the processor **10** to determine the inline cache data relating to the instance of the dynamic software operation stored on the inline cache memory **18**. The functional unit **50** may use the data from the processor **10** and execute the dynamic software operation for which it is configured to implement. The result of the initialized instance of the dynamic software operation may be the constant value stored on the inline cache memory **18**, and may be passed to the processor **10** to complete the execution of the instance of the dynamic software operation.

In aspects including the functional unit **50**, processors **10** may or may not be configured to communicate with the functional unit **50** and may or may not require alterations to the processors' instruction set architecture. In an aspect, data associated with the instances of the dynamic software operations may be explicitly passed from the processor **10** to the functional unit **50**. In another aspect, the functional unit **50** may be able to communicate directly with the memory **12** and the data cache **14**. This direct communication may allow for the functional unit **50** to implicitly receive data associated with the instances of the dynamic software operations from the processor **10** by way of the memory **12** and the data cache **14**. Function calls to the functional unit **50** by the processor **10** may include blocking type calls, where only one call may be handled at a time. Function calls to the functional unit **50** by the processor **10** may include non-blocking or asynchronous type calls, which may allow for multiple calls concurrently in the inline cache pipeline **22a**, **22b**. Similarly, the function calls to the functional unit **50** by the processor **10** may parallel type calls, which may allow for multiple calls simultaneously in the inline cache pipeline **22a**, **22b**.

FIG. 6 illustrates computing device having an inline cache memory attached to a processor pipeline, for hardware acceleration for inline caches in dynamic languages, in accordance with an aspect. Similar to the computing device illustrated in FIG. 1, the computing device may include a processor **10**, registers or the memory **12**, a data cache **14**, an inline cache memory **18**, a processor pipeline **20a**, **20b**, **20c**, and an inline cache pipeline **22a**. In an aspect, the inline cache hardware accelerator may be the inline cache memory **18** configured to store results of the instances of the dynamic software operations. In an aspect, the inline cache memory **18** may be one or more inline cache memories **18** shared for multiple instances of the dynamic software operations, one or more inline cache memories **18** each dedicated to storing multiple instances of the same dynamic software operation.

or one inline cache memory 18 dedicated to storing all of the instances of all of the dynamic software operations.

As described in further detail below, the processor 10 may determine whether the instance of the dynamic software operation is initialized or uninitialized similar to how the processor 10 makes the determination that is described above. When the instance of the dynamic software operation is uninitialized, the processor 10 may execute the instance of the dynamic software operation to determine the result. The processor 10 may store the result of the executed instance of the dynamic software operation on the inline cache memory 18, such that the instance of the dynamic software operation may be initialized. When the processor 10 determines that the instance of the dynamic software operation is initialized, the processor 10 may retrieve the data stored on the inline cache memory 18 for the initialized instance of the dynamic software operation without having to fully execute the operation as it would if the operation were uninitialized.

In each of the foregoing aspect computing devices, the components, including the processor 10, the registers or file memory 12, the data cache 14, the inline cache hardware accelerator 16, the inline cache memory 18, the processor pipeline 20a, 20b, 20c, the inline cache pipeline 22a, 22b, 22, and the coprocessor 30, and the functional unit 50 may be configured in various combinations. Some or all of the components may comprise individual or combined components. Similarly, some or all of the components may be included as part of an SoC or one or more integrated circuits.

FIG. 7 illustrates an aspect method 700 for inline cache code and context initializing using hardware acceleration for inline caches in dynamic languages. The computing device and its components, including the processor, the registers or the memory for register files, the data cache, the inline cache hardware accelerator, the inline cache memory, the processor pipeline, the inline cache pipeline, the coprocessor, and the functional unit, may execute the method 700. In block 702, the computing device may run the dynamic language executable program. In an aspect, the dynamic language executable program may be run in a web browser, a web application, or a stand-alone application. In block 704, the computing device may implement a function for an object, or the instance of the dynamic software operation. Common dynamic software operations, such as loading data, storing data, calling certain functions, and performing binary operations on data may be the types of dynamic software operations that may be included for hardware acceleration. In block 706, a compiler (e.g., a static compiler, a runtime compiler, or a dynamic compiler) may generate executable instructions for the processor to instruct the processor or a coprocessor to interact with and run the inline cache hardware accelerator for the instance of the dynamic software operation. These instructions may cause the processor or a coprocessor to read from and write to the inline cache memory either directly or through the inline cache hardware accelerator and cause the inline cache hardware accelerator to function as described below.

In determination block 708, the computing device may determine whether the computing device has previously run the function for the object, or the instance of the dynamic software operation. In an aspect, the computing device may accomplish this by checking the inline cache memory for stored data related to the instance of the dynamic software operation. The determined presence of data related to the instance of the dynamic software operation on the inline cache memory may signify to the computing device that the instance of the dynamic software operation may be initialized. However the determination that the dynamic software

operation is initialized may not be certain without a further determination discussed below. For example, data related to the instance of the dynamic software operation may be stored and thus may signify that a prior instance of the dynamic software operation was initialized, but that the data may not be current.

When the computing device determines that the instance of the dynamic software operation was previously run (i.e., determination block 708 “Yes”), the computing device may determine whether the object implementation for the instance of the dynamic software operation has changed in determination block 710. Determining whether the object implementation of the instance of the dynamic software operation has changed may indicate to the computing device whether or not the data stored on the inline cache memory for the instance of the dynamic software operation is current data. When the computing device determines that the object implementation for the instance of the dynamic software operation has not changed (i.e., determination block 710 “No”), the computing device may look up the value for the instance of the dynamic software operation stored on the inline cache memory in block 712. Determining that data exists in the inline cache memory for the instance of the dynamic software operation and that the data is current together may signify to the computing device that the instance of the dynamic software operation is initialized. In block 722, the computing device may execute any operations on the data from the inline cache memory requested by the processor as part of the dynamic software operation. In block 724, the computing device may return the data related to the instance of the dynamic software operation to the processor. The data may be returned to the processor either directly or indirectly through the memory or register files and/or the data cache.

When the computing device determines that the instance of the dynamic software operation was not previously run (i.e., determination block 708 “No”) or when the computing device determines that the object implementation for the instance of the dynamic software operation has changed (i.e., determination block 710 “Yes”), the computing device may lookup related fields for the operation in an object implementation for the operation in block 714. It may be common for the processor to traverse the object implementation item by item until it finds the appropriate field associated with an object value for the operation. This process can be time and resource consuming, and inline caching of the appropriate value helps to avoid repetition of this step for future instances of the dynamic software operation. Hardware acceleration furthers this effort by taking some of the processing involved in hardware caching from the processor and allocating it to the inline hardware accelerator, or by simplifying the storage and retrieval of the inline cached data.

In determination block 716, the computing device may determine whether an inline cache data exists for an instance of the dynamic software operation. This may be possible because a previous instance of the dynamic software operation may have been initialized but the current data for the instance of the dynamic operation may not match that of the previously initialized instance. When the computing device determines that no inline cache data exists for the instance dynamic software operation (i.e., determination block 716 “No”), the computing device may create, or initialize, an inline cache data for the instance of the dynamic software operation in block 718. The computing device may continue to block 722 to execute any operations on the data requested by the processor as part of the instance of the dynamic

software operation, and may perform the operations of block 724 to return the result of the instance.

When the computing device determines that an inline cache data exists for the instance dynamic software operation (i.e., determination block 716 “Yes”), the computing device may create or initialize an inline cache data for the instance of the dynamic software operation including the inline cache data for the previous instance in block 720. For example, the computing device may initialize a second inline cache including a first inline cache configured to replace the initialized first inline cache when a first instance of a dynamic software operation and a second instance of the dynamic software operation are different. The computing device may continue to block 722 to execute any operations on the data requested by the processor as part of the instance of the dynamic software operation, and may perform the operations of block 724 to return the result of the instance.

In various aspects, the inline cache may have multiple allocation and replacement policies. For example, in blocks 718 and 720, the computing device may not initialize the inline cache data for every instance of the dynamic software operation. For example, an inline cache data may be initialized only after a second execution of a dynamic software operation. Other specified criteria may determine when an inline cache data may be initialized for a dynamic software operation, such as frequency of execution according to various metrics, complexity and/or cost of execution in terms of time and/or resources, or operation by a running program or application. In an aspect, initialized inline cache data may be removed, or uninitialized, for various reasons. For example, an inline cache data may be explicitly evicted with an operation from the inline cache. By conventional policies such as least recently used, least frequently used, first-in-first-out, last-in-first-out, and age, other criteria, such as least complex and/or costliest, most inline cache memory usage, may also be factors for removing inline cache data. In some aspects, the performance or space availability of the inline cache memory may determine when an inline cache data may be considered for removal. Some criteria for initializing or uninitialized may take into account current and/or historical data in determining whether to initialize or uninitialized inline cache data for a dynamic software operation. Removed inline cache data may be initialized again according to any of the described criteria; it may be required to meet other, potentially more stringent criteria than inline cache data yet to be initialized, or may be prevented from initializing for at least a certain period of until a certain event occurs. For example, removed inline cache data may be prevented from initializing again until a next running of a program, or until the inline caching is reset, like upon a reboot of the computing device.

FIG. 8 illustrates an aspect method 800 for use of inline cache code and constants by a processor for inline caches at dynamic languages. The computing device and its components, including the processor, the registers or the memory for register files, the data cache, the inline cache hardware accelerator, the inline cache memory, the processor pipeline, the inline cache pipeline, the coprocessor, and the functional unit, may execute the method 800. In block 802, the computing device may receive a function call for an instance of an object, or an instance of the dynamic software operation. In determination block 804, the computing device may determine whether the received data is an object, rather than a small integer, for example. When the computing device determines that the received data is not an object (i.e.,

determination block 804 “No”), the computing device may return control of the function call to the processor in block 816.

When the computing device determines that the received data is an object (i.e., determination block 804 “Yes”), the computing device may load the current object implementation for the instance of the dynamic software operation at block 816. In block 808, the computing device may load the expected object implementation for the instance of the dynamic software operation. The expected object implementation may be the object implementation used to initialize a previous instance of the dynamic software operation. In determination block 810, the computing device may determine whether the current object implementation and the expected object implementation are the same object implementation. This determination may be made by comparing an encoding of the current object with an encoding of the object stored with the inline cache at a time when the code in the inline cache was generated. This encoding may be an address of the object prototype, or an encoding of the object fields, or any other mechanism that can uniquely identify the structure of the object. It may be possible for an object implementation to change after the initialization of an instance of the dynamic software operation, and the result may be that the data from the object implementation for a current instance of the dynamic software operation may no longer match the data from the previous instance. Reusing the wrong data based on the previous instance of the dynamic software operation may cause errors in the execution of the dynamic software. When the computing device determines that the current object implementation and the expected object implementation are different (i.e., determination block 810 “No”), the computing device may return control of the function call to the processor in block 816.

When the computing device determines that the current object implementation and the expected object implementation are the same (i.e., determination block 810 “Yes”), the computing device may retrieve the inline cache data in block 812. In block 814, the computing device may return the data related to the instance of the dynamic software operation to the processor. The data may be returned to the processor either directly or indirectly through the memory or register files and/or the data cache.

FIG. 9 illustrates an aspect method 900 for use of inline cache code and constants by a functional unit for inline caches in dynamic languages. The computing device and its components, including the processor, the registers or the memory for register files, the data cache, the inline cache hardware accelerator, the inline cache memory, the processor pipeline, the inline cache pipeline, the coprocessor, and the functional unit, may execute the method 900. In block 902, the computing device may load an object type. The object type may indicate to the computing device what dynamic software operation is being run. In block 904, the computing device may load an object identifier. The object identifier may indicate to the computing device a specific instance of the dynamic software operation is being run, including an input value, for example. Using the information from the instance, in block 906, the computing device may load the inline cache data for the instance of the dynamic software operation. In block 908, the computing device may execute the dynamic software operation on the data for the instance. The computing device may use the object type, the object identifier, and the inline cache data as parameters to execute the dynamic software operation. In block 910, the computing device may return the data related to the instance of the dynamic software operation to the processor. The data

may be returned to the processor either directly or indirectly through the memory for register files and/or the data cache 116. It illustrates an exemplary mobile computing device 1000 suitable for use with the various aspects. The mobile computing device 1000 may include a processor 1002 coupled to a touchscreen controller 1004 and an internal memory 1006. The processor 1002 may be one or more multi-core integrated circuits designed for general or specific processing tasks. The internal memory 1006 may be volatile or non-volatile memory, and may also be secure and/or encrypted memory, or insecure and/or unencrypted memory, or any combination thereof. The touchscreen controller 1004 and the processor 1002 may also be coupled to a touchscreen panel 1012, such as a resistive-sensing touchscreen, capacitive-sensing touchscreen, infrared-sensing touchscreen, etc. Additionally, the display of the mobile computing device 1000 need not have touch screen capability.

The mobile computing device 1000 may have one or more radio signal transceivers 1008 (e.g., Peanut, Bluetooth, Zigbee, WiFi, BT, radio and antenna 1010) for sending and receiving communications, coupled to each other and/or to the processor 1002. The transceivers 1008 and antenna 1010 may be used with the above-mentioned circuitry to implement the various wireless transmission protocol stacks and interfaces. The mobile computing device 1000 may include a cellular network wireless modem chip 1016 that enables communication via a cellular network and is coupled to the processor.

The mobile computing device 1000 may include a peripheral device connection interface 1018 coupled to the processor 1002. The peripheral device connection interface 1018 may be singularly configured to accept one type of connection, or may be configured to accept various types of physical and communication connections, common or proprietary, such as USB, FireWire, Thunderbolt, or PCIe. The peripheral device connection interface 1018 may also be coupled to a similarly configured peripheral device connection port (not shown).

The mobile computing device 1000 may also include speakers 1014 for providing audio outputs. The mobile computing device 1000 may also include a housing 1020, constructed of a plastic, metal, or a combination of materials, for containing all or some of the components discussed herein. The mobile computing device 1000 may include a power source 1022 coupled to the processor 1002, such as a disposable or rechargeable battery. The rechargeable battery may also be coupled to the peripheral device connection port to receive a charging current from a source external to the mobile device 1000. The mobile computing device 1000 may also include a physical button 1024 for receiving user inputs. The mobile computing device 1000 may also include a power button 1026 for turning the mobile device 1000 on and off.

The various aspects described above may also be implemented within a variety of other types of computing devices, such as a laptop computing device 1100 illustrated in FIG. 11. Many laptop computers include a touchpad/touch surface 1117 that serves as the computer's pointing device, and thus may receive drag, scroll, and flick gestures similar to those implemented on computing devices equipped with a touch screen display, as described above. A laptop computing device 1100 will typically include a processor 1111 coupled to volatile memory 1112 and a large capacity nonvolatile memory, such as a disk drive 1113. This memory, Additionally, the computing device 1100 may have one or more antennas 1108 for sending and receiving electromagnetic radiation that may be connected to a wireless data link

and/or cellular telephone transceiver 1116 coupled to the processor 1111. The computing device 1100 may also include a floppy disc drive 1114 and a compact disc (CD) drive 1115 coupled to the processor 1111. In a notebook configuration, the computing device housing includes the touchpad 1117, the keyboard 1118, and the display 1119 all coupled to the processor 1111. Other configurations of the computing device may include a computer mouse or trackball coupled to the processor (e.g., via a USB input) as are well known, which may also be use in conjunction with the various aspects. A desktop computer may similarly include these computing device components in various configurations, including separating and combining the components in one or more separate but connectable parts.

The various embodiments may also be implemented in any of a variety of commercially available server devices, such as the server 1200 illustrated in FIG. 12. Such a server 1200 typically includes a processor 1201 coupled to volatile memory 1202 and a large capacity nonvolatile memory, such as a disk drive 1204. The server 1200 may also include a floppy disc drive, compact disc (CD), or DVD disc drive 1206 coupled to the processor 1201. The server 1200 may also include network access ports 1203 coupled to the processor 1201 for establishing network interface connections with a network 1205, such as a local area network coupled to other broadcast system computers and servers, the Internet, the public switched telephone network, and/or a cellular data network (e.g., CDMA, TDMA, GSM, PCS, 3G, 4G, LTE, or any other type of cellular data network).

The foregoing method descriptions and the process flow diagrams are provided merely as illustrative examples, and are not intended to require or imply that the operations of the various aspects must be performed in the order presented. As will be appreciated by one of skill in the art the order of operations in the foregoing aspects may be performed in any order. Words such as "hereafter," "then," "next," etc. are not intended to limit the order of the operations; these words are simply used to guide the reader through the description of the methods. Further, any reference to plural elements in the singular, for example, using the articles "a," "an" or "the," is not to be construed as limiting the element to the singular.

The various illustrative logical blocks, modules, circuits, and algorithm operations described in connection with the various aspects may be implemented as electronic hardware, computer software, or combinations of both. To clearly illustrate this interchangeability of hardware and software, various illustrative components, blocks, modules, circuits, and operations have been described above generally in terms of their functionality. Whether such functionality is implemented as hardware or software depends upon the particular application and design constraints imposed on the overall system. Skilled artisans may implement the described functionality in varying ways for each particular application, but such implementation decisions should not be interpreted as causing a departure from the scope of the present invention.

The hardware used to implement the various alternative logics, logical blocks, modules, and circuits described in connection with the aspects disclosed herein may be implemented or performed with a general purpose processor, a digital signal processor (DSP), an application specific integrated circuit (ASIC), a field programmable gate array (FPGA), or other programmable logic device, discrete gate or transistor logic, discrete hardware components, or any combination thereof designed to perform the functions described herein. A general-purpose processor may be a microprocessor, but, in the alternative, the processor may be any conventional processor, controller, microcontroller, or state

machine. A processor may also be implemented as a combination of computing devices, e.g., a combination of a DSP and a microprocessor; a plurality of microprocessors, one or more microprocessors in conjunction with a DSP core, or any other such configuration. Alternatively, some operations or methods may be performed by circuitry that is specific to a given function.

In one or more aspects, the functions described may be implemented in hardware, software, firmware, or any combination thereof. If implemented in software, the functions may be stored in one or more instructions, operations, or code on a non-transitory computer-readable medium or a non-transitory processor-readable medium. The operations of a method or algorithm disclosed herein may be embodied in a processor-executable software module comprising processor-executable instructions or operations that may reside on a non-transitory computer-readable or processor-readable storage medium. Non-transitory computer-readable or processor-readable storage media may be any storage media that may be accessed by a computer or a processor. By way of example but not limitation, such non-transitory computer-readable or processor-readable media may include RAM, ROM, EPROM, FLASH memory, CD-RAM or other optical disk storage, magnetic disk storage or other magnetic storage devices, or any other medium that may be used to store desired program code in the form of instructions, operations, or data structures and that may be accessed by a computer. Disk and disc, as used herein, includes compact disc (CD), laser disc, optical disc, digital versatile disc (DVD), floppy disk, and blu-ray disc where disks usually reproduce data magnetically, while discs reproduce data optically with lasers. Combinations of the above are also included within the scope of non-transitory computer-readable and processor-readable media. Additionally, the operations of a method or algorithm may reside as one or any combination or set of codes, instructions, and/or operations on a non-transitory processor-readable medium and/or computer-readable medium, which may be incorporated into a computer program product.

The preceding description of the disclosed aspects is provided to enable any person skilled in the art to make or use the present invention. Various modifications to these aspects will be readily apparent to those skilled in the art, and the generic principles defined herein may be applied to other aspects without departing from the spirit or scope of the invention. Thus, the present invention is not intended to be limited to the aspects shown herein but is to be accorded the widest scope consistent with the following claims and the principles and novel features disclosed herein.

What is claimed is:

1. A method for increasing a processing speed of dynamic language software on a computing device, comprising:
initializing a first inline cache for a first instance of a dynamic software operation by a processor;
storing the first inline cache in a memory configured to provide fast access for storing and retrieving the first inline cache;
receiving a second instance of the dynamic software operation in a coprocessor;
determining whether the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same by the coprocessor;
executing the second instance of the dynamic software operation by the coprocessor using the first inline cache from the memory in response to determining that the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same;
2. The method of claim 1, wherein determining whether the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same comprises:
returning a result of executing the second instance of the dynamic software operation by the coprocessor to the processor;
3. The method of claim 1, further comprising:
initializing a second inline cache for the second instance of the dynamic software operation including the first inline cache configured to replace the initialized first inline cache in response to determining that the first instance of the dynamic software operation and the second instance of the dynamic software operation are different;
storing the second inline cache in the memory configured to provide fast access for storing and retrieving the second inline cache; and
executing the second instance of the dynamic software operation by the coprocessor using the second inline cache from the memory in response to determining that the first instance of the dynamic software operation and the second instance of the dynamic software operation are different;
4. The method of claim 1, further comprising determining whether the first inline cache exists for the first instance of the dynamic software operation, wherein initializing the first inline cache for the first instance of the dynamic software operation by the processor comprises initializing the first inline cache for the first instance of the dynamic software operation by the processor in response to determining that the first inline cache for the first instance of the dynamic software operation does not exist;
5. The method of claim 1, wherein initializing the first inline cache for the first instance of the dynamic software operation by the processor comprises:
traversing an object implementation for the dynamic software operation until identifying a data of the object implementation relating to the first instance of the dynamic software operation;
executing the dynamic software operation of the first instance of the dynamic software operation; and
returning a result of the first instance of the dynamic software operation;
6. The method of claim 1, wherein returning the result of executing the second instance of the dynamic software operation by the coprocessor to the processor comprises returning the result to the processor indirectly through a data cache accessible to the processor and the coprocessor;
7. The method of claim 1, wherein returning the result of executing the second instance of the dynamic software operation by the coprocessor comprises returning the result to the processor directly to the processor;
8. The method of claim 1, wherein:
storing the first inline cache in the memory configured to provide fast access for storing and retrieving the first inline cache comprises receiving the first inline cache from the processor disposed on a processor pipeline at

the memory disposed on an inline cache pipeline connected to the processor pipeline, receiving the second instance of the dynamic software operation at the coprocessor comprises receiving the second instance of the dynamic software operation from the processor disposed on the processor pipeline at the coprocessor disposed on the inline cache pipeline connected to the processor pipeline, and returning the result of executing the second instance of the dynamic software operation by the coprocessor comprising sending the result of executing the second instance of the dynamic software operation from the coprocessor disposed on the inline cache pipeline to the processor disposed on the processor pipeline connected to the inline cache pipeline.

9. The method of claim 1, further comprising:

- generating executable operations for the coprocessor by a computer; and
- instructing the processor to cause the coprocessor to execute the generated executable operations to perform operations comprising:

initializing the first inline cache in the first instance of the dynamic software operation by a processor;

storing the first inline cache in the memory configured to provide fast access for storing and retrieving the first inline cache;

receiving the second instance of the dynamic software operation in the coprocessor;

determining whether the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same;

executing the second instance of the dynamic software operation by the coprocessor;

10. A computing device, comprising:

- a memory configured to provide fast access for storing and retrieving at least one inline cache communicatively connected to the processor;
- a coprocessor communicatively connected to the processor and the memory;
- the processor configured with processor-executable instructions to perform operations comprising:

initializing a first inline cache for a first instance of a dynamic software operation;

storing the first inline cache in the memory;

the coprocessor configured with processor-executable instructions to perform operations comprising:

receiving a second instance of the dynamic software operation;

determining whether the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same;

executing the second instance of the dynamic software operation using the first inline cache from the memory in response to determining that the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same; and

returning a result of executing the second instance of the dynamic software operation to the processor.

11. The computing device of claim 10, wherein the coprocessor is further configured with processor-executable instructions to perform operations such that determining whether the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same comprises:

- comparing a first object implementation related to the first instance of the dynamic software operation with a second object implementation related to the second instance of the dynamic software operation; and
- determining whether the first object implementation and the second object implementation are the same.

12. The computing device of claim 10, wherein:

the processor is further configured with processor-executable instructions to perform operations comprising:

- initializing a second inline cache for the second instance of the dynamic software operation including the first inline cache configured to replace the initialized first inline cache in response to determining that the first instance of the dynamic software operation and the second instance of the dynamic software operation are different;
- storing the second inline cache in the memory; and
- the coprocessor is further configured with processor-executable instructions to perform operations comprising:

executing the second instance of the dynamic software operation using the second inline cache from the memory in response to determining that the first instance of the dynamic software operation and the second instance of the dynamic software operation are different.

13. The computing device of claim 10, wherein the coprocessor is further configured with processor-executable instructions to perform operations such that returning the result of executing the second instance of the dynamic software operation to the processor comprises returning the result directly to the processor.

14. The computing device of claim 10, further comprising:

- a data cache communicatively connected to the processor and the coprocessor, wherein the coprocessor is further configured with processor-executable instructions to perform operations such that returning the result of executing the second instance of the dynamic software operation to the processor comprises returning the result to the processor indirectly through the data cache accessible to the processor and the coprocessor.

15. The computing device of claim 10, further comprising:

- a processor pipeline communicatively connected to the processor;
- an inline cache pipeline communicatively connected to the processor pipeline, the coprocessor, and the memory;
- wherein the processor is further configured with processor-executable instructions to perform operations such that storing the first inline cache in the memory comprises sending the first inline cache from the processor to the memory via the processor pipeline and the inline cache pipeline;
- wherein the coprocessor is further configured with processor-executable instructions to perform operations such that:

receiving the second instance of the dynamic software operation comprises receiving the second instance of the dynamic software operation from the processor at the coprocessor disposed via the processor pipeline and the inline cache pipeline; and

- returning the result of executing the second instance of the dynamic software operation by the coprocessor comprises sending the result of executing the second instance of the dynamic software operation from the coprocessor to the processor via the inline cache pipeline and the processor pipeline.
- 16.** The computing device of claim 10, wherein:
 the processor is further configured with processor executable instructions to perform operations comprising:
 generating executable operations for the coprocessor using a compiler;
 instructing the coprocessor to execute the generated executable operations to perform operations comprising:
 initializing the first inline cache for the first instance of the dynamic software operation;
 storing the first inline cache in the memory;
 receiving the second instance of the dynamic software operation;
 determining whether the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same;
 executing the second instance of the dynamic software operation using the first inline cache from the memory in response to determining that the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same; and
 removing the result of executing the second instance of the dynamic software operation to the processor.
- 17.** A non-transitory processor-readable medium having stored thereon processor-executable software instructions to cause a processor and a coprocessor to perform operations comprising:
 initializing a first inline cache for a first instance of a dynamic software operation by the processor;
 storing the first inline cache in a memory configured to provide fast access for storing and retrieving the first inline cache;
 receiving a second instance of the dynamic software operation in the coprocessor;
 determining whether the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same by the coprocessor;
 executing the second instance of the dynamic software operation by the coprocessor using the first inline cache from the memory in response to determining that the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same; and
 returning a result of executing the second instance of the dynamic software operation by the coprocessor to a processor.
- 18.** The non-transitory processor-readable medium of claim 17, wherein the stored processor-executable software instructions are configured to cause the processor and the coprocessor to perform operations such that determining whether the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same comprises:
 comparing a first object implementation related to the first instance of the dynamic software operation with a second object implementation related to the second instance of the dynamic software operation; and
 determining whether the first object implementation and the second object implementation are the same.
- 19.** The non-transitory processor-readable medium of claim 17, wherein the stored processor-executable software instructions are configured to cause the processor and the coprocessor to perform operations further comprising:
 initializing a second inline cache for the second instance of the dynamic software operation including the first inline cache configured to replace the initialized first inline cache in response to determining that the first instance of the dynamic software operation and the second instance of the dynamic software operation are different;
 storing the second inline cache in the memory configured to provide fast access for storing and retrieving the second inline cache; and
 executing the second instance of the dynamic software operation by the coprocessor using the second inline cache from the memory in response to determining that the first instance of the dynamic software operation and the second instance of the dynamic software operation are different.
- 20.** The non-transitory processor-readable medium of claim 17, wherein the stored processor-executable software instructions are configured to cause the processor and the coprocessor to perform operations such that returning the result of executing the second instance of the dynamic software operation by the coprocessor to the processor comprises returning the result directly to the processor.
- 21.** The non-transitory processor-readable medium of claim 17, wherein the stored processor-executable software instructions are configured to cause the processor and the coprocessor to perform operations such that returning the result of executing the second instance of the dynamic software operation by the coprocessor to the processor comprises returning the result to the processor indirectly through a data cache accessible to the processor and the coprocessor.
- 22.** The non-transitory processor-readable medium of claim 17, wherein the stored processor-executable software instructions are configured to cause the processor and the coprocessor to perform operations such that:
 storing the first inline cache in the memory configured to provide fast access for storing and retrieving the first inline cache comprises receiving the first inline cache from the processor disposed on a processor pipeline at the memory disposed on an inline cache pipeline connected to the processor pipeline; and
 receiving the second instance of the dynamic software operation at the coprocessor comprises receiving the second instance of the dynamic software operation from the processor disposed on the processor pipeline at the coprocessor disposed on the inline cache pipeline connected to the processor pipeline; and
 returning the result of executing the second instance of the dynamic software operation by the coprocessor comprises sending the result of executing the second instance of the dynamic software operation from the coprocessor disposed on the inline cache pipeline to the processor disposed on the processor pipeline connected to the inline cache pipeline.
- 23.** The non-transitory processor-readable medium of claim 17, wherein the stored processor-executable software instructions are configured to cause the processor and the coprocessor to perform operations further comprising:
 generating executable operations for the coprocessor by a compiler; and
 instructing the processor to cause the coprocessor to execute the generated executable operations to perform operations comprising:

- initializing the first inline cache for the first instance of the dynamic software operation by a processor; storing the first inline cache in the memory configured to provide fast access for storing and retrieving the first inline cache; receiving the second instance of the dynamic software operation in the coprocessor; determining whether the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same; executing the second instance of the dynamic software operation by the coprocessor using the first inline cache from the memory in response to determining that the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same; and returning the result of executing the second instance of the dynamic software operation by the coprocessor.
- 24.** A computing device, comprising:
 means for initializing a first inline cache for a first instance of a dynamic software operation;
 means for storing the first inline cache configured to provide fast access for storing and retrieving the first inline cache;
 means for receiving a second instance of the dynamic software operation;
 means for determining whether the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same;
 means for executing the second instance of the dynamic software operation using the first inline cache from the means for storing the first inline cache in response to determining that the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same; and
 means for returning a result of executing the second instance of the dynamic software operation.
- 25.** The computing device of claim 24, wherein means for determining whether the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same comprises:
 means for comparing a first object implementation related to the first instance of the dynamic software operation with a second object implementation related to the second instance of the dynamic software operation; and
 means for determining whether the first object implementation and the second object implementation are the same.
- 26.** The computing device of claim 24, further comprising:
 means for initializing a second inline cache for the second instance of the dynamic software operation including the first inline cache configured to replace the initialized first inline cache in response to determining that the first instance of the dynamic software operation and the second instance of the dynamic software operation are different;
 means for storing the second inline cache configured to provide fast access for storing and retrieving the second inline cache; and
 means for executing the second instance of the dynamic software operation using the second inline cache from the second inline cache in response to determining that the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same; and
 means for returning the result of executing the second instance of the dynamic software operation.
- means for executing the second instance of the dynamic software operation using the second inline cache from the means for storing the second inline cache in response to determining that the first instance of the dynamic software operation and the second instance of the dynamic software operation are different;
- 27.** The computing device of claim 24, wherein means for returning the result of executing the second instance of the dynamic software operation comprises means for returning the result directly to a processor.
- 28.** The computing device of claim 24, wherein means for returning the result of executing the second instance of the dynamic software operation comprises means for returning the result to a processor indirectly through a data cache accessible to the processor and a coprocessor.
- 29.** The computing device of claim 24, wherein:
 means for storing the first inline cache configured to provide fast access for storing and retrieving the first inline cache comprises means for receiving the first inline cache from a processor disposed on a processor pipeline at means for storing the first inline cache on an inline cache pipeline connected to the processor pipeline;
 means for receiving the second instance of the dynamic software operation comprises means for receiving the second instance of the dynamic software operation from the processor disposed on the processor pipeline at means for receiving the second instance of the dynamic software operation disposed on the inline cache pipeline connected to the processor pipeline; and
 means for returning the result of executing the second instance of the dynamic software operation comprises means for sending the result of executing the second instance of the dynamic software operation disposed on the inline cache pipeline to the processor disposed on the processor pipeline connected to the inline cache pipeline.
- 30.** The computing device of claim 24, further comprising:
 means for generating executable operations by a compiler; and
 means for instructing to cause means for executing the generated executable operations to perform operations comprising:
 initializing the first inline cache for the first instance of the dynamic software operation;
 storing the first inline cache to provide fast access for storing and retrieving the first inline cache;
 receiving the second instance of the dynamic software operation;
 determining whether the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same;
 executing the second instance of the dynamic software operation using the first inline cache in response to determining that the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same; and
 returning the result of executing the second instance of the dynamic software operation.

* * * *



US009740504B2

(12) **United States Patent**
Robatmili et al.(10) **Patent No.:** US 9,740,504 B2
(11) **Date of Patent:** *Aug. 22, 2017(15) **HARDWARE ACCELERATION FOR INLINE CACHES IN DYNAMIC LANGUAGES**(21) **Applicant:** QUALCOMM Incorporated, San Diego, CA (US)(22) **Inventors:** Behnam Robatmili, San Jose, CA (US); Gheorghie Calin Casenval, Palo Alto, CA (US); Madhukar Nagaraja Kedlaya, Santa Clara, CA (US); Dario Suarez Graula, Santa Clara, CA (US)(23) **Assignee:** QUALCOMM Incorporated, San Diego, CA (US)(14) **Notice:** Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 446 days.
This patent is subject to a terminal disclaimer.(21) **Appl. No.:** 14/262,871(22) **Filed:** Apr. 28, 2014(16) **Prior Publication Data**

US 2015/0205720 A1 Jul. 23, 2015

(17) **Related U.S. Application Data**

(160) Provisional application No. 61/939,818, filed on Jun. 23, 2014.

(151) **Int. CL**

G06F 9/44 (2006.01)

G06F 12/08 (2016.01)

G06F 12/0802 (2016.01)

(152) **U.S. Cl.**

CPC G06F 9/441 (2013.01); G06F 12/0802 (2013.01)

(158) **Field of Classification Search**

CPC G06F 9/4431; G06F 12/0802

See application file for complete search history

(16) **References Cited**

U.S. PATENT DOCUMENTS

7,153,664 B1 4,352,878 Chong et al. 2002/0087598 A1 * 7,395,722 Perinchery 6,064,177 3,048 711,131 (Continued)

OTHER PUBLICATIONS

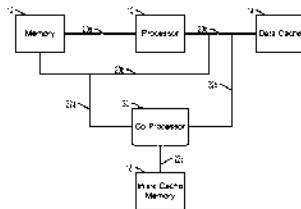
Advances in JavaScript Performance in 21 IoT and Windows 8, 29(2), Retrieved on May 2, 2014. Retrieved from the Internet : URL <http://blogs.msdn.com/b/mburns/2012/06/24/advances-in-javascript-performance-in-iot-and-windows-8.aspx>, 17 Pages

(Continued)

Primary Examiner: Michael Alsip
(74) Attorney, Agent or Firm: The Marbury Law Group, PLLC(17) **ABSTRACT**

Aspects include apparatuses, systems, and methods for hardware acceleration for inline caches in dynamic languages. An inline cache may be initialized for instance of a dynamic software operation. A call of an initialized instance of the dynamic software operation may be executed by an inline cache hardware accelerator. The inline cache may be checked to determine that its data is current. When the data is current, the initialized instance of the dynamic software operation may be executed using the reduced inline cache data. When the data is not current, a new inline cache may be initialized for the instance of the dynamic software operation, including the not current data of a previously initialized instance of the dynamic software operation. The inline cache hardware accelerator may include an inline cache memory, a coprocessor, and/or a functional unit one or more inline cache pipeline connected to a processor pipeline

30 Claims, 8 Drawing Sheets



(56)

References Cited**U.S. PATENT DOCUMENTS**

- 20090093128 AU 1 2009 Peterson 4654844
201210304159 AU* 11 2012 Czec 717149
201310295260 AU 8 2013 Borodughi et al 6061120815
20150987267 AU* 3 2015 Pizlo 711141
20150987658 AU 3 2015 Halenborg
20150205720 AU 7 2015 Robennil et al

OTHER PUBLICATIONS

- J. S., et al., "TypeCast: Dynamic Typing of JavaScript Applications," High Performance Embedded Architectures and Compilers, 2011, 11 pages
Zakaria, "A study of superpositions and dynamic method optimization," July 2011, 10 Pages
Cao, S., et al., "JVM Virtual Method Invoking Optimization Based on VM Table," 10th IEEE International Conference on Network, Architecture and Storage (NCS), Vol. 28, 3rd Edt., 28, 2011, pp. 122-128, XID33048112, DOI: 10.1109/NCS.2011.11, ISBN: 978-1-4577-1172-8
Vasudevan, A., et al., "Object-Oriented Architectural Support for a Java Virtual Machine," In Lecture Notes in Computer Sciences, 5th Ed., 1998, Chan, L., 1998, Springer Berlin Heidelberg, Berlin, Heidelberg, XID33176346, ISSN: 0302-9723, ISBN: 978-3-540-62348-8, vol. 1445, pp. 330-354, DOI: 10.1007/3-540-62348-8
International Search Report and Written Opinion - PCT/US2015/022529, ISA/PO, Mar. 26, 2015

* cited by examiner

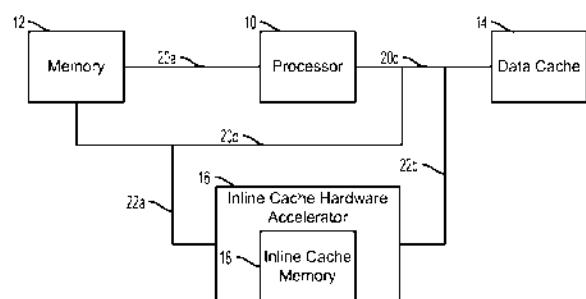


FIG. 1

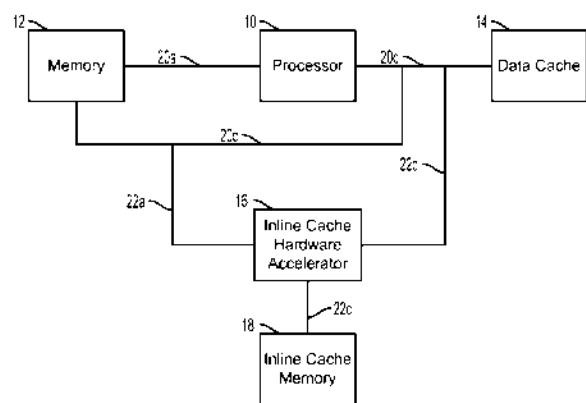


FIG. 2

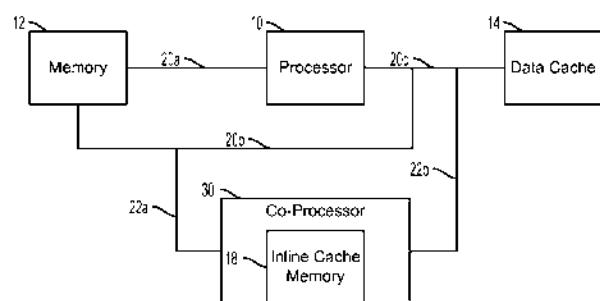


FIG. 3

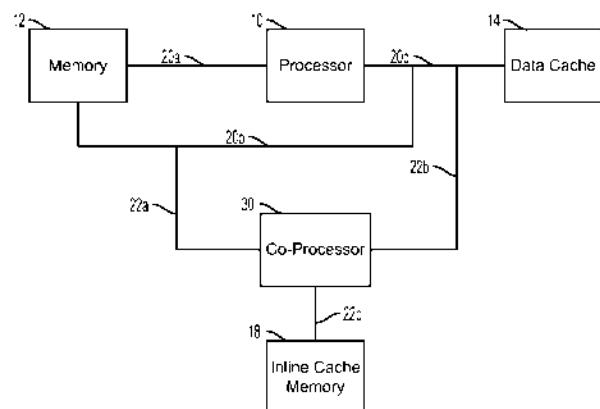


FIG. 4

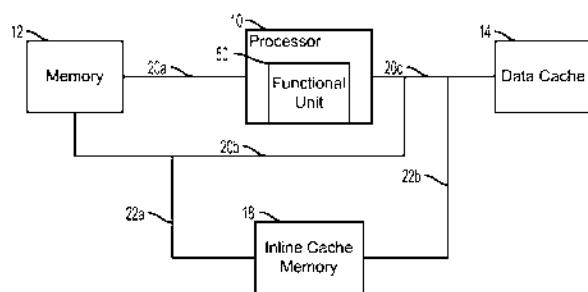


FIG. 5

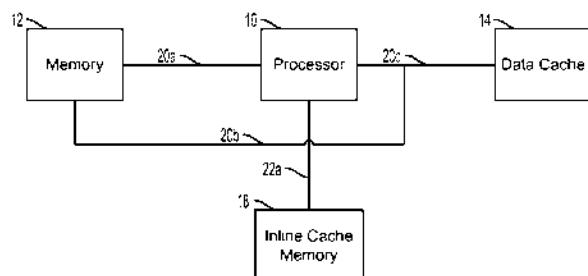


FIG. 6

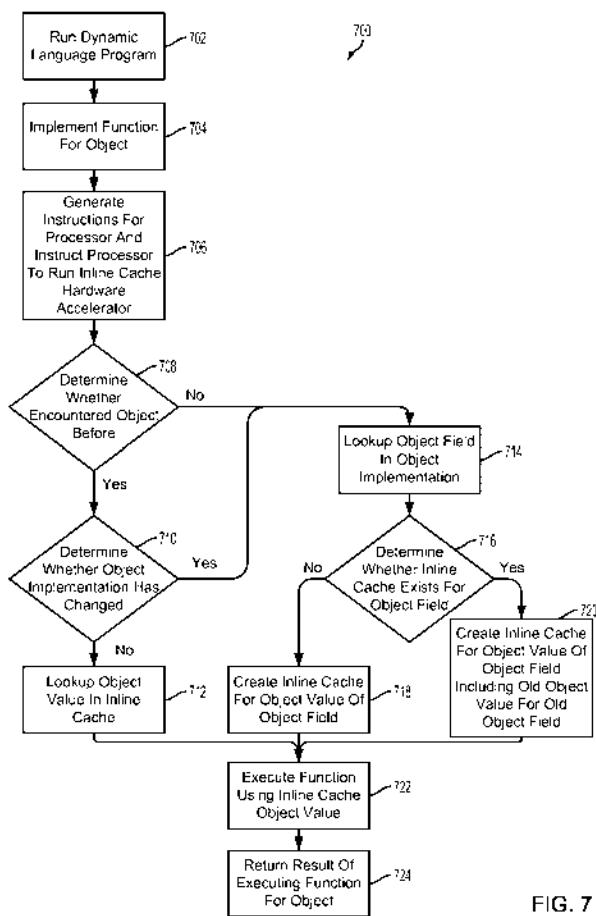


FIG. 7

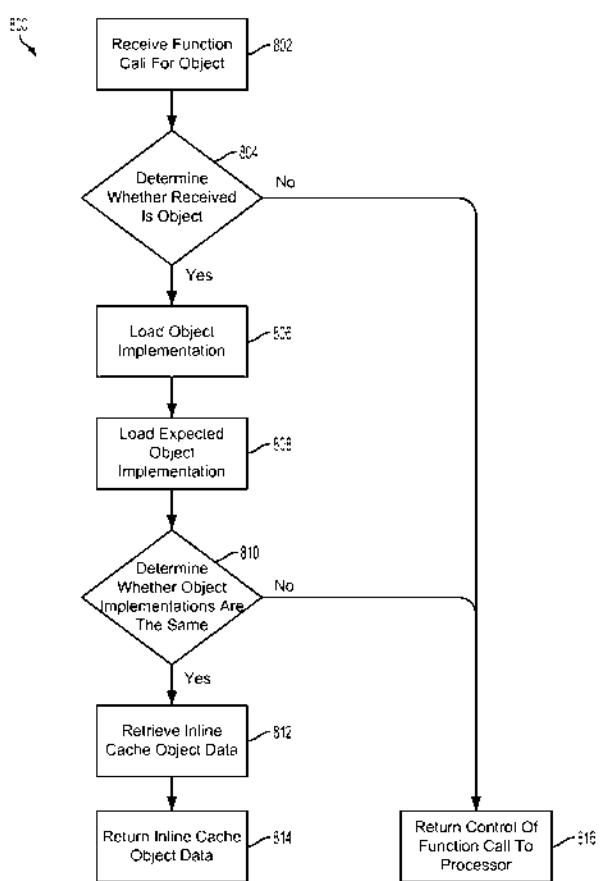


FIG. 8

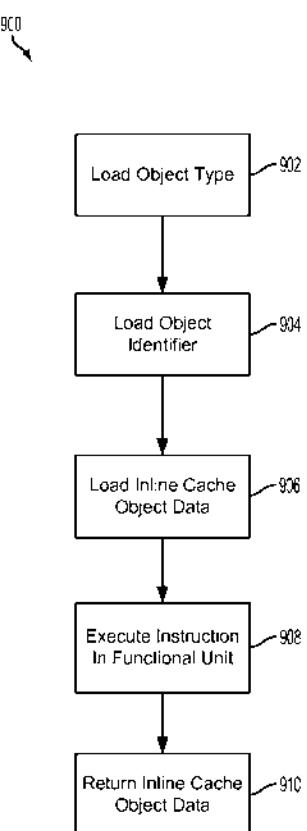


FIG. 9

U.S. Patent

Aug. 22, 2017

Sheet 7 of 8

US 9,740,504 B2

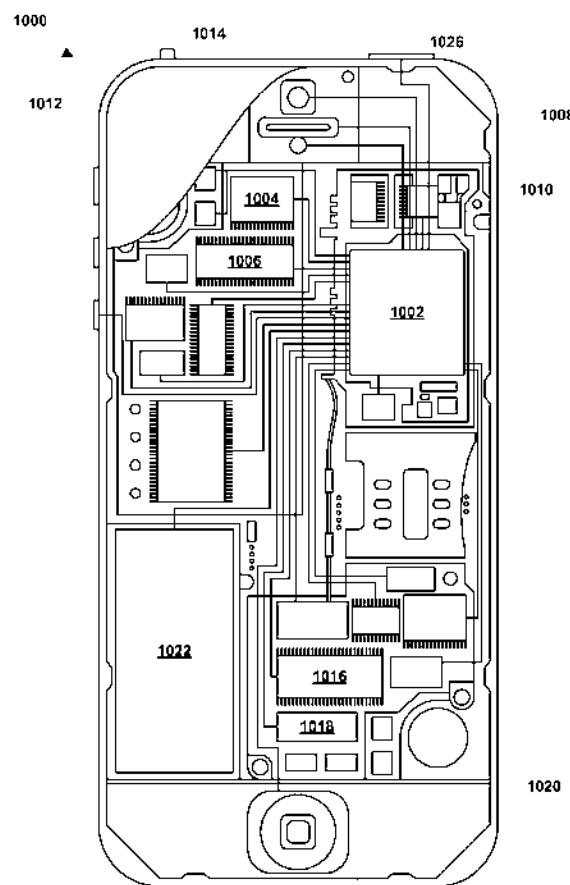


FIG. 10

U.S. Patent

Aug. 22, 2017

Sheet 8 of 8

US 9,740,504 B2

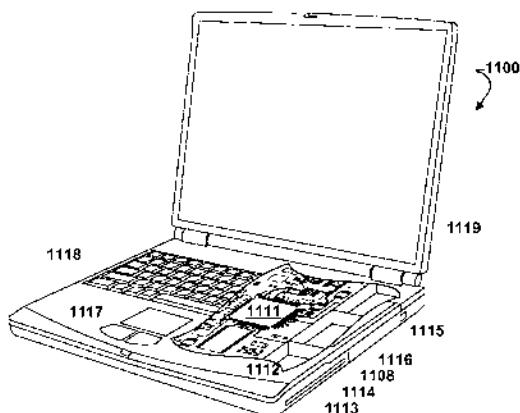


FIG. 11

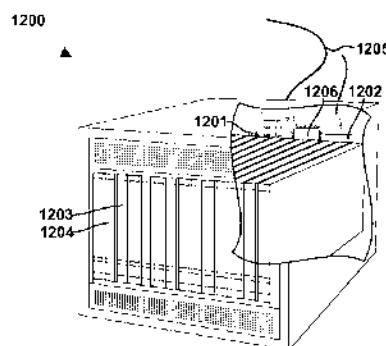


FIG. 12

1

2

HARDWARE ACCELERATION FOR INLINE CACHES IN DYNAMIC LANGUAGES

RELATED APPLICATIONS

This application claims the benefit of priority to U.S. Provisional Application No. 61,930,818 entitled "Hardware Acceleration For Inline Caches In Dynamic Languages" filed Jan. 23, 2014, the entire contents of which are hereby incorporated by reference.

BACKGROUND

Dynamic programming languages, such as JavaScript, Python and Ruby, are often used to execute common behaviors at runtime that other languages may execute while compiling the code. Dynamic programming languages increase the flexibility of a software program, often slowing down execution due to additional runtime compilation. Inline caches are a technique frequently used to reduce code execution overhead for dynamic languages by generating "first calls" from common templates for the common behaviors. However, inline caches increase the memory usage of the program by storing additional inline cached code and constant values. In particular for mobile devices, memory is a constrained resource.

SUMMARY

The various aspects focus on methods and apparatuses for increasing the processing speed of dynamic language software on a computing device. Aspect methods may include initializing a first inline cache for an first instance of a dynamic software operation by a processor, storing the first inline cache in a memory configured to provide fast access for storing and retrieving the first inline cache, receiving a second instance of the dynamic software operation in the processor, determining whether the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same, executing the second instance of the dynamic software operation by a functional unit using the first inline cache from the memory in response to determining that the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same, and returning the result of executing the second instance of the dynamic software operation by the functional unit.

An aspect method may further include receiving an object type for the second instance of the dynamic software operation by the functional unit, receiving an object identifier for the second instance of the dynamic software operation by the functional unit, and receiving inline cache data for the second instance of the dynamic software operation by the functional unit. In an aspect, executing the second instance of the dynamic software operation by the functional unit using the first inline cache from the memory in response to determining that the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same may include using the object type, the object identifier, and the inline cache data for the second instance of the dynamic software operation as parameters for executing the second instance of the dynamic software operation.

In an aspect, determining whether the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same may include comparing a first object implementation related to the first

instance of the dynamic software operation with a second object implementation related to the second instance of the dynamic software operation, and determining whether the first object implementation and the second object implementation are the same.

An aspect method may further include initializing a second inline cache for the second instance of the dynamic software operation including the first inline cache configured to replace the initialized first inline cache in response to determining that the first instance of the dynamic software operation and the second instance of the dynamic software operation are different, storing the second inline cache in the memory configured to provide fast access for storing and retrieving the second inline cache, and executing the second instance of the dynamic software operation by the functional unit using the second inline cache from the memory in response to determining that the first instance of the dynamic software operation and the second instance of the dynamic software operation are different.

An aspect method may further include determining whether the first inline cache exists for the first instance of the dynamic software operation, and in which initializing the first inline cache for the first instance of the dynamic software operation by the processor may include initializing the first inline cache for the first instance of the dynamic software operation by the processor in response to determining that the first inline cache for the first instance of the dynamic software operation does not exist.

In an aspect, initializing the first inline cache for the first instance of the dynamic software operation by the processor may include traversing an object implementation for the dynamic software operation until identifying a data of the object implementation relating to the first instance of the dynamic software operation, executing the dynamic software operation of the first instance of the dynamic software operation, and returning a result of the first instance of the dynamic software operation.

In an aspect, returning the result of executing the second instance of the dynamic software operation by the functional unit may include returning the result directly to the processor.

In an aspect, returning the result of executing the second instance of the dynamic software operation by the functional unit may include returning the result to the processor indirectly through a data cache accessible to the processor and the functional unit.

An aspect method may further include generating software instructions for the processor by a compiler, and instructing the processor to run the functional unit to perform operations that may include initializing the first inline cache for the first instance of the dynamic software operation by a processor, storing the first inline cache in the memory configured to provide fast access for storing and retrieving the first inline cache, receiving the second instance of the dynamic software operation in the processor, determining whether the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same, executing the second instance of the dynamic software operation by the functional unit using the first inline cache from the memory in response to determining that the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same, and returning the result of executing the second instance of the dynamic software operation by the functional unit.

An aspect includes a computing device having a processor communicatively connected to a memory and configured

with processor-executable instructions to perform operations of one or more of the aspect methods described above.

An aspect includes a non-transitory computer-readable medium having stored thereon processor-executable software instructions to cause a processor and a functional unit to perform operations of one or more of the aspect methods described above.

An aspect includes a computing device having means for performing functions of one or more of the aspect methods described above.

BRIEF DESCRIPTION OF THE DRAWINGS

The accompanying drawings, which are incorporated herein and constitute part of this specification, illustrate exemplary aspects of the invention, and together with the general description given above and the detailed description given below, serve to explain the features of the invention.

FIG. 1 is a component block diagram illustrating a computing device having an inline cache hardware accelerator and an inline cache memory attached to a processor pipeline, for hardware acceleration for inline caches in dynamic languages, in accordance with another aspect.

FIG. 2 is a component block diagram illustrating a computing device having an inline cache hardware accelerator and an inline cache memory attached to a processor pipeline, for hardware acceleration for inline caches in dynamic languages, in accordance with another aspect.

FIG. 3 is a component block diagram illustrating a computing device having a coprocessor and an inline cache memory attached to a processor pipeline, for hardware acceleration for inline caches in dynamic languages, in accordance with another aspect.

FIG. 4 is a component block diagram illustrating a computing device having a coprocessor and an inline cache memory attached to a processor pipeline, for hardware acceleration for inline caches in dynamic languages, in accordance with another aspect.

FIG. 5 is a component block diagram illustrating a computing device having a functional unit and an inline cache memory attached to a processor pipeline, for hardware acceleration for inline caches in dynamic languages, in accordance with another aspect.

FIG. 6 is a component block diagram illustrating a computing device having an inline cache memory attached to a processor pipeline, for hardware acceleration for inline caches in dynamic languages, in accordance with another aspect.

FIG. 7 is a process flow diagram illustrating an aspect method for utilizing inline cache code and constant initialization using hardware acceleration for inline caches in dynamic languages.

FIG. 8 is a process flow diagram illustrating an aspect method for use of inline cache code and constants by a functional unit for inline caches in dynamic languages.

FIG. 9 is a process flow diagram illustrating an aspect method for use of inline cache code and constants by a functional unit for inline caches in dynamic languages.

FIG. 10 is component block diagram illustrating an exemplary mobile computing device suitable for use with the various aspects.

FIG. 11 is component block diagram illustrating an exemplary computing device suitable for use with the various aspects.

FIG. 12 is component block diagram illustrating an exemplary server device suitable for use with the various aspects.

DETAILED DESCRIPTION

The various aspects will be described in detail with reference to the accompanying drawings. Wherever pos-

sible, the same reference numbers will be used throughout the drawings to refer to the same or like parts. References made to particular examples and implementations are for illustrative purposes, and are not intended to limit the scope of the invention or the claims.

The word "exemplary" is used herein to mean "serving as an example, instance, or illustration." Any implementation described herein as "exemplary" is not necessarily to be construed as preferred or advantageous over other implementations.

The term "computing device" is used interchangeably herein to refer to any one or all of cellular telephones, smartphones, personal or mobile multimedia players, personal data assistants (PDAs), laptop computers, tablet computers, phablets, smartbooks, ultrabooks, palm-top computers, wireless electronic mail receivers, multimedia internet enabled cellular telephones, wireless gaming controllers, desktop computers, servers and similar personal or commercial electronic devices which include a memory, and a programmable processor.

The terms "system-on-chip" (SoC) and "integrated circuit" (IC) are used interchangeably herein to refer to a set of interconnected electronic circuits, typically, but not exclusively, including one or more hardware cores, memory units, and communication interfaces. A hardware core may include a variety of different types of processors, such as a general purpose processor, a central processing unit (CPU), a digital signal processor (DSP), a graphics processing unit (GPU), an accelerated processing unit (APU), an auxiliary processor, a single-core processor, and a multi-core processor. A hardware core may further embody other hardware and hardware combinations, such as a field programmable gate array (FPGA), an application-specific integrated circuit (ASIC), other programmable logic device, discrete gate logic, transistor logic, performance monitoring hardware, watchdog hardware, and time references. Integrated circuits may be configured such that the components of the integrated circuit reside on a single piece of semiconductor material, such as silicon. Such a configuration may also be referred to as the IC components being on a single chip.

Inline caching is a technique used to speedup dynamic compilation for languages such as JavaScript, PHP, Python, and Ruby. A compiler (e.g., a static compiler, a runtime compiler, or a dynamic compiler) may identify patterns of bytecode that exhibit common behaviors and may use code templates to generate executable code. The generated code may be parameterized with some object information and stored in an inline cache. The compiler may place guards to check whether the object matches the generated code, retrieve the code from the inline cache, and call it. The inline cache code may obviate the need for generating the same sequence repeatedly.

However, inline caching introduces a different set of performance issues. In a typical computing device (or computing system), inline cache items are still executed by the processor and stored in the memory. Also, many of the common tasks may generate an inline cache item for various objects when the same inline cache item could provide the correct result when using the proper parameters. Thus, inline caching techniques may clutter up the processor, memory bus or pipeline and use up computing and power resources.

The various aspects include methods, devices, and non-transitory processor-readable storage media for increasing the processing speed of dynamic language software on a computing device. Hardware acceleration for inline caches in dynamic languages may utilize dedicated resources to manage the inline cache items for common tasks to take

the burden off of the typical components of a computing device, such as an application processor, pipeline, and memory. In particular, an inline cache hardware accelerator may include at least some dedicated memory and some dedicated processing. The dedicated memory may be implemented as a separate memory, as a part of a data cache, or as part of a system memory. The dedicated processing may be implemented as an extension to an existing processor execution pipeline, as a processor functional unit, as a coprocessor communicating with a core processor through existing coprocessor interfaces, or as a separate unit communicating with the processor through a proprietary interface. Hardware acceleration for inline caches in dynamic languages may be implemented along with the constant inline caching, such that certain tasks are dedicated for hardware acceleration while others, likely the less common tasks, may continue to be managed by the typical components of the computing device.

The components for hardware acceleration for inline caches in dynamic languages may be separate but connected to the typical components of the computing device. For example, the computing device may contain a processor pipeline for transmitting signals between the various other components, such as the processor and a data cache. The components for hardware acceleration for inline caches in dynamic languages may be connected to each other and to the processor pipeline by an inline cache pipeline that may allow the components for hardware acceleration for inline caches in dynamic languages to transmit signals to each other and to the typical components of the computing device. This arrangement may allow the hardware acceleration for inline caches in dynamic languages to operate without impeding the operation of the typical components until communication with the typical components is necessary. The components for hardware acceleration for inline caches in dynamic languages may communicate with certain of the typical components either directly via the connected pipelines, or indirectly through the other typical components.

FIG. 1 illustrates a computing device having an inline cache hardware accelerator and an inline cache memory attached to a processor pipeline, for hardware acceleration for inline caches in dynamic languages, in accordance with an aspect. A computing device may include a processor **10**, registers or a memory **12**, such as an accumulator or for storing stacks or register files, a data cache **14**, an inline cache hardware accelerator **16**, an inline cache memory **18**, a processor pipeline **20a**, **20b**, **20c**, and an inline cache pipeline **22a**, **22b**. The processor **10** may be one or more of a variety of different types of processors as described above. The processor **10** may be configured to execute coupled and/or interpreted executable operations from software programs. In an aspect, the software programs may be written in a dynamic programming language and the executable operations may be compiled or interpreted at runtime. The processor **10** may also be configured to execute sets of instructions in storage and interact with other components of the computing device to execute portions of the dynamic programming language software programs.

The memory **12** may be configured to store state values for various states of the computing device. The state values stored in the memory **12** may be accessible for read and write operations by the processor **10**. The data cache **14** may be configured to store data related to the executable operations executed by the processor **10**. The data stored in the data cache **14** may be input to the processor **10** to execute the executable operations, or output by the processor **10** result-

ing from the executed operations and stored for later use or for access by other components of the computing device. The processor **10**, the memory **12**, and the data cache **14** may be connected to each other by the processor pipeline **20a**, **20b**, **20c**. The processor pipeline **20a**, **20b**, **20c** may be configured to transmit signals representing the executable operations, the computing device states, and the software program data between the components connected to it.

The inline cache hardware accelerator **16** which may be configured to increase the speed of processing executable operations from dynamic programming language software programs for “dynamic software operations”. The configurations of the inline cache hardware accelerator **16** are discussed in further detail below. The inline cache memory **18** may be configured to provide fast access for storing and retrieving data, and may be configured to store constant values associated with instances of the dynamic software operations executed by the processor **10**, at least initially. The constant values may be associated with a particular instance of an object used to execute a dynamic software operation, such that the constant value may be recalled for future execution of the same instance of the object, for the dynamic software operation. In an aspect, the dynamic software operation may include one operation or a set of operations, and therefore references to singular operation and plural operations are not intended to limit the scope of the claims in terms of the number of executable operations explicitly recited in a claim.

The inline cache memory **18** may be dedicated for this specific purpose, and therefore may be relatively small and fast memory, because the storing of constants may not require a lot of space, and providing quick access to the inline cache memory **18** for read and write operations may facilitate increasing the processing speed of the dynamic software operations. Access to the inline cache memory **18** may be limited to the inline cache hardware accelerator **16**. In an aspect, the inline cache memory **18** may be included as part of the inline cache hardware accelerator **16**. An inline cache pipeline **22a**, **22b** may connect the inline cache hardware accelerator **16** and the inline cache memory **18** to the processor **10**, the memory **12**, and the data cache **14**. The inline cache pipeline **22a**, **22b** may connect directly to the other components of the computing device, or may connect to the processor pipeline **20a**, **20b**, **20c**. The inline cache pipeline **22a**, **22b** may be configured to transmit signals representing data used to execute dynamic software operations, the computing device states, and/or data resulting from the execution of the dynamic software, including the constant values stored on the inline cache memory **18**.

FIG. 2 illustrates a computing device having an inline cache hardware accelerator and an inline cache memory attached to a processor pipeline, for hardware acceleration for inline caches in dynamic languages, in accordance with an aspect. Similar to the computing device illustrated in FIG. 1, the computing device may include the processor **10**, the registers or the memory **12**, the data cache **14**, the inline cache hardware accelerator **16**, the inline cache memory **18**, the processor pipeline **20a**, **20b**, **20c**, and the inline cache pipeline **22a**, **22b**. In an aspect the inline cache memory **18** may be separate from the inline cache hardware accelerator **16**. In various aspects, the inline cache memory **18** may be a standalone memory, memory integrated into the cache hierarchy, or a portion of the system memory of the computing device. The computing device may also include an inline cache pipeline **22**, further configured to connect the inline cache hardware accelerator **16** and the inline cache memory **18**. The inline cache memory **18** may be accessible

by the inline cache hardware accelerator 16, and by the other components, such as the processor 10, through the inline cache hardware accelerator 16.

FIG. 3 illustrates a computing device having a coprocessor and an inline cache memory attached to a processor pipeline, for hardware acceleration for inline caches in dynamic languages, in accordance with an aspect. Similar to the computing device illustrated in FIG. 1, the computing device may include the processor 10, the registers or the memory 12, the data cache 14, the inline cache memory 18, the processor pipeline 20a, 20b, 20c, and the inline cache pipeline 22a, 22b. In an aspect the inline cache hardware accelerator may be a coprocessor 30 configured to execute the dynamic software operations. The coprocessor 30, like the processor 10, may be one or more of a variety of different types of processors as described above. In an aspect, the coprocessor 30 may be a programmable logic device programmed to execute one or more dynamic software operations. In an aspect, the programmable coprocessor 30 may be reprogrammable to execute a different dynamic software operation. In another aspect, the computing device may include multiple coprocessors 30, each configured to execute a specific dynamic software operation. The dynamic software operations may be typical executable operations and the one or more coprocessors 30 may be provided for inclusion in the computing device on the basis of the typical dynamic software operations for the computing device. In an aspect, the inline cache memory 18 may be included as part of and/or dedicated to the coprocessor 30. In another aspect, the inline cache memory 18 may be one or more inline cache memories 18 shared across multiple coprocessors 30. In an aspect, the coprocessor 30 may be implemented as a functional extension of the processor pipeline 20a, 20b, 20c, and configured with instructions to initialize the inline cache memory 18, to execute the code stored in the inline cache memory 18, and to query the inline cache memory 18 for the presence of code for a first instance of a dynamic software operation. In an aspect, the coprocessor 30 may be implemented as a standalone unit communicating with the processor 10 using standard coprocessor interfaces. Such communications may include transferring initial values, initiating computation, and transferring output results. In another aspect, the coprocessor 30 may be implemented as a standalone unit communicating with the processor 10 using an enhanced interface, allowing the processor 10 to provide more information to the coprocessor 30 (e.g., an object map).

As described in further detail below, the processor 10 may pass data to the coprocessor 30 relating to an instance of a dynamic software operation. The coprocessor 30 may determine whether there are any inline cache data relating to the instance of the dynamic software operation. When the coprocessor 30 determines that the instance of the dynamic software operation is not yet initialized (i.e., the dynamic software operation is “uninitialized”), or that it is the first instance of the dynamic software operation, the coprocessor 30 may signal the processor 10 indicating that it cannot process the instance of the dynamic software operation. The coprocessor 30 may determine that the instance of the dynamic software operation is initialized by comparing data stored in the inline cache memory 18 with the data received from the processor 10. When there is no data in the inline cache memory 18 for the instance of the dynamic software operation, the coprocessor 30 may determine that the instance is uninitialized. The processor 10 may initialize

the instance of the dynamic software operation and store the results of the instance as a constant value in the inline cache memory 18.

When there is data in the inline cache memory 18 for the instance of the dynamic software operation, the coprocessor 30 may determine that the instance is initialized. The coprocessor 30 may then determine whether the data stored in the inline cache memory 18 for the instance of the dynamic software operation is the correct data for the instance. The coprocessor 30 may determine that the stored data is not the correct data for the instance of the dynamic software operation by comparing the data stored in the inline cache memory 18 with the data received from the processor 10. When the data stored for the instance of the dynamic software operation does not match the data provided by the processor 10, the coprocessor 30 may determine that the data stored for the instance of the dynamic software operation is not the correct data, and as a result may signal the processor 10 indicating that the coprocessor 30 cannot process the instance of the dynamic software operation. The processor 10 may then initialize this instance of the dynamic software operation and store the results of the instance as another constant value in the inline cache memory 18.

Stored data for an instance of a dynamic software operation may not match data received from the processor 10 due to a change in an implementation for the objects, such as object maps, for the dynamic software operations. Such a change may associate instances of the objects with different values, causing different results for functions for the instances of the objects, or instances of the dynamic software operations. Similar to the uninitialized instance above, an instance of a dynamic software operation may be initialized due to such a change.

When there is data in the inline cache memory 18 for the instance of the dynamic software operation and the data is correct data, the coprocessor 30 may return the constant value for the instance of the dynamic software operation to the processor 10. In an aspect, the above initialization of the instance for the dynamic software operation may include the processor 10 running the instance for the dynamic software operation as normal to implement its result. Some dynamic software operations are common and various instances of the dynamic software operations are called repeatedly. In many cases the same instance of the dynamic software operation is called repeatedly. The coprocessor 30 may be configured to increase the speed of the dynamic software operations by providing predetermined or precalculated results to the initialized instances of the dynamic software operation stored in a memory dedicated to storing the results (e.g., the inline cache memory 18).

FIG. 4 illustrates a computing device having a coprocessor and an inline cache memory attached to a processor pipeline, for hardware acceleration for inline caches in dynamic languages, in accordance with an aspect. Similar to the computing device illustrated in FIG. 3, the computing device may include the processor 10, the registers or the memory 12, the data cache 14, the inline cache memory 18, the processor pipeline 20a, 20b, 20c, the inline cache pipeline 22a, 22b, and the coprocessor 30. In an aspect the inline cache memory 18 may be separate from the coprocessor 30. The computing device may also include the inline cache pipeline 22c, further configured to connect the coprocessor 30 and the inline cache memory 18, like in FIG. 2. The inline cache memory 18 may be accessible by the coprocessor 30, and by the other components, such as the processor 10, through the coprocessor 30. In an aspect, the inline cache memory 18 may be dedicated to the coprocessor

US 9,740,504 B2

9

10

30 In another aspect, the inline cache memory **18** may be one or more inline cache memories **18** shared across multiple coprocessors **30**. Regardless of the difference in structures of the computing device illustrated in FIGs. 4 and the computing device illustrated in FIG. 3, the operation of the computing devices and their components are substantially similar.

In aspects including the coprocessor **30**, processors **10** may be configured to communicate with the coprocessor **30** and may not require alterations to the processors' instruction set architecture. In an aspect, data associated with the instances of the dynamic software operations may be explicitly passed from the processor **10** to the coprocessor **30**. In another aspect, the coprocessor **30** may be able to communicate directly with the memory **12** and the data cache **14**. This direct communication may allow for the coprocessor **30** to implicitly receive that data associated with the instances of the dynamic software operations from the processor **10** by way of the memory **12** and the data cache **14**. Function calls to the coprocessor **30** by the processor **10** may parallel type calls, which may allow for multiple calls simultaneously at the inline cache pipeline **22a**, **22b**, **22c**.

FIG. 5 illustrates computing device having a functional unit and an inline cache memory attached to a processor pipeline, for hardware acceleration for inline caches in dynamic languages, in accordance with an aspect. Similar to FIG. 2, the computing device may include the processor **10**, the registers or the memory **12**, the data cache **14**, the inline cache memory **18**, the processor pipeline **20a**, **20b**, **20c**, and the inline cache pipeline **22a**, **22b**. In an aspect the inline cache hardware accelerator may be a functional unit **50** configured to execute the dynamic software operations. The functional unit **50** may be a circuit configured to perform a specific function or calculation. For example, the functional unit **50** may be an adder or a multiplier. One or more functional units **50** may be combined to achieve the specific function or calculation. Multiple functional units **50** or groups of functional units **50** may be configured to implement a variety of different dynamic software operations. The functional unit **50** may be an integrated component of the processor **10**. In an aspect the inline cache memory **18** may be separate from the functional unit **50**. The computing device may also include an inline cache pipeline further reconfigured to connect the functional unit **50** and the inline cache memory **18**. The inline cache memory **18** may be accessible by the functional unit **50** and the other components, such as the processor **10**, through the functional unit **50**. In an aspect, the inline cache memory **18** may be dedicated to functional unit **50**. In another aspect, the inline cache memory **18** may be one or more inline cache memories **18** shared across multiple functional units **50**.

As described in further detail below, the processor **10** may determine whether the instance of the dynamic software operation is initialized or uninitialized similar to how the coprocessor makes the determination in the description above. When the instance of the dynamic software operation is uninitialized, the processor **10** may execute the instance of the dynamic software operation to determine the result. The processor **10** may store the result of the executed instance of the dynamic software operation in the inline cache memory **18**, such that the instance of the dynamic software operation

may be initialized. When the processor **10** determines that the instance of the dynamic software operation is initialized, the processor **10** may pass data to one or more selected functional units **50** relating to the initialized instance of the dynamic software operation. The functional unit **50** selected by the processor **10** may be configured specifically to execute the dynamic software operation received by the processor **10**. The processor **10** may match the dynamic software operation with the appropriate functional unit **50**, and pass the appropriate data to the selected functional unit **50**. The functional unit **50** may operate using the data from the processor **10** to determine the inline cache data relating to the instance of the dynamic software operation stored in the inline cache memory **18**. The functional unit **50** may use the data from the processor **10** and execute the dynamic software operation for which it is configured to implement. The result of the initialized instance of the dynamic software operation may be the constant value stored on the inline cache memory **18**, and may be passed to the processor **10** to complete the execution of the instance of the dynamic software operation.

In aspects including the functional unit **50**, processors **10** may or may not be configured to communicate with the functional unit **50** and may or may not require alterations to the processors' instruction set architecture. In an aspect, data associated with the instances of the dynamic software operations may be explicitly passed from the processor **10** to the functional unit **50**. In another aspect, the functional unit **50** may be able to communicate directly with the memory **12** and the data cache **14**. This direct communication may allow for the functional unit **50** to implicitly receive that data associated with the instances of the dynamic software operations from the processor **10** by way of the memory **12** and the data cache **14**. Function calls in the functional unit **50** by the processor **10** may include blocking type calls, where only one call may be handled at a time. Function calls to the functional unit **50** by the processor **10** may include non-blocking or asynchronous type calls, which may allow for multiple calls concurrently in the inline cache pipeline **22a**, **22b**. Similarly, the function calls to the functional unit **50** by the processor **10** may parallel type calls, which may allow for multiple calls simultaneously in the inline cache pipeline **22a**, **22b**.

FIG. 6 illustrates computing device having an inline cache memory attached to a processor pipeline, for hardware acceleration for inline caches in dynamic languages, in accordance with an aspect. Similar to the computing device illustrated in FIG. 1, the computing device may include a processor **10**, registers or the memory **12**, a data cache **14**, an inline cache memory **18**, a processor pipeline **20a**, **20b**, **20c**, and an inline cache pipeline **22a**. In an aspect, the inline cache hardware accelerator may be the inline cache memory **18** configured to store results of the instances of the dynamic software operations. In an aspect, the inline cache memory **18** may be one or more inline cache memories **18** shared for multiple instances of the dynamic software operations, one or more inline cache memories **18** each dedicated to storing multiple instances of the same dynamic software operation, or one inline cache memory **18** dedicated to storing all of the instances of all of the dynamic software operations.

As described in further detail below, the processor **10** may determine whether the instance of the dynamic software operation is initialized or uninitialized similar to how the coprocessor makes the determination in the description above. When the instance of the dynamic software operation is uninitialized, the processor **10** may execute the instance of the dynamic software operation to determine the result. The

US 9,740,504 B2

11

12

processor 10 may store the result of the executed instance of the dynamic software operation in the inline cache memory 18, such that the instance of the dynamic software operation may be initialized. When the processor 10 determines that the instance of the dynamic software operation is initialized, the processor 10 may receive the data stored in the inline cache memory 18 for the initialized instance of the dynamic software operation without having to fully execute the operation as it would if the operation were uninitialized.

In each of the foregoing aspect computing devices, the components, including the processor 10, the registers or the memory 12, the data cache 14, the inline cache hardware accelerator 16, the inline cache memory 18, the processor pipeline 20a, 20b, 20c, the inline cache pipeline 22a, 22b, 22c, and the coprocessor 30, and the functional unit 50 may be configured in various combinations. Some or all of the components may comprise individual or combined components. Similarly, some or all of the components may be included as part of a SoC or one or more integrated circuits.

FIG. 7 illustrates an aspect method 700 for inline cache code and constant initialization using hardware acceleration for inline caches in dynamic languages. The computing device and its components, including the processor, the registers or the memory for register files, the data cache, the inline cache hardware accelerator, the inline cache memory, the processor pipeline, the inline cache pipeline, the coprocessor, and the functional unit, may execute the method 700. In block 702, the computing device may run the dynamic language executable program. In an aspect, the dynamic language executable program may be run in a web browser, a web application, or a standalone application. In block 704, the computing device may implement a function for an object or the instance of the dynamic software operation. Common dynamic software operations, such as loading data, storing data, calling certain functions, and performing binary operations on data may be the types of dynamic software operations that may be included for hardware acceleration. In block 706, a compiler (e.g., a static compiler, a runtime compiler, or a dynamic compiler) may generate executable instructions for the processor to instruct the processor or a coprocessor to interact with and run the inline cache hardware accelerator for the instance of the dynamic software operation. These instructions may cause the processor or a coprocessor to read from and write to the inline cache memory either directly or through the inline cache hardware accelerator and cause the inline cache hardware accelerator to function as described below.

In determination block 708, the computing device may determine whether the computing device has previously run the function for the object, or the instance of the dynamic software operation. In an aspect, the computing device may accomplish this by checking the inline cache memory for stored data related to the instance of the dynamic software operation. The determined presence of data related to the instance of the dynamic software operation on the inline cache memory may signify to the computing device that the instance of the dynamic software operation may be initialized. However the determination that the dynamic software operation is initialized may not be certain without a further determination discussed below. For example, data related to the instance of the dynamic software operation may be stored and thus may signify that a prior instance of the dynamic software operation was initialized, but that the data may not be current.

When the computing device determines that the instance of the dynamic software operation was previously run (i.e., determination block 708 “Yes”), the computing device may

determine whether the object implementation for the instance of the dynamic software operation has changed in determination block 710. Determining whether the object implementation of the instance of the dynamic software operation has changed may indicate to the computing device whether or not the data stored in the inline cache memory for the instance of the dynamic software operation is current data. When the computing device determines that the object implementation for the instance of the dynamic software operation has not changed (i.e., determination block 710 “No”), the computing device may look up the value for the instance of the dynamic software operation stored on the inline cache memory in block 712. Determining that data exists in the inline cache memory for the instance of the dynamic software operation and that the data is current together may signify to the computing device that the instance of the dynamic software operation is initialized. In block 722, the computing device may execute any operations on the data from the inline cache memory requested by the processor as part of the dynamic software operation. In block 724, the computing device may return the data related to the instance of the dynamic software operation to the processor. The data may be returned to the processor either directly or indirectly through the memory, for register files and/or the data cache.

When the computing device determines that the instance of the dynamic software operation was not previously run (i.e., determination block 708 “No”) or when the computing device determines that the object implementation for the instance of the dynamic software operation has changed (i.e., determination block 710 “Yes”), the computing device may lookup related fields for the operation in block 714. It may be common for the processor to traverse the object implementation item by item until it finds the appropriate field associated with an object value for the operation. This process can be time and resource consuming, and inline caching of the appropriate value helps to avoid repetition of this step for future instances of the dynamic software operation. Hardware acceleration furthers this effort by taking some of the processing involved in hardware caching from the processor and allocating it to the inline hardware accelerator, or by simplifying the storage and retrieval of the inline cached data.

In determination block 716, the computing device may determine whether an inline cache data exists for an instance of the dynamic software operation. This may be possible because a previous instance of the dynamic software operation may have been initialized but the current data for the instance of the dynamic operation may not match that of the previously initialized instance. When the computing device determines that no inline cache data exists for the instance dynamic software operation (i.e., determination block 716 “No”), the computing device may create, or initialize, an inline cache data for the instance of the dynamic software operation in block 718. The computing device may continue to block 722 to execute any operations on the data requested by the processor as part of the instance of the dynamic software operation, and may perform the operations of block 724 to return the result of the instance.

When the computing device determines that an inline cache data exists for the instance dynamic software operation (i.e., determination block 716 “Yes”), the computing device may create, or initialize, an inline cache data for the instance of the dynamic software operation including the inline cache data for the previous instance in block 720. For example, the computing device may initialize a second

inline cache including a first inline cache configured to replace the initialized first inline cache when a first instance of a dynamic software operation and a second instance of the dynamic software operation are different. The computing device may continue to block 722 to execute any operations on the data requested by the processor as part of the instance of the dynamic software operation, and may perform the operations of block 724 to return the result of the instance.

In various aspects, the inline cache may have multiple allocation and replacement policies. For example, in blocks 718 and 720, the computing device may not initialize the inline cache data for every instance of the dynamic software operation. For example, an inline cache data may be initialized only after a second execution of a dynamic software operation. Other specific criteria may determine when an inline cache data may be initialized for a dynamic software operation, such as frequency of execution according to various metrics, complexity and/or cost of execution in terms of time and/or resources, or operation by a running program or application. In an aspect, initialized inline cache data may be removed, or uninitialized, for various reasons. For example, an inline cache data may be explicitly evicted with an operation from the inline cache, by conventional policies such as least recently used, least frequently used, first-in-first-out, last-in-first-out, and age. Other criteria, such as least complex and/or costly or most inline cache memory usage, may also be factors for removing inline cache data. In some aspects, the performance or space availability of the inline cache memory may determine when an inline cache data may be considered for removal. Some criteria for initializing or uninitialized may take into account current and/or historical data in determining whether to initialize or uninitialized inline cache data for a dynamic software operation. Removed inline cache data may be initialized again according to any of the described criteria. It may be required to meet other, potentially more stringent criteria than inline cache data yet to be initialized, or may be prevented from initializing for at least a certain period or until a certain event occurs. For example, removed inline cache data may be prevented from initializing again until a next running of a program, or until the inline caching is reset, like upon a reboot of the computing device.

FIG. 8 illustrates an aspect method 800 for use of inline cache code and constants by a processor for inline caches in dynamic languages. The computing device and its components, including the processor, the registers or the memory for register files, the data cache, the inline cache hardware accelerator, the inline cache memory, the processor pipeline, the inline cache pipeline, the coprocessor, and the functional unit, may execute the method 800. In block 802, the computing device may receive a function call for an instance of an object, or an instance of the dynamic software operation. In determination block 804, the computing device may determine whether the received data is an object, rather than a small integer, for example. When the computing device determines that the received data is not an object (i.e., determination block 804 “No”), the computing device may return control of the function call to the processor in block 816.

When the computing device determines that the received data is an object (i.e., determination block 804 “Yes”), the computing device may load the current object implementation for the instance of the dynamic software operation in block 806. In block 808, the computing device may load the expected object implementation for the instance of the dynamic software operation. The expected object implementation may be the object implementation used to initialize a

previous instance of the dynamic software operation. In determination block 810, the computing device may determine whether the current object implementation and the expected object implementation are the same object implementation. This determination may be made by comparing an encoding of the current object with an encoding of the object stored with the inline cache at a time when the code in the inline cache was generated. This encoding may be an address of the object prototype, or an encoding of the object fields, or any other mechanism that can uniquely identify the structure of the object. It may be possible for an object implementation to change after the initialization of an instance of the dynamic software operation, and the result may be that the data from the object implementation for a current instance of the dynamic software operation may no longer match the data from the previous instance. Returning the wrong data based on the previous instance of the dynamic software operation may cause errors in the execution of the dynamic software. When the computing device determines that the current object implementation and the expected object implementation are different (i.e., determination block 810 “No”), the computing device may return control of the function call to the processor in block 816.

When the computing device determines that the current object implementation and the expected object implementation are the same (i.e., determination block 810 “Yes”), the computing device may retrieve the inline cache data at block 812. In block 814, the computing device may return the data related to the instance of the dynamic software operation to the processor. The data may be returned to the processor either directly or indirectly through the memory for register files and/or the data cache.

FIG. 9 illustrates an aspect method 900 for use of inline cache code and constants by a functional unit for inline caches in dynamic languages. The computing device and its components, including the processor, the registers or the memory for register files, the data cache, the inline cache hardware accelerator, the inline cache memory, the processor pipeline, the inline cache pipeline, the coprocessor, and the functional unit, may execute the method 900. At block 902, the computing device may load an object type. The object type may indicate to the computing device what dynamic software operation is being run. In block 904, the computing device may load an object identifier. The object identifier may indicate to the computing device which instance of the dynamic software operation is being run, including an input value, for example. Using the information from the instance, in block 906, the computing device may load the inline cache data for the instance of the dynamic software operation. In block 908, the computing device may execute the dynamic software operation on the data for the instance. The computing device may use the object type, the object identifier, and the inline cache data as parameters to execute the dynamic software operation. In block 910, the computing device may return the data related to the instance of the dynamic software operation to the processor. The data may be returned to the processor either directly or indirectly through the memory for register files and/or the data cache.

FIG. 10 illustrates an exemplary mobile computing device suitable for use with the various aspects. The mobile computing device 1000 may include a processor 1002 coupled to a touchscreen controller 1004 and an internal memory 1006. The processor 1002 may be one or more multi-core integrated circuits designated for general or specific processing tasks. The internal memory 1006 may be volatile or non-volatile memory, and may also be secure and/or encrypted memory, or unsecure and/or unencrypted memory, or any

combination thereof. The touchscreen controller **1004** and the processor **1002** may also be coupled to a touchscreen panel **1012**, such as a resistive-sensing touchscreen, capacitive-sensing touchscreen, infrared sensing touchscreen, etc. Additionally, the display of the mobile computing device **1000** need not have touch screen capability.

The mobile computing device **1000** may have one or more radio signal transceivers **1008** (e.g., Penman, Bluetooth, Zigbee, Wi-Fi, BT, radio) and antenna **1010**, for sending and receiving communications, coupled to each other and/or to the processor **1002**. The transceivers **1008** and antenna **1010** may be used with the above-mentioned circuitry to implement the various wireless transmission protocol stacks and interfaces. The mobile computing device **1000** may include a cellular network wireless modem chip **1016** that enables communication via a cellular network and is coupled to the processor.

The mobile computing device **1000** may include a peripheral device connection interface **1018** coupled to the processor **1002**. The peripheral device connection interface **1018** may be singularly configured to accept one type of connection, or may be configured to accept various types of physical and communication connections, common or proprietary, such as USB, IrWire, Thunderbolt, or PC. The peripheral device connection interface **1018** may also be coupled to a similarly configured peripheral device connection port (not shown).

The mobile computing device **1000** may also include speakers **1014** for providing audio outputs. The mobile computing device **1000** may also include a housing **1020**, constructed of a plastic, metal, or a combination of materials, for containing all or some of the components discussed herein. The mobile computing device **1000** may include a power source **1022** coupled to the processor **1002**, such as a disposable or rechargeable battery. The rechargeable battery may also be coupled to the peripheral device connection port to receive a charging current from a source external to the mobile device **1000**. The mobile computing device **1000** may also include a physical button **1024** for receiving user inputs. The mobile computing device **1000** may also include a power button **1026** for turning the mobile device **1000** on and off.

The various aspects described above may also be implemented within a variety of other types of computing devices, such as a laptop computing device **1100** illustrated in FIG. 11. Many laptops include a touchpad/touch surface **1117** that serves as the computer's pointing device, and thus may receive drag, scroll, and flick gestures similar to those implemented on computing devices equipped with a touch screen display, and described above. A laptop computing device **1100** will typically include a processor **1101** coupled to volatile memory **1112** and large capacity nonvolatile memory, such as a disk drive **1113** of Flash memory. Additionally, the computing device **1100** may have one or more antennas **1108** for sending and receiving electromagnetic radiation that may be connected to a wireless data link and/or cellular telephone transceiver **1116** coupled to the processor **1101**. The computing device **1100** may also include a floppy disc drive **1114** and a compact disc (CD) drive **1115** coupled to the processor **1101**. In a notebook configuration, the computing device housing includes the touchpad **1117**, the keyboard **1118**, and the display **1119** all coupled to the processor **1101**. Other configurations of the computing device may include a computer mouse or trackball coupled to the processor (e.g., via a USB input); as are well known, which may also be used in conjunction with the various aspects. A desktop computer may similarly include

these computing device components in various configurations, including separating and combining the components in one or more separate but connectable parts.

The various embodiments may also be implemented on any of a variety of commercially available server devices, such as the server **1200** illustrated in FIG. 12. Such a server **1200** typically includes a processor **1201** coupled to volatile memory **1202** and a large capacity nonvolatile memory, such as a disk drive **1204**. The server **1200** may also include a floppy disc drive, compact disc (CD) or DVD disc drive **1206** coupled to the processor **1201**. The server **1200** may also include network access ports **1203** coupled to the processor **1201** for establishing network interface connections with a network **1205**, such as a local area network coupled to other broad-area system computers and servers, the Internet, the public switched telephone network, and/or a cellular data network (e.g., CDMA, TDMA, GSM, PCS, 3G, 4G, 5G), or any other type of cellular data network.

The foregoing method descriptions and flow process flow diagrams are provided merely as illustrative examples and are not intended to require or imply that the operations of the various aspects must be performed in the order presented. As will be appreciated by one of skill in the art in the order of operations in the foregoing aspects may be performed in any order. Words such as "hereafter," "then," "next," etc. are not intended to limit the order of the operations; these words are simply used to guide the reader through the description of the methods. Further, any reference to clause elements in the singular, for example, using the articles "a," "an" or "the," is not to be construed as limiting the element to the singular.

The various illustrative logical blocks, modules, circuits, and algorithm operations described in connection with the various aspects may be implemented in electronic hardware, computer software, or combinations of both. To clearly illustrate this interchangeability of hardware and software, various illustrative components, blocks, modules, circuits, and operations have been described above generally in terms of their functionality. Whether such functionality is implemented as hardware or software depends upon the particular application and design constraints imposed on the overall system. Skilled artisans may implement the described functionality in varying ways for each particular application, but such implementation decisions should not be interpreted as causing a departure from the scope of the present invention.

The hardware used to implement the various illustrative logics, logical blocks, modules, and circuits described in connection with the aspects disclosed herein may be implemented in general with a general purpose processor, a digital signal processor (DSP), an application specific integrated circuit (ASIC), a field programmable gate array (FPGA), or other programmable logic device, discrete gate or transistor logic, discrete hardware components, or any combination thereof designed to perform the functions described herein. A general-purpose processor may be a microprocessor, but, in the alternative, the processor may be any conventional processor, controller, microcontroller, or state machine. A processor may also be implemented as a combination of computing devices, e.g., in combination of a DSP and a microprocessor, a plurality of microprocessors, one or more microprocessors in conjunction with a DSP core, or any other such configuration. Alternatively, some operations or methods may be performed by circuitry that is specific to a given function.

In one or more aspects, the functions described may be implemented in hardware, software, firmware, or any combination thereof. If implemented in software, the functions may be stored as one or more instructions, operations, or

code on a non-transitory computer-readable medium or a non-transitory processor-readable medium. The operations of a method or algorithm disclosed herein may be embodied in a processor-executable software module comprising processor-executable instructions or operations that may reside on a non-transitory computer-readable or processor-readable storage medium. Non-transitory computer-readable or processor-readable storage media may be any storage media that may be accessed by a computer or a processor. By way of example but not limitation, such non-transitory computer-readable or processor-readable media may include RAM, ROM, EPROM, FLASH memory, CD-ROM or other optical disk storage, magnetic disk storage or other magnetic storage devices, or any other medium that may be used to store desired program code in the form of instructions, operations, or data structures and that may be accessed by a computer. Disk and disc, as used herein, includes compact disc (CD), laser disc, optical disc, digital versatile disc (DVD), floppy disk, and blue-ray disc where disks usually reproduce data magnetically; while discs reproduce data optically with lasers. Combinations of the above are also included within the scope of non-transitory computer-readable and processor-readable media. Additionally, the operations of a method or algorithm may reside as one or any combination of a set of codes, instructions, and/or operations on a non-transitory processor-readable medium and/or processor-readable medium, which may be incorporated into a computer program product.

The preceding description of the disclosed aspects is provided to enable any person skilled in the art to make or use the present invention. Various modifications to these aspects will be readily apparent to those skilled in the art, and the generic principles defined herein may be applied to other aspects without departing from the spirit or scope of the invention. Thus, the present invention is not intended to be limited to the aspects shown herein but is to be accorded the widest scope consistent with the following claims and the principles and novel features disclosed herein.

What is claimed is:

1. A method for increasing the processing speed of dynamic language software on a computing device, comprising:
 initializing a first inline cache for a first instance of a dynamic software operation by a processor;
 storing the first inline cache in a memory configured to provide fast access for storing and retrieving the first inline cache;
 receiving a second instance of the dynamic software operation in the processor;
 determining whether the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same;
 executing the second instance of the dynamic software operation by a functional unit using the first inline cache from the memory in response to determining that the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same; and
 returning a result of executing the second instance of the dynamic software operation by the functional unit.

2. The method of claim 1, further comprising:
 receiving an object type for the second instance of the dynamic software operation by the functional unit;
 receiving an object identifier for the second instance of the dynamic software operation by the functional unit; and

receiving inline cache data for the second instance of the dynamic software operation by the functional unit.
 3. The method of claim 2, wherein executing the second instance of the dynamic software operation by the functional unit using the first inline cache from the memory in response to determining that the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same comprises using the object type, the object identifier, and the inline cache data for the second instance of the dynamic software operation as parameters for executing the second instance of the dynamic software operation.

4. The method of claim 1, wherein determining whether the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same comprises:

comparing a first object implementation related to the first instance of the dynamic software operation with a second object implementation related to the second instance of the dynamic software operation; and determining whether the first object implementation and the second object implementation are the same.

5. The method of claim 1, further comprising:
 initializing a second inline cache for the second instance of the dynamic software operation including the first inline cache configured to replace the initialized first inline cache in response to determining that the first instance of the dynamic software operation and the second instance of the dynamic software operation are different.

storing the second inline cache in the memory configured to provide fast access for storing and retrieving the second inline cache; and

executing the second instance of the dynamic software operation by the functional unit using the second inline cache from the memory in response to determining that the first instance of the dynamic software operation and the second instance of the dynamic software operation are different.

6. The method of claim 1, further comprising:
 determining whether the first inline cache exists for the first instance of the dynamic software operation, and wherein initializing the first inline cache for the first instance of the dynamic software operation by the processor comprises initializing the first inline cache for the first instance of the dynamic software operation by the processor in response to determining that the first inline cache for the first instance of the dynamic software operation does not exist.

7. The method of claim 1, wherein initializing the first inline cache for the first instance of the dynamic software operation by the processor comprises:

traversing an object implementation for the dynamic software operation until identifying a data of the object implementation relating to the first instance of the dynamic software operation.

executing the dynamic software operation at the first instance of the dynamic software operation; and returning a result of the first instance of the dynamic software operation.

8. The method of claim 1, wherein returning the result of executing the second instance of the dynamic software operation by the functional unit comprises returning the result directly to the processor.

9. The method of claim 1, wherein returning the result of executing the second instance of the dynamic software operation by the functional unit comprises returning the

result to the processor indirectly through a data cache accessible to the processor and the functional unit;

19. The method of claim 1, further comprising:

- generating software instructions for the processor by a computer; and
- instructing the processor to run the functional unit to perform operations comprising:
- initializing the first inline cache for the first instance of the dynamic software operation by a processor;
- storing the first inline cache in the memory configured to provide last access for storing and retrieving the first inline cache;
- receiving the second instance of the dynamic software operation in the processor;
- determining whether the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same;
- executing the second instance of the dynamic software operation by the functional unit using the first inline cache from the memory in response to determining that the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same; and
- returning the result of executing the second instance of the dynamic software operation by the functional unit;

21. A computing device, comprising:

- a memory configured to provide last access for storing and retrieving at least one inline cache;
- a processor communicatively connected to the memory and configured with processor-executable instructions to perform operations comprising:
- initializing a first inline cache for a first instance of a dynamic software operation;
- storing the first inline cache in the memory;
- receiving a second instance of the dynamic software operation;
- determining whether the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same;
- a functional unit communicatively connected to the processor and the memory and configured with functional unit-executable instructions to perform operations comprising:
- executing the second instance of the dynamic software operation using the first inline cache from the memory in response to determining that the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same; and
- returning a result of executing the second instance of the dynamic software operation to the processor;

22. The computing device of claim 21, wherein the functional unit is further configured with functional unit-executable instructions to perform operations comprising:

- receiving an object type for the second instance of the dynamic software operation;
- receiving an object identifier for the second instance of the dynamic software operation; and
- receiving inline cache data for the second instance of the dynamic software operation.

23. The computing device of claim 22, wherein the functional unit is further configured with functional unit-executable instructions to perform operations such that using the first inline cache from the memory in response to determining that the first instance of the dynamic software operation and the second instance of the dynamic software

operation are the same comprises using the object type, the object identifier, and the inline cache data for the second instance of the dynamic software operation as parameters for executing the second instance of the dynamic software operation.

24. The computing device of claim 21, wherein the processor is further configured with processor-executable instructions to perform operations such that determining whether the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same comprises:

comparing a first object implementation related to the first instance of the dynamic software operation with a second object implementation related to the second instance of the dynamic software operation; and

25. determining whether the first object implementation and the second object implementation are the same;

26. The computing device of claim 21, wherein:

the processor is further configured with processor-executable instructions to perform operations comprising:

- initializing a second inline cache for the second instance of the dynamic software operation including the first inline cache configured to replace the initialized first inline cache in response to determining that the first instance of the dynamic software operation and the second instance of the dynamic software operation are different; and
- storing the second inline cache in the memory; and
- the functional unit is further configured with functional unit-executable instructions to perform operations comprising executing the second instance of the dynamic software operation using the second inline cache from the memory in response to determining that the first instance of the dynamic software operation and the second instance of the dynamic software operation are different;

27. The computing device of claim 21, wherein the processor is further configured with processor-executable instructions to perform operations comprising determining whether the first inline cache exists for the first instance of the dynamic software operation, and such that initializing the first inline cache for the first instance of the dynamic software operation comprises:

initializing the first inline cache for the first instance of the dynamic software operation in response to determining that the first inline cache for the first instance of the dynamic software operation does not exist;

28. traversing an object implementation for the dynamic software operation until identifying a data of the object implementation related to the first instance of the dynamic software operation;

executing the dynamic software operation of the first instance of the dynamic software operation; and

29. returning a result of the first instance of the dynamic software operation.

30. The computing device of claim 21, wherein the processor is further configured with processor-executable instructions to perform operations comprising:

- generating software instructions by a computer; and
- instructing the functional unit to perform operations comprising:
- initializing the first inline cache for the first instance of the dynamic software operation;
- storing the first inline cache in the memory;
- receiving the second instance of the dynamic software operation;

- determining whether the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same;
executing the second instance of the dynamic software operation using the fast inline cache from the memory in response to determining that the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same; and
removing the result of executing the second instance of the dynamic software operation to the processor.
- 18. A non-transitory computer-readable medium having stored thereon processor-executable software instructions to cause a processor and a functional unit to perform operations comprising:**
- initializing a first inline cache for a first instance of a dynamic software operation by a processor;**
 - storing the first inline cache in a memory configured to provide fast access for storing and retrieving the first inline cache;**
 - receiving a second instance of the dynamic software operation in the processor;**
 - determining whether the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same;**
 - executing the second instance of the dynamic software operation by the functional unit using the first inline cache from the memory in response to determining that the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same; and**
 - returning a result of executing the second instance of the dynamic software operation by the functional unit.**
- 19. The non-transitory processor-readable medium of claim 18, wherein the stored processor-executable software instructions are configured to cause the processor and the functional unit to perform operations comprising:**
- receiving an object type for the second instance of the dynamic software operation by the functional unit;**
 - receiving an object identifier for the second instance of the dynamic software operation by the functional unit;**
 - and**
 - receiving inline cache data for the second instance of the dynamic software operation by the functional unit.**
- 20. The non-transitory processor-readable medium of claim 19, wherein the stored processor-executable software instructions are configured to cause the processor and the functional unit to perform operations such that executing the second instance of the dynamic software operation by the functional unit using the first inline cache from the memory in response to determining that the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same comprises using the object type, the object identifier, and the inline cache data for the second instance of the dynamic software operation as parameters for executing the second instance of the dynamic software operation.**
- 21. The non-transitory processor-readable medium of claim 18, wherein the stored processor-executable software instructions are configured to cause the processor and the functional unit to perform operations such that determining whether the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same comprises:**
- comparing a first object implementation related to the first instance of the dynamic software operation with a second object implementation related to the second instance of the dynamic software operation; and**
 - determining whether the first object implementation and the second object implementation are the same.**
- 22. The non-transitory processor-readable medium of claim 18, wherein the stored processor-executable software instructions are configured to cause the processor and the functional unit to perform operations comprising:**
- initializing a second inline cache for the second instance of the dynamic software operation including the first inline cache configured to replace the initialized first inline cache in response to determining that the first instance of the dynamic software operation and the second instance of the dynamic software operation are different;**
 - storing the second inline cache in the memory configured to provide fast access for storing and retrieving the second inline cache; and**
 - executing the second instance of the dynamic software operation by the functional unit using the second inline cache from the memory in response to determining that the first instance of the dynamic software operation and the second instance of the dynamic software operation are different.**
- 23. The non-transitory processor-readable medium of claim 18, wherein the stored processor-executable software instructions are configured to cause the processor and the functional unit to perform operations comprising determining whether the first inline cache exists for the first instance of the dynamic software operation, and such that initializing the first inline cache for the first instance of the dynamic software operation by the processor comprises:**
- initializing the first inline cache for the first instance of the dynamic software operation by the processor in response to determining that the first inline cache for the first instance of the dynamic software operation does not exist;**
 - traversing an object implementation for the dynamic software operation until identifying a data of the object implementation relating to the first instance of the dynamic software operation;**
 - executing the dynamic software operation of the first instance of the dynamic software operation; and**
 - returning a result of the first instance of the dynamic software operation.**
- 24. The non-transitory processor-readable medium of claim 18, wherein the stored processor-executable software instructions are configured to cause the processor and the functional unit to perform operations comprising:**
- generating software instructions for the processor by a compiler; and**
 - instructing the processor to run the functional unit to perform operations comprising:**
 - initializing the first inline cache for the first instance of the dynamic software operation;**
 - storing the first inline cache in the memory;**
 - receiving the second instance of the dynamic software operation;**
 - determining whether the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same;**
 - executing the second instance of the dynamic software operation using the first inline cache from the memory in response to determining that the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same; and**

23

returning the result of executing the second instance of the dynamic software operation to the processor.

25. A computing device, comprising:
 means for initializing a first inline cache for a first instance of a dynamic software operation;
 means for storing the first inline cache configured to provide fast access for storing and retrieving the first inline cache;
 means for receiving a second instance of the dynamic software operation;
 means for determining whether the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same;
 means for executing the second instance of the dynamic software operation using the first inline cache from means for storing the first inline cache in response to determining that the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same; and
 means for returning a result of executing the second instance of the dynamic software operation.

26. The computing device of claim 25, further comprising:
 means for receiving an object type for the second instance of the dynamic software operation;
 means for receiving an object identifier for the second instance of the dynamic software operation; and
 means for receiving inline cache data for the second instance of the dynamic software operation.
 27. The computing device of claim 26, wherein means for executing the second instance of the dynamic software operation using the first inline cache from the means for storing the first inline cache in response to determining that the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same comprises means for using the object type, the object identifier, and the inline cache data for the second instance of the dynamic software operation as parameters for executing the second instance of the dynamic software operation.

28. The computing device of claim 25, wherein means for determining whether the first instance of the dynamic software operation and the second instance of the dynamic software operation are the same comprises:
 means for comparing a first object implementation related to the first instance of the dynamic software operation

24

with a second object implementation related to the second instance of the dynamic software operation; and means for determining whether the first object implementation and the second object implementation are the same.

29. The computing device of claim 25, further comprising:
 means for initializing a second inline cache for the second instance of the dynamic software operation including the first inline cache configured to replace the initialized first inline cache in response to determining that the first instance of the dynamic software operation and the second instance of the dynamic software operation are different;
 means for storing the second inline cache configured to provide fast access for storing and retrieving the second inline cache; and
 means for executing the second instance of the dynamic software operation using the second inline cache from means for storing the second inline cache in response to determining that the first instance of the dynamic software operation and the second instance of the dynamic software operation are different.

30. The computing device of claim 25, further comprising:
 means for determining whether the first inline cache exists for the first instance of the dynamic software operation; and
 wherein means for initializing the first inline cache for the first instance of the dynamic software operation comprises:
 means for initializing the first inline cache for the first instance of the dynamic software operation in response to determining that the first inline cache for the first instance of the dynamic software operation does not exist;
 means for traversing an object implementation for the dynamic software operation until identifying a data of the object implementation relating to the first instance of the dynamic software operation;
 means for executing the dynamic software operation of the first instance of the dynamic software operation; and
 means for returning a result of the first instance of the dynamic software operation.

< < < < >



US09632569B2

**(12) United States Patent
Suarez Gracia et al.****(10) Patent No.: US 9,632,569 B2
(11) Date of Patent: Apr. 25, 2017****(54) DIRECTED EVENT SIGNALING FOR
MULTIPROCESSOR SYSTEMS**(71) Applicant: **QUALCOMM Incorporated**, San Diego, CA (US)(72) Inventors: **Darin Suarez Gracia**, Santa Clara, CA (US); **Han Zhou**, Sunnyvale, CA (US); **Pablo Montesinos Oregón**, Fremont, CA (US); **Gheorghe Calin Cusecavil**, Palo Alto, CA (US); **James Xenidis**, Cedar Park, TX (US)(73) Assignee: **QUALCOMM Incorporated**, San Diego, CA (US)

(19) (1) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 291 days.

(21) Appl. No. 14/481,628

(22) Filed: Aug. 5, 2014

(65) Prior Publication Data

US 2016/0041852 A1 — Feb. 11, 2016

(51) Int. Cl.**G06F 9/30** (2006.01)
G06F 1/32 (2006.01)
G06F 9/32 (2006.01)
G06F 9/48 (2006.01)**(52) U.S. Cl.****CPC G06F 1/3296** (2013.01); **G06F 9/486** (2013.01); **G06F 9/489** (2013.01); **G06F 9/26** (2013.01); **362K 60/144** (2013.01)**(58) Field of Classification Search**

None

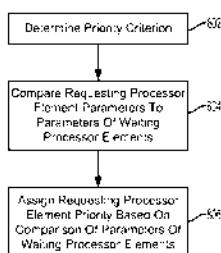
See application file for complete search history.

(56) References Cited**U.S. PATENT DOCUMENTS**6,058,114 A 5,355,000 Munkindalsom et al.
8,578,079 B2 11,391,13 De Cesare et al.
(Continued)**FOREIGN PATENT DOCUMENTS**

EP 707,309 B2 — 6-2009

OTHER PUBLICATIONSInternational Search Report and Written Opinion — PCT/US2015/02036; ISA (PPO) — Mar. 1, 2016
(Continued)*Primary Examiner*: William B. Bratridge
(43) *Attorney, Agent, or Firm*: The Marbury Law Group, PLLC**(57) ABSTRACT**

Multi processor computing device methods, manage resource access by a signaling event manager signaling processor elements requesting access to a resource to wake up to access the resource when the resource is available or wait for an event when the resource is busy. Processor elements may enter a sleep state while awaiting access to the requested resource. When multiple elements are waiting for the resource, the processor element with a highest assigned priority is signaled to wake up when the resource is available without waking other elements. Priorities may be assigned to processor elements waiting for the resource based on a heuristic or parameter that may depend on a state of the computing device or the processor elements. A sleep duration may be estimated for a processor element waiting for a resource and the processor element may be removed from a scheduling queue or assigned to another thread during the sleep duration.

12 Claims, 14 Drawing Sheets

US 9,632,569 B2

Page 2

- (56) **References Cited**
- U.S. PATENT DOCUMENTS**
- 2004/013712 AL* 7 2004 Armstrong 6561 9-544
2011/016159 AL* 6 2011 Anand 6561 9-526
2011/016150 AL 6 2011 Cheng et al.
2013/0081005 AL 3 2013 Ocioyo et al.
2014/004994 AL 2 2014 Onisto et al.
- 711 152
710 290
- OTHER PUBLICATIONS**
- Swaminathan, et al., "Energy-Conscious Deterministic Task Scheduling in Hard Real-Time Systems," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, IEEE Service Center, Piscataway, NJ, US, Jul. 1, 2003, vol. 22, No. 7, pp. 847-858, XP002532583, ISSN: 0278-0070, DOI: 10.1109/TCAD.2003.813415 [retrieved on Jul. 1, 2003].
- Muller, F., "Prioritized token-based mutual exclusion for distributed systems", Parallel and Distributed Real-Time Systems, 1997. Proceedings of the Joint Workshop on Geneva, Switzerland Apr. 1-3, 1997, Los Alamitos, CA, USA, IEEE Comput. Soc, US Apr. 1, 1997, pp. 72-80, XP01028193, DOI: 10.1109/WPDRTS.1997.647866 ISBN: 0780338688096, abstract: Section 1: Introduction, p. 72, p. 73, right-hand column; line 2-11 p. 73, right-hand column, line 3-22 p. 76, right-hand column, line 19, figure 6.
- Partial International Search Report, 19, IUS2015-042026, ISA IPO, Nov. 10, 2015.
- Suleiman M.A., et al., "Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures," Mar. 7, 2009 (Mar. 7, 2009), Proceedings of the 14th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Washington DC, Mar. 2009, pp. 1-12, X0972008111.
- Jiang J.R., "A prioritized busy-skip mutual exclusion algorithm with minimum degree of concurrency for mobile ad hoc networks and distributed systems", Parallel & Distributed Computing, Applications & Technologies, 2003. PDCAT. 2003. Proceedings of the 10th International Conference on Aug. 27-29, 2003, Piscataway, NJ, US, Aug. 27, 2003, pp. 329-334.
- Brändenborg, B.B., "Scheduling and Locking in Multiprocessor Real-Time Operating Systems," Chapel Hill dissertation submitted for the year of 2001, pp. 1-614.
- * cited by examiner

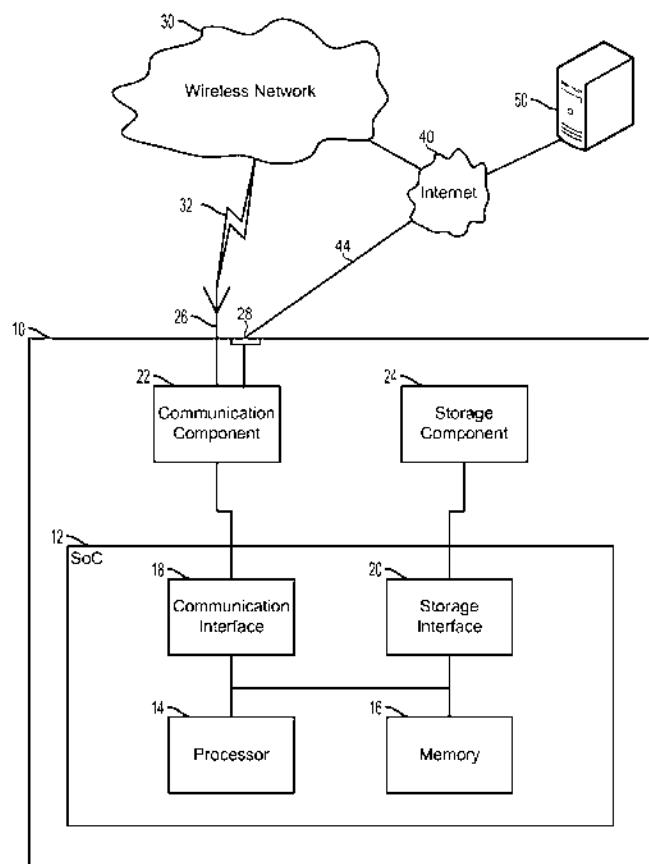


FIG. 1

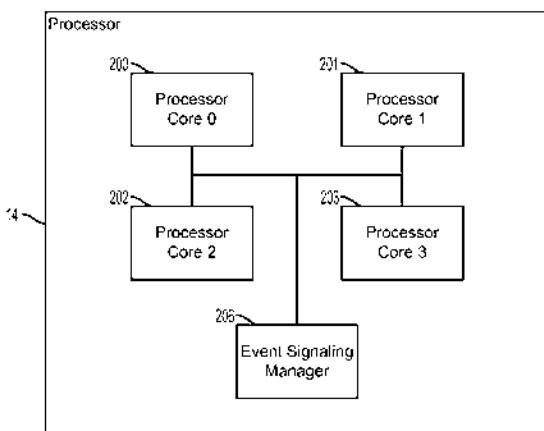


FIG. 2A

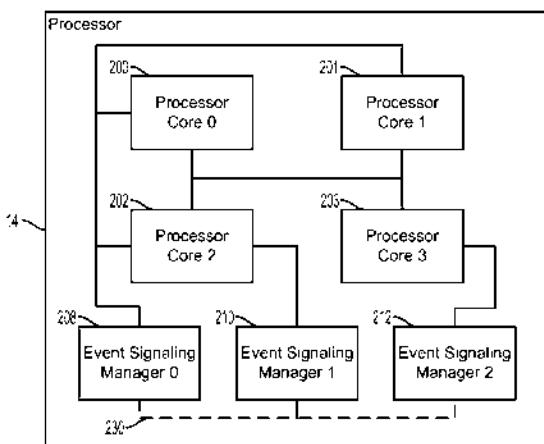


FIG. 2B

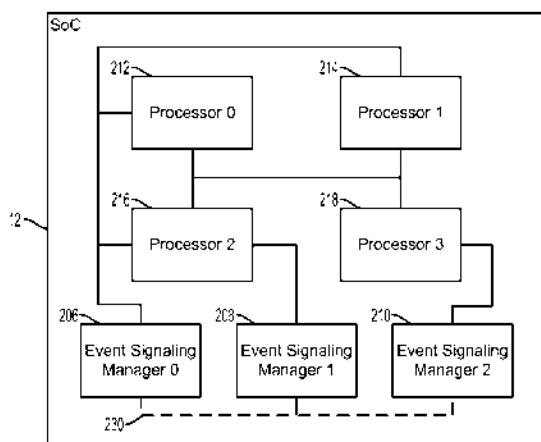


FIG. 2C

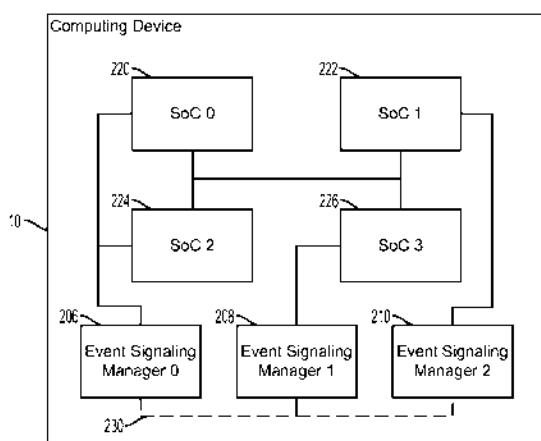


FIG. 2D

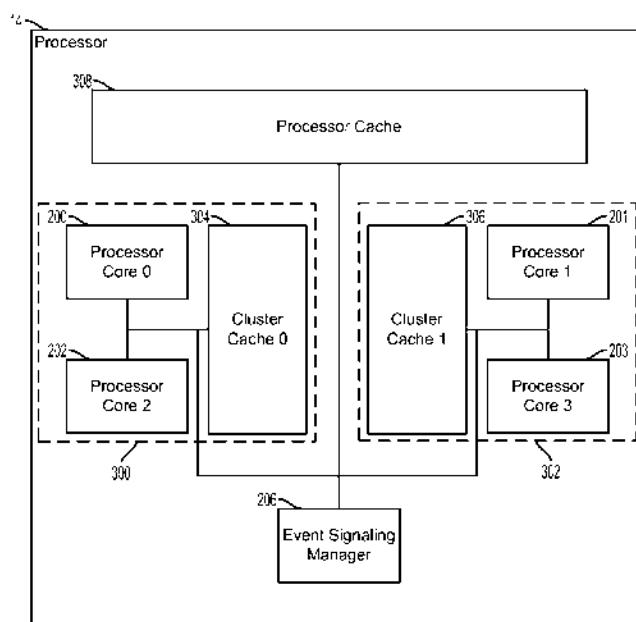


FIG. 3

Processor Element	Priority	Element Temperature	Number of Element Wake Ups	Time Since Last Element Wake Up
Processor Element 0	3	31.9°C	253	5.2ms
Processor Element 1	5	35.6°C	425	3.6ms
Processor Element 2	1	28.1°C	356	10.1ms
Processor Element 3	8	38.2°C	227	1.7ms
:	:	:	:	:
Processor Element N-1	2	31.4°C	314	8.8ms

FIG. 4

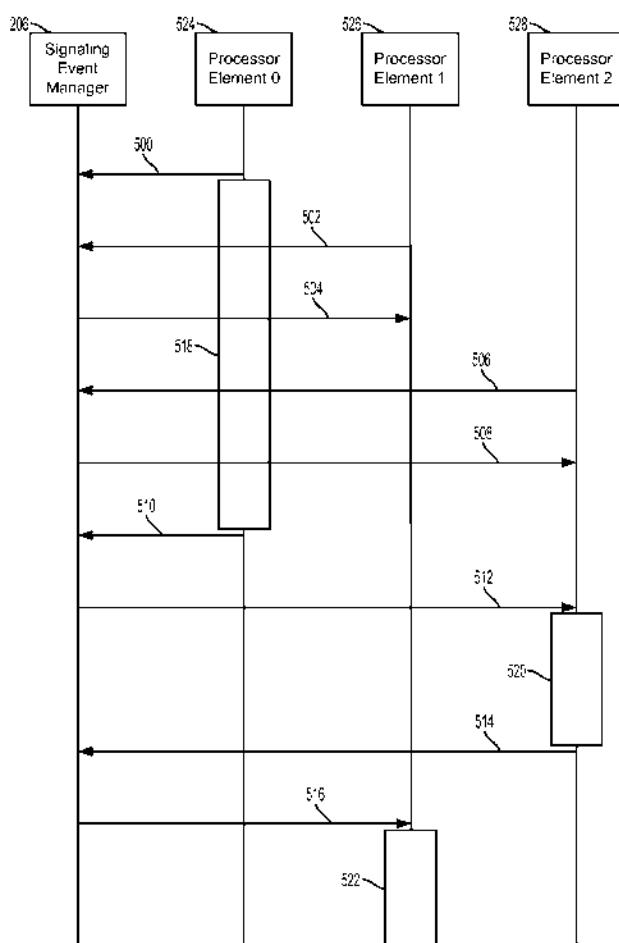


FIG. 5

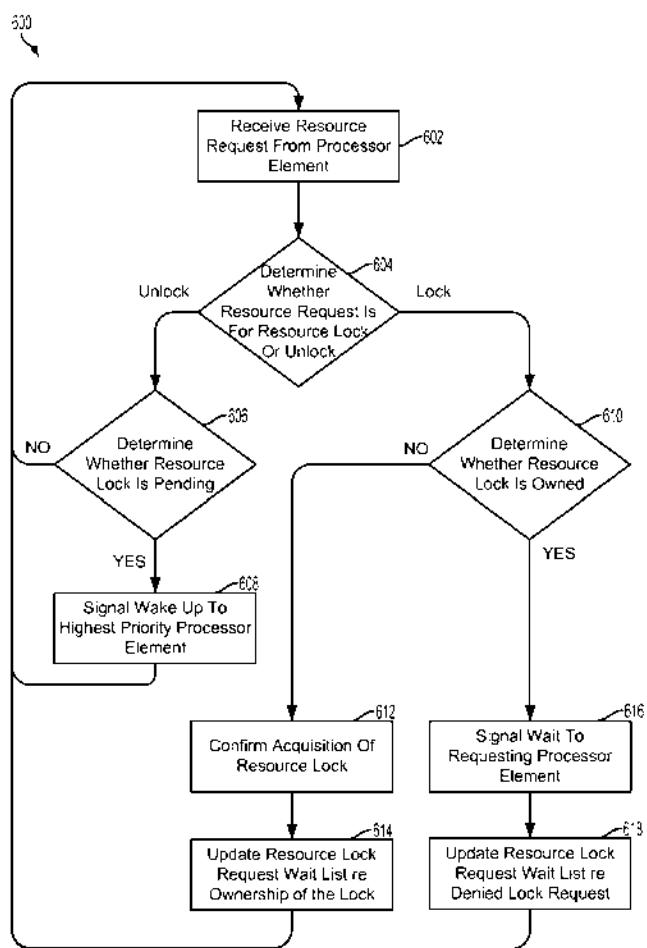


FIG. 6

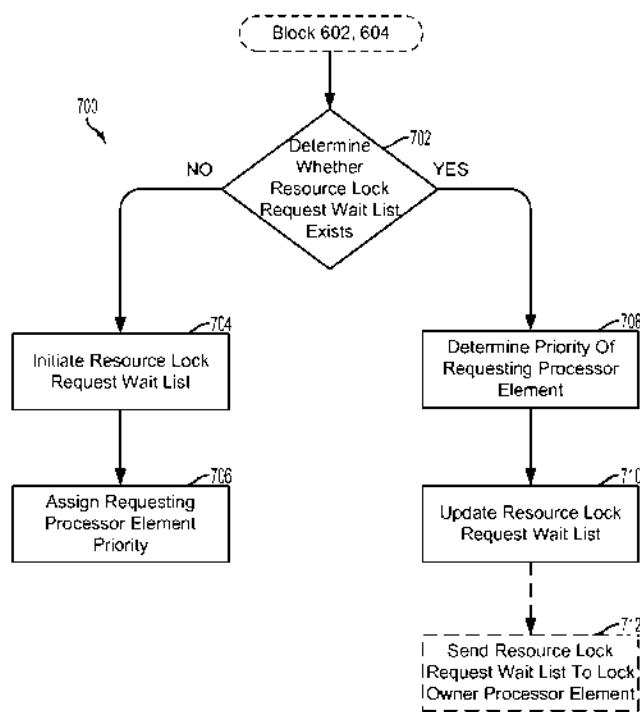


FIG. 7

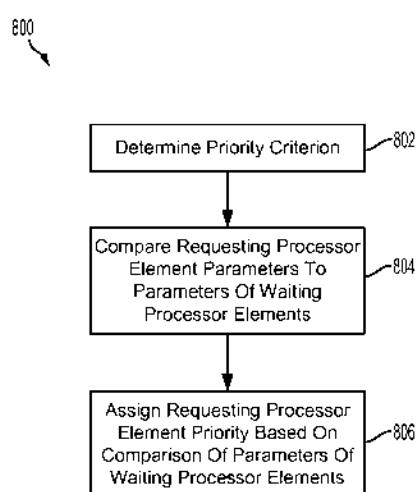


FIG. 8

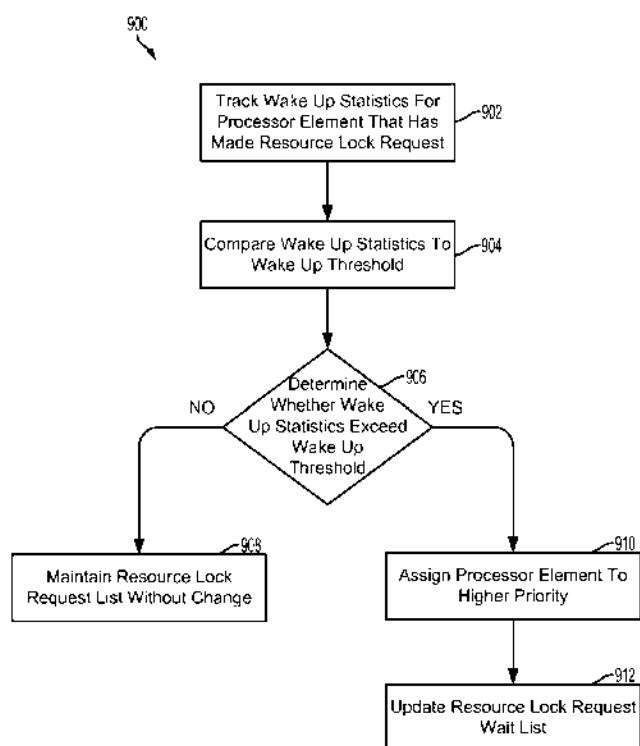


FIG. 9

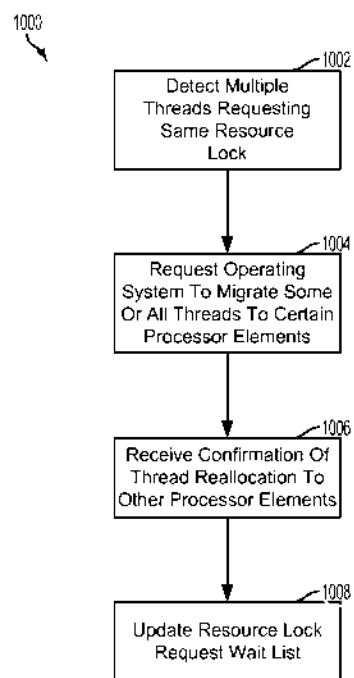


FIG. 10

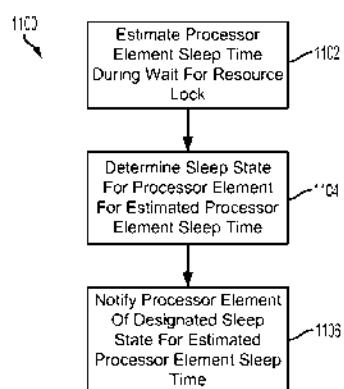


FIG. 11

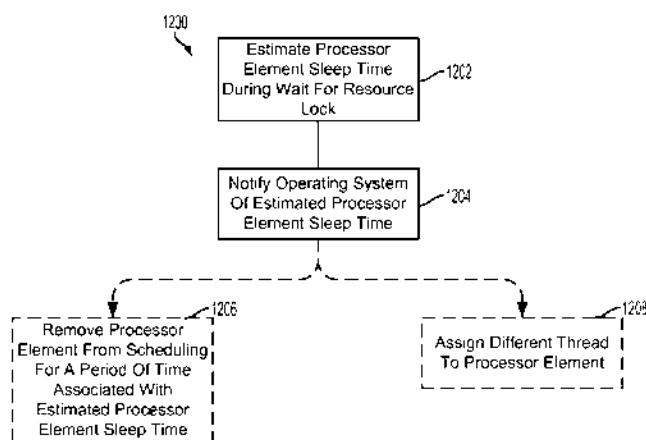


FIG. 12

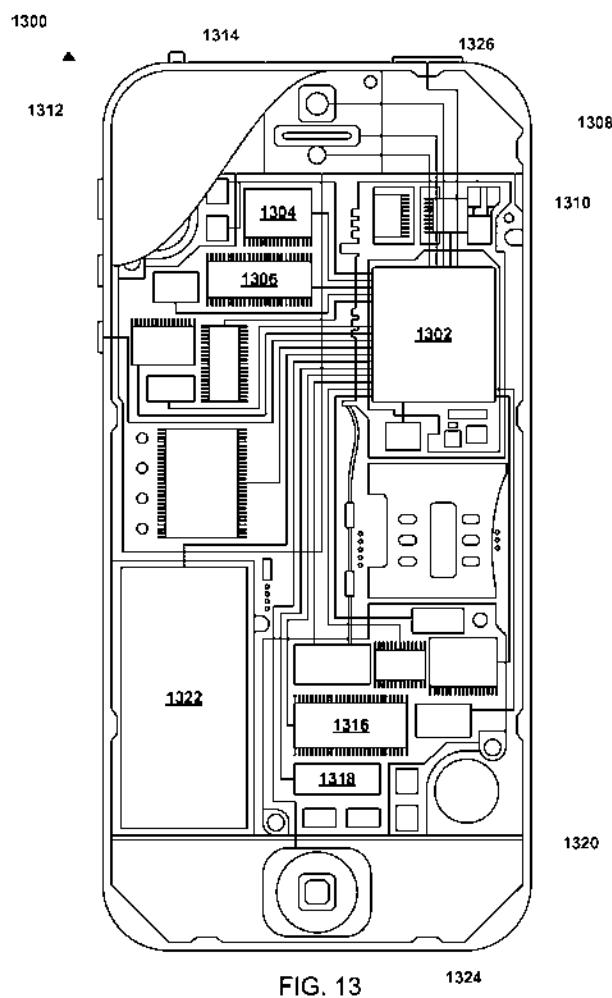


FIG. 13

1324

U.S. Patent

Apr. 25, 2017

Sheet 14 of 14

US 9,632,569 B2

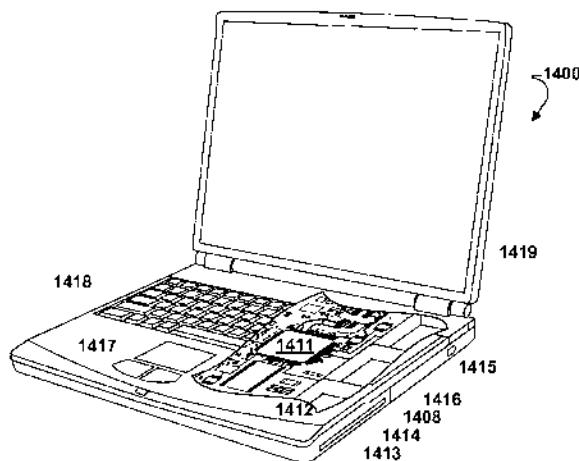


FIG. 14

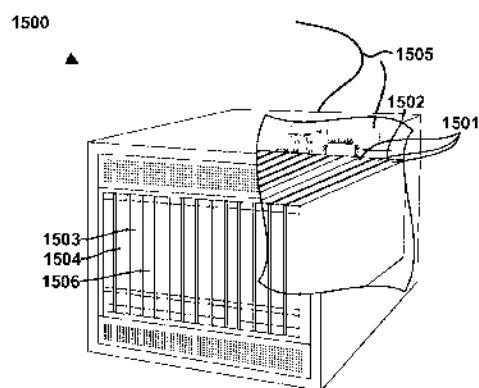


FIG. 15

DIRECTED EVENT SIGNALING FOR MULTIPROCESSOR SYSTEMS

BACKGROUND

Multiprocessors rely on atomicity to guarantee correctness of parallel applications. To prevent other processes from interrupting the execution of atomic operations, one implemented solution is for the processes to obtain locks on the resources needed to execute the critical section in mutual-exclusion so that other processes are prevented from using the same resources until the resources are released from the locks. An example of one such lock is a spinlock. A spinlock is an atomic operation that actively waits until the lock is acquired, repeatedly checking whether the lock is available. Since the spinlock process remains active but is not performing a useful task, this active wait operation consumes energy because the process is continuously loading the lock from memory. Other locks, or other resource acquisition schemes, may similarly be associated with some cost to the efficiency and performance of the implementing device. To save energy, some processor architectures have instructions to wait for events and to signal events (e.g., ARM wait-for-event ("WFE") set event ("SEV"))¹. These architectures employ these instructions in an indiscriminate manner, where the instructions are broadcast to all waiting processor cores and create a race condition between the processor cores to claim an available lock for executing the critical sections.

When signaling a processor core with a wake up instruction to execute an atomic process, some operating systems, such as Linux, reorder the wake up in software to avoid cache ping-pong issues. In these instances operating system level handoff may require kernel activity and context switches. This may result in reduced performance and increased resource usage in an architecture employing the above described indiscriminate instruction signaling in which multiple processor cores would be signaled to wake up to obtain an available lock when only one core will be able to use the resource.

SUMMARY

The methods and apparatuses of various embodiments provide circuits and methods for directed event signaling for multiprocessor systems. Embodiment methods may include: signaling a processor element requesting access to a resource to wait for an event in response to determining that the resource is not available; and signaling the processor element to access the resource in response to the resource becoming available. Embodiments may include assigning a priority to the processor element for accessing the resource in response to the resource's availability; receiving a signal indicating availability of the resource; and identifying one of the plurality of processor elements that is assigned a highest priority for accessing the resource in response to the signal indicating availability of the resource; in which signaling the processor element to access the resource in response to the resource becoming available may include: signaling the processor element that is assigned the highest priority for accessing the resource to access the resource in response to the resource becoming available. In an embodiment, signaling a processor element requesting access to a resource to wait for an event in response to determining that the resource is not available may include: triggering the processor element to enter a sleep state; and signaling the processor element to

access the resource in response to the resource becoming available may include signaling the processor element to wake up.

In an embodiment assigning a priority to the processor element for accessing the resource may include: receiving a parameter for use in calculating the priority of the processor element; comparing the parameter of the processor element with parameters for any of the plurality of processor elements that waiting to access the resource; determining the priority of the processor element for accessing the resource based on the comparison; and storing the determined priority of the processor element in memory. In an embodiment, receiving a parameter for use in calculating the priority of the processor element may include: signaling the processor element to return the parameter for use in calculating the priority of the processor element; receiving the parameter from the processor element; and storing the parameter in memory. An embodiment may include: determining the parameter for use in calculating the priority of the processor element based on a state of the computing device; in which signaling the processor element to return the parameter for use in calculating the priority of the processor element may include: signaling the processor element to return the determined parameter.

In an embodiment, receiving a parameter for use in calculating the priority of the processor element may include: receiving a temperature related to the processor element. In an embodiment, receiving the parameter for use in calculating the priority of the processor element may include: receiving a frequency related to the processor element. In an embodiment, receiving the parameter for use in calculating the priority of the processor element may include: receiving an indication of a cache shared by the processor element with another processor element. In an embodiment, receiving the parameter for use in calculating the priority of the processor element may include: receiving an indication of proximity of the processor element to another processor element.

In an embodiment, assigning a priority to the processor element for accessing the resource in response to its availability may include: tracking a wake up statistic for the processor element; determining whether the wake up statistic for the processor element exceeds wake up threshold; and assigning a high priority to the processor element for accessing the resource in response to determining that the wake up statistic for the processor element exceeds the wake up threshold.

An embodiment method may include: sending the priority assigned to the processor element to another processor element currently accessing the resource; and in which signaling the processor element to access the resource in response to the resource becoming available may include: signaling the processor element by the another processor element currently accessing the resource upon releasing the resource. An embodiment method may include: detecting a plurality of threads requesting access to the resource; and requesting an operating system to migrate the plurality of threads to a selected group of the plurality of processor elements.

An embodiment may include: estimating a sleep time for the processor element while waiting for the resource; determining a sleep state for the processor element based on the estimated sleep time; and notifying the processor element of the sleep state to implement while waiting for the resource. An embodiment method may include: estimating a sleep time for the processor element while waiting for the resource; and notifying an operating system

US 9,632,569 B2

3

of the computing device of the estimated sleep time for the processor element. An embodiment method may include removing the processor element from a scheduling queue for a period of time associated with the estimated sleep time for the processor element. An embodiment method may include assigning a thread to the processor element during the estimated sleep time. An embodiment method may include communicating between signaling event managers to manage signaling of the plurality of processor elements.

An embodiment includes a computing device having a plurality of processor elements and a signaling event manager communicatively connected to each other, and the signaling event manager configured with signaling event manager-executable instructions to perform operations of one or more of the embodiment methods described above.

An embodiment includes a non-transitory processor-readable medium having stored thereon processor-executable software instructions to cause a processor to perform operations of one or more of the embodiment methods described above.

An embodiment includes a computing device having means for performing functions of one or more of the embodiment methods described above.

BRIEF DESCRIPTION OF THE DRAWINGS

The accompanying drawings, which are incorporated herein and constitute part of this specification, illustrate example embodiments of the invention, and together with the general description given above and the detailed description given below, serve to explain the features of the invention.

FIG. 1 is a component block diagram illustrating a computing device suitable for implementing an embodiment.

FIGS. 2-3D are component block diagrams illustrating example multi-core processors and SoCs including at least one signaling event manager suitable for implementing an embodiment.

FIG. 3 is a component block illustrating an example multi-core processor having a cluster architecture and including an signaling event manager suitable for implementing an embodiment.

FIG. 4 is an example table illustrating a relationship between processor elements and their assigned priorities in accordance with an embodiment.

FIG. 5 is a process flow and signaling diagram illustrating directed event signaling in a multiprocessor system in accordance with an embodiment.

FIG. 6 is a process flow diagram illustrating an embodiment method for implementing illustrating directed event signaling in a multiprocessor system.

FIG. 7 is a process flow diagram illustrating an embodiment method for implementing a lock resource access request wait list for directed event signaling in a multiprocessor system.

FIG. 8 is a process flow diagram illustrating an embodiment method for implementing prioritization of processor elements for directed event signaling in a multiprocessor system.

FIG. 9 is a process flow diagram illustrating an embodiment method for implementing starvation avoidance for directed event signaling in a multiprocessor system.

FIG. 10 is a process flow diagram illustrating an embodiment method for implementing lock-aware scheduling with directed event signaling in a multiprocessor system.

4

FIG. 11 is a process flow diagram illustrating an embodiment method for implementing lock stall actions with directed event signaling in a multiprocessor system.

FIG. 12 is a process flow diagram illustrating an embodiment method for implementing lock stall actions with directed event signaling in a multiprocessor system.

FIG. 13 is a component block diagram illustrating an example mobile computing device suitable for use with the various embodiments.

FIG. 14 is a component block diagram illustrating an example mobile computing device suitable for use with the various embodiments.

FIG. 15 is a component block diagram illustrating an example server suitable for use with the various embodiments.

DETAILED DESCRIPTION

The various embodiments will be described in detail with reference to the accompanying drawings. Wherever possible, the same reference numbers will be used throughout the drawings to refer to the same or like parts. References made to particular examples and implementations are for illustrative purposes, and are not intended to limit the scope of the invention or the claims.

The terms "computing device" and "mobile computing device" are used interchangeably herein to refer to any one or all of cellular telephones, smartphones, personal or mobile multi-media players, personal data assistants (PDAs), laptop computers, tablet computers, smartphones, ultrabooks, palm-top computers, wireless electronic mail receivers, multimedia Internet enabled cellular telephones, wireless gaming controllers, and similar personal electronic devices that include a memory, and a programmable multiprocessor (e.g., multi-core, multi-socket, or multi-CPU). While the various embodiments are particularly useful for mobile computing devices, such as smartphones, which have limited memory and battery resources, the embodiments are generally useful in any electronic device that implements a parity of memory devices and a limited power budget in which reducing the power consumption of the processors can extend the battery operating time of the mobile computing device.

The term "system-on-chip" (SoC) is used herein to refer to a set of interconnected electronic circuits typically, but not exclusively, including a hardware core, a memory, and a communication interface. A hardware core may include a variety of different types of processor elements, such as a general purpose processor, a central processing unit (CPU), a digital signal processor (DSP), a graphics processing unit (GPU), an accelerated processing unit (APU), an auxiliary processor, a single-core processor, a multi-core processor, and individual processor cores. A hardware core may further embody other hardware and hardware combinations, such as a field programmable gate array (FPGA), an application-specific integrated circuit (ASIC), other programmable logic device, discrete gate logic, transistor logic, performance monitoring hardware, watchdog hardware, and timer references. Integrated circuits may be configured such that the components of the integrated circuit reside on a single piece of semiconductor material, such as silicon.

Embodiments include methods and computing devices implementing such methods for dynamically changing the time at which each processor element in a multiprocessor element system receives a signal by using system state information. A signaling event manager may be implemented in hardware and/or software for managing the event

and wake up signaling to each of the processor elements by ordering events based on system and environment states. One or more signaling event managers may compute heuristic-based notification priorities and send notifications of events (e.g., wake up and wait signals) to individual processor elements in an order determined by the priorities determined from a heuristic. The heuristic for determining processor element priorities that is applied in response to a particular signaling event may be selected based on the signaling event, and may be supplied to the signaling event manager by being directly encoded in a received signaling event instruction, read from a register or memory location associated with the signaling event, or predefined and stored in the signaling event manager for different signaling modes or events. In an embodiment, a heuristic may be used to determine the order of a signaling priority for the processor elements based on temperatures of the processor elements (from low to high) to reduce the chance of thermal throttling. In an embodiment, the heuristic may be based on operating frequency of the elements (from high to low) to minimize execution time. In an embodiment, the heuristic may be based on sharing cache or other resources, such as accelerators or functional units, between certain processor elements and proximity of the processor elements (preferring a local processor element than a remote processor elements).

Using one or more of the embodiment heuristics for determining the priority of the processor elements, the signaling event manager may create and manage a list of processor elements and their priorities. The list may be used to send out event notifications to the processor elements signaling to one of the processor elements (i.e., all processor cores as in the prior art) the availability of a resource and to accept verify requests for a resource from all processor elements. The list may be managed such that the priority of the processor elements may be rearranged based on the heuristic factors described above.

In an embodiment, when a processor element receives an event, the processor element may wake up and try to grab a lock. The signaling event manager may receive a request from the processor element to grab a lock for a resource and identify the priority of the processor element. If a lock is available and the processor element has the highest priority, the signaling event manager may allow the processor element to obtain the available lock. However, if locks on the resource are unavailable or the processor element does not have the highest priority, the signaling event manager may instruct the processor element to wait upon a lock becoming available and the signaling event manager identifying a highest priority processor element, the signaling event manager may signal the highest priority processor element to wake up and request to obtain the lock. When a processor element releases a lock and the lock becomes available, the signaling event manager may send a signal to a next waiting processor element according to a previous priority ordering of the processor elements, or the signaling event manager may update the list of waiting processor elements and send the signal to a new highest priority waiting processor element.

The heuristic used by the signaling event manager to determine the priorities of the processor elements may be provided by the processor elements. In an embodiment, the processor elements may include a wait-for-event ("WFE") instruction that may be executed when requesting a resource lock. The wait-for-event instruction may list the heuristic information that is desired for determining the priority of the processor elements. Executing the wait-for-event instruction may cause the processor elements to forward the heuristic

information along with the request for the resource lock. For example, the processor element may return thermal information in response to an indication of a thermal-based heuristic, cache locality information in response to an indication of a location-based heuristic, power state information in response to an indication of a frequency-based heuristic. A processor element may provide one or more of these information types to the signaling event manager in the request to grab the lock. The heuristic information type sent may be predetermined by a type of system implementing the signaling event manager. In an embodiment, the signaling event manager may respond to the request for the resource lock and heuristic information with a WFE signal, which may trigger the requesting processor element to enter a sleep low power state.

In an embodiment in which the processor elements do not include a WFE instruction and the heuristic information type is indicated by the signaling event manager-selected or in which the signaling event manager changes the heuristic information that it requires to determine priority, the type of heuristic information requested may be included in the WFE signal sent to the processor elements. Bits in the WFE signal may prompt the processor element to respond to the signaling event manager with state information relevant to a heuristic so that the signaling event manager may apply that information to the heuristic in order to determine the priority of the processor element. As above, examples may include the processor element returning one or more of thermal information in response to an indication of a thermal-based heuristic, cache locality information in response to an indication of a location-based heuristic, power state information in response to an indication of a frequency-based heuristic in response to an indication of a signaling event manager-selected heuristic. The wait-for-event signal may also trigger the sleep state in the requesting processor elements.

In an embodiment, the heuristic information for the requesting processor elements may be determined, rather than retrieved, by the signaling event manager in response to resource lock requests. The signaling event manager may still send the wait-for-event signal to the requesting processor elements to trigger the sleep state in the requesting processor elements.

The signaling event manager may keep track of the number of wake ups and/or when the wake ups last occurred for each processor element. The signaling event manager may compare this data to a starvation threshold to prioritize processor elements so that all processor elements eventually receive the lock. For example, when a processor element exceeds the threshold indicating that the processor element is receiving a skewed number of locks (high or low), the signaling event manager may adjust the priorities of the processor elements to balance the number of locks obtained by the various processor elements to prompt forward progress of the waiting processes.

The various embodiments may be useful for any synchronization primitive when multiple participants attempt to access the resources of the computing device and there is a potential for contention for accessing the resources. For ease of explanation, the various embodiments are described with reference to resource locks, however such descriptions are not meant to be limiting.

FIG. 1 illustrates a system including a computing device 10 in communication with a remote computing device 50 suitable for use with the various embodiments. The computing device 10 may include a SoC 12 with a processor 14, a memory 16, a communication interface 18, and a storage interface 20. The computing device may further include a

communication component 22 such as a wired or wireless modem, a storage component 24, an antenna 26 for establishing a wireless connection 32 to a wireless network 30, and/or the network interface 28 for connecting to a wired connection 44 to the Internet 40. The processor 14 may include any of a variety of processor elements, including a number of processor cores. The SoC 12 may include one or more processors 14. The computing device 10 may include more than one SoC 12, thereby increasing the number of processors 14, processor elements, and processor cores. The computing device 10 may also include processor 14 that are not associated with an SoC 12. Individual processors 14 may be multi-core processors as described below with reference to FIGS. 2A and 2B. The processors 14 may each be configured for specific purposes that may be the same as different from other processors 14 of the computing device 10. As such, various processors 14 may include various processor elements of the same or different type. One or more of the processors 14 and processor cores of the same or different configurations may be grouped together.

The memory 16 of the SoC 12 may be a volatile or non-volatile memory configured for storing data and processor-executable code for access by the processor 14. In an embodiment, the memory 16 may be configured to store data structures at least temporarily, such as intermediate processing data output by one or more of the processors 14. In an embodiment, the memory 16 may be cache memory or random access memory (RAM) configured to store information for prioritizing processor elements making lock resource access requests. The memory 16 may include non-volatile read-only memory (ROM) in order to retain the information for prioritizing processor elements attempting to obtain a lock on a resource on the computing device.

The computing device 10 and/or SoC 12 may include one or more memories 16 configured for various purposes. In one embodiment, one or more memories 16 may be configured to be dedicated to storing the information for prioritizing processor elements for a designated set of processor elements. The memory 16 may store the information in a manner that enables the information to be interpreted by the signaling event manager for directed event signaling for multi-processor element systems.

The communication interface 18, communication component 22, antenna 26, and/or network interface 28 may work in unison to enable the computing device 10 to communicate over a wireless network 30 via a wireless connection 32, and/or a wired network 44 with the remote computing device 50. The wireless network 30 may be implemented using a variety of wireless communication technologies, including, for example, radio frequency spectrum used for wireless communications, to provide the computing device 10 with a connection to the Internet 40 by which it may exchange data with the remote computing device 50.

The storage interface 20 and the storage component 24 may work in unison to allow the computing device 10 to store data on a non-volatile storage medium. The storage component 24 may be configured much like an embodiment of the memory 16 in which the storage component 24 may store the information for prioritizing processor elements making lock resource access request, such that information may be accessed by one or more processors 14. The storage component 24, being non-volatile, may return the information even after the power of the computing device 10 has been shut off. When the power is turned back on and the computing device 10 reboots, the information stored on the storage component 24 may be available to the computing device 10. The storage interface 20 may control access to the

storage device 24 and allow the processor 14 to read data from and write data to the storage device 24.

Some or all of the components of the computing device 10 may be differently arranged and/or combined while still serving the necessary functions. Moreover, the computing device 10 may not be limited to one of each of the components, and multiple instances of each component may be included in various configurations of the computing device 10.

FIG. 2A illustrates a processor element, such as a multi-core processor 14, suitable for implementing an embodiment. The multi-core processor 14 may have a plurality of heterogeneous or homogeneous processor elements, such as processor cores 200, 201, 202, 203. The processor cores 200, 201, 202, 203 may be heterogeneous in that the processor cores 200, 201, 202, 203 of a single processor 14 may be configured for the same purpose and have the same or similar performance characteristics. For example, the processor 14 may be a general purpose processor, and the processor cores 200, 201, 202, 203 may be homogeneous general purpose processor cores. Alternatively, the processor 14 may be a graphics processing unit or a digital signal processor, and the processor cores 200, 201, 202, 203 may be homogeneous graphics processor cores or digital signal processor cores, respectively.

Through variations in the manufacturing process and materials, the performance characteristics of homogeneous processor cores 200, 201, 202, 203, may differ from processor core to processor core within the same multi-core processor 14 or within another multi-core processor 14 using the same designed processor cores.

The processor cores 200, 201, 202, 203 may be heterogeneous in that the processor cores 200, 201, 202, 203 of a single processor 14 may be configured for different purposes and/or have different performance characteristics. Examples of such heterogeneous processor cores may include what are known as “big.LITTLE” architectures in which slower, low-power processor cores may be coupled with more powerful and power-hungry processor cores.

The multi-core processor 14 may further include a signaling event manager 206 in communication with each the processor cores 200, 201, 202, 203. The signaling event manager for the signaling manager 206, may receive resource access requests, including requests to lock or unlock a resource of the computing device, prioritize each processor core, and signal the processor cores to wait for an event and signal prioritized processor cores when the event occurs. In an embodiment, the event includes the release of a lock on a resource for which the processor core attempts to obtain a lock.

FIG. 2B illustrates a processor element suitable for implementing an embodiment much like FIG. 2A. In an embodiment, the processor element, in this instance the multi-core processor 14, may include multiple signaling event managers 208, 210, 212. In various embodiments, the signaling event managers 208, 210, 212 may be in communication with a variety of combinations of processor cores 200, 201, 202, 203. In an embodiment, multiple processor cores, like processor core 1 201, processor core 1 201, and processor core 2 202 may be in communication with a shared signaling event manager, like signaling event manager 0 208. In an embodiment, one or more processor cores, like processor core 2 202, may be in communication with multiple signaling event managers, like signaling event manager 0 208 and signaling event manager 1 210. In an embodiment, a processor core, like processor core 3 203, may be in communication with a dedicated signaling event manager, like

signaling event manager 2 212. The signaling event managers 206, 208, 210 may be in communication with each other via a common communication bus of the processor element, or via a dedicated signaling event manager communication bus 230.

FIG. 2C illustrates an embodiment in which the signaling event managers 206, 208, 210 are local to the processor elements, like single or multi-core processor 221, 214, 216, 218, 210. FIG. 2D illustrates an embodiment in which the signaling event managers 206, 208, 210 are local to the processing elements, like SoCs 220, 222, 224, 226. Similar to the example in FIG. 2B, the signaling event managers 206, 208, 210 may be in communication with each other via a common communication bus of the SoC 12 at FIG. 2C or the computing device 10 in FIG. 2D, or via a dedicated signaling event manager communication bus 230. In an embodiment, there may be layers of signaling event managers 206, 208, 210 such that the signaling event managers 206, 208, 210 may be local to the processor cores 210, 201, 202, 203 as in FIG. 2B and in communication with signaling event managers 206, 208, 210 local to higher level components, such as the multi-core processors 221, 214, 216, 218 or the SoCs 220, 222, 224, 226. Similarly, the signaling event managers 206, 208, 210 local to the multi-core processors 221, 214, 216, 218 may be in communication with the signaling event managers 206, 208, 210 local to SoCs 220, 222, 224, 226.

In the examples illustrated in FIGS. 2A and 2B, the multi-core processor 14 includes four processor cores 200, 201, 202, 203 (i.e., processor core 0, processor core 1, processor core 2, and processor core 3). The example illustrated in FIG. 2C, the SoC 12 includes four processor 212, 214, 216, 218 (i.e., processor 0, processor 1, processor 2, and processor 3). In the example illustrated in FIG. 2D, the computing device 10 includes four SoCs 220, 222, 224, 226 (i.e., SoC 0, SoC 1, SoC 2, and SoC 3). For ease of explanation, the various embodiments may be described with reference to any of the processor elements, including the four processor cores 200, 201, 202, 203, processors 212, 214, 216, 218, or SoCs 220, 222, 224, 226 illustrated in FIGS. 2A-2D. However, the processor elements illustrated in FIGS. 2A-2D may be referred to herein merely as an example and in no way are meant to limit the various embodiments to a four-core processor system, a four-processor system, or a four-SoC system. The computing device 10, the SoC 12, or the multi-core processor 14 may individually or in combination include fewer or more than four processor cores, processor, or SoCs.

The multiple signaling event managers 206, 208, 210 may communicate with each other to manage the availability of resources requested by processing elements, the priorities of the processor elements, and signaling the processor elements to wake up and obtain a lock on the requested resource, as will be discussed further herein. The examples discussed herein may be explained in terms of a single signaling event manager; however this is only intended as an example and in no way meant to limit the various embodiments for single signaling event manager systems. The computing device 10, the SoC 12, or the multi-core processor 14 may individually or in combination include more than one signaling event manager.

FIG. 3 illustrates an example processor element, such as a multi-core processor 14, employing a cluster architecture suitable for implementing an embodiment. Much like the example multi-core processor 14 in FIGS. 2A and 2B, the multi-core processor 14 illustrated in FIG. 3 may include four processor cores 200, 201, 202, 203, although this

configuration of processor cores is not limiting and the various embodiments function in a similar manner with fewer or more processor cores. Also similar to the multi-core processor 14 in FIGS. 2A and 2B, the processor cores 200, 201, 202, 203 may all be in communication with the signaling event managers 206, 208, 210, 212. The multi-core processor illustrated in FIG. 3 may further include grouping the processor cores 210, 201, 202, 203 into clusters 300, 302. The multi-core processor 14 may employ the clusters 300, 302 to efficiently execute processing of program instructions and data.

The clusters 300, 302 may be static, maintaining the same processor cores 200, 201, 202, 203 in the cluster 300, 302. The clusters 300, 302 may also be dynamic, changing the processor cores' membership in one or more clusters 300, 302 based on various factors. The example illustrated in FIG. 3 shows two processor cores 200, 201, 202, 203 in each cluster 300, 302. The number and combination of processor cores 200, 201, 202, 203 in the clusters 300, 302 are not meant to be limiting. Further, the number of clusters 300, 302 and configuration of each cluster 300, 302 is also not meant to be limiting. The clusters 300, 302 may include any combination of processor cores 200, 201, 202, 203, including processor cores 200, 201, 202, 203 included in both clusters 300, 302.

In addition to the processor cores 200, 201, 202, 203, the clusters 300, 302 may each include a cluster cache 304, 306, which, in an embodiment, may be configured much like the memory device 16 in FIG. 1, described above. The cluster cache 304, 306 may store data and program instructions for fast access by the processor cores 200, 201, 202, 203 in their respective clusters 300, 302. The cluster cache 304, 306 may store data related to prioritizing the processor cores 200, 201, 202, 203 for resource access request. In an embodiment, the items stored in the cluster cache 304, 306 may be relevant only to the respective cluster 300, 302 of the cluster cache 304, 306. The cluster cache 304, 306 may also store the responses for which the processor cores 200, 201, 202, 203 of the respective clusters 300, 302 issue resource access requests. The multi-core processor 14 may also include a processor cache 308 that may contain data and program instructions similar to that in the cluster cache 304, 306. The processor cache 308 may maintain such information for one or more of the processor cores 200, 201, 202, 203 allocated to one or more of the clusters 300, 302.

The clusters 300, 302 may include processors cores 200, 201, 202, 203 capable of similar or disparate processing. The clusters 300, 302 may also include components, such as the processor cores 200, 201, 202, 203 and cluster caches 304, 306 at relatively close locality compared to other components of the multi-core processor 14 that may be options for inclusion in the clusters 300, 302. The clusters 300, 302 may be configured to more efficiently execute processes and threads than without clustering the processor cores 200, 201, 202, 203.

FIG. 4 illustrates an example table 400 demonstrating relationship between processor elements and their assigned priorities in accordance with an embodiment. The table 400 is an exemplary and non-limiting format for relating some or all of the data shown in table 400. The table 400 illustrates an example of how processor elements listed in column 402 may be correlated (by rows) with alternative columns 404-410, associated with the processor elements, although such data may be stored and related using any number of known data structures. Examples of information regarding each processor element may include a priority in column 404, one or more parameters for determining the priority in

US 9,632,569 B2

11

12

column 406 (e.g., processor element temperature, operating frequency, and/or proximity of clearing of resources by the elements), a number of times the processor element has been signaled to wake up to obtain a resource lock in column 408, and an amount of time since the last occurrence of the processor element being signaled to wake up in column 410, or a time of the last occurrence of the processor element being signaled to wake up. In an embodiment, the parameters for determining the priority may be preprogrammed in software running on the computing device, hard-coded to the signaling event manager, or selected based on various considerations by the signaling event manager.

The table 400 may include all processor elements that make a lock resource access request. In an embodiment, during a particular session on the computing device, until a first processor element makes a lock resource access request, the table 400 may be empty or may not exist. The table 400 may be initiated upon a first lock resource access request by a first processor element. The table 400 may be further populated as other processor elements make other lock resource access requests.

In an embodiment the table 400 may include a row 414, 416, 418, 420, 422, 424 for relating the information of such processor element that makes a lock resource access request. A row 414-424 may be included in the table for each of N number processor elements (i.e., processor element 0 to processor element N-1). Each row 414-424 may be retained in the table 400 after the respective processor elements make an unlock resource access request. Each of the rows 414-424 may be updated upon the respective processor elements making another lock resource access request, upon a processor element making an unlock resource access request, and/or periodic intervals. In an embodiment, rows 414-424 for processor elements making unlock resource access request may be removed from the table 400. The table 400 may be a universal table for all lock resource access requests.

The table 400 may maintain information for each row 414-424 identifying the resource for which the associated processor element (listed in column 402) is attempting to obtain a lock. In such a table 400, it is possible for different processor elements to have the same priority listed in column 404 as long as such priority relates to different resources.

In an embodiment, multiple tables 400 may be implemented in memory, in which case each of the tables 400 may be dedicated for a specific resource.

As an example of the relationship of various processor elements with their respective priorities, and other information relating to the processor elements that may be stored in table 400, table 400 illustrates how a criterion for determining priority according to a heuristic may be the processor element temperature 406. The example shows that the priority listed in column 404 for each processor element may be determined based on an inverse relationship with the processor element temperature listed in column 406. In this example, the lowest processor element temperature may correlate with the highest priority. Thus, processor element 2 in row 418, having the lowest processor element temperature of 28.1 °C., may be assigned the highest priority of 1. Conversely, processor element 3 in row 420, having the highest processor element temperature of 38.2 °C., may be assigned the highest priority of 8. As noted above, parameters other than the processor element temperature 406 may be used as the criterion for determining the priority 404 of the processor elements. In an embodiment, any combination of parameters may be used as criteria for determining priorities of processor elements.

In an embodiment, the number of times that the processor element has been signaled to wake up to obtain a resource lock listed in column 408, and/or the amount of time since the last occurrence of the processor element being signaled to wake up listed in column 410 may be used in a heuristic for setting priorities of processor elements in order to avoid a starvation condition for any of the processor elements. A starved processor element may be forced to wait so long that the process in which the starved processor element attempts to obtain the resource lock may not be progressing in a timely manner. Starvation may occur when a processor element is signaled to wake up significantly less often than other processor elements or waits significantly longer to be signaled to wake up than the other processor elements. The number of times that the processor element has been signaled to wake up to obtain a resource lock may be compared to a wake up number threshold. Additionally or alternatively, the amount of time since the last occurrence of the processor element being signaled to wake up may be compared to a time-since-wake up threshold. The comparison may occur periodically and/or upon a processor element issuing an unlock resource access request. In response to determining that one or more of the starvation thresholds are exceeded, the priority of the starved processor element may be increased to enable the starved processor element to obtain the resource lock before other processor elements that previously had higher priority.

The table 400 may be stored in any of the above mentioned memory and storage devices. For example, the table 400 may be stored in the memory device 16 shown in FIG. 1, the processor cache 308 shown in FIG. 3, or in the signaling event manager 206 shown in FIGS. 2 and 3. In another example, the table 400 may be stored in the above mentioned memory device 16, the processor cache 308, the signaling event manager 206, the cluster cache 304, or the cluster cache 306 shown in FIG. 3. The signaling event manager may access the table 400 from any of these storage locations.

FIG. 5 illustrates a process and signaling flows of cleared event signaling in a multiprocessor system in accordance with an embodiment. For ease of explanation, the example in FIG. 5 includes signaling between the signaling event manager 206, processor element 0 524, processor element 1 526, and processor element 2 528. This example is not meant to be limiting regarding the number of processor elements that may communicate with a signaling event manager, or the number of signaling event managers that may communicate with a processor element. In this example, processor element 0 524 may send a lock resource access request 500 to the signaling event manager 206. In response to an other processor element 526, 528 having a resource lock on the same resource for which processor element 0 524 is requesting the resource lock, the processor element 0 524 may obtain a resource lock 518. The signaling event manager 206 may respond to the lock resource access request 500, but it may not respond in some embodiments when the resource lock is available at the time of the lock resource access request 500.

The signaling event manager 206 may update or initiate a table for associating the processor elements 524, 526, 528 with respective priorities. In an embodiment, various schemes may be used to indicate the priority of a processor element that obtains a resource lock. For example, the processor element having the resource lock may be given the highest priority until it relinquishes the lock, at which point its priority may be updated based on the criterion for priority, whether the processor element assumes another lock

resource access request, and/or based on an opposite use of the starvation information. In other examples, the processor element having the resource lock may be assigned the lowest priority or no priority until making another lock resource access request.

When the processor element 1 526 sends a lock resource access request 502 to the signaling event manager 206 for the same resource locked by resource lock 518, the signaling event manager 206 may update the table and assign processor element 1 526 a priority for obtaining the requested resource lock. The signaling event manager 206 may return a wait-for-event signal 504 to the processor element 1 526 in response to the request resource being locked by the resource lock 518. The wait-for-event signal 504 may trigger the processor element 1 526 to enter a sleep state until it receives a signal that the resource is available for a resource lock (e.g., set event signal). However, the processor element 1 526 may not receive such a set event signal upon a first instance of the resource being available for a resource lock if it does not have the highest priority.

For the above lock resource access requests 500, 502 the information for the criterion for assigning priority may or may not be required by the signaling event manager 206. For the lock resource access request 500 the information may not be required as it is the first lock resource access request and no other processor elements 524, 528 are vying for the resource lock. The lock resource access request 502 may require the information, but it may not be necessary for assigning the priority for processor element 1 526 if it is the next processor element in line for the resource lock. However, the information may be stored in the table for comparison of subsequent lock resource access request as they may cause the signaling event manager 206 to alter the priority of processor element 1 526 in the future.

When processor element 2 528 sends a lock resource access request 506 to the signaling event manager 206 for the same resource as processor element 0 524 and processor element 1 526, the signaling event manager may update the table by comparing the information of the priority criterion for the processor element 2 528 with the information in the table for the other waiting processor elements, in this case processor element 1 526. In this example, processor element 2 528 has parameter values for the priority criterion that result in a higher priority than processor element 1. If the processor element 1 524 still has the resource lock 518, the signaling event manager may return a wait-for-event signal 508 to the processor element 2 528. The wait-for-event signal 508 may trigger the processor element 2 528 to enter a sleep state until it receives a signal that the resource is available for a resource lock (e.g., set event signal).

Processor element 0 524 may send an unlock resource access request 510 to the signaling event manager 206, indicating that the processor element 0 524 is relinquishing the resource lock 518, making that resource available for another of the processor elements 524, 528 to obtain a resource lock on the resource. In response, the signaling event manager 206 may check the table to determine the processor element 524, 526, 528 that has the highest priority. In response to determining that processor element 2 528 has the highest priority, the signaling event manager 206 may send a set event signal, or a wake up signal 512 to that processor element 2 528. The wake up signal 512 may trigger the processor element 2 528 to wake up from its sleep state to obtain a resource lock 520. When it is finished with the resource, the processor element 2 528 may send an unlock resource access request 514 to the set signaling event manager 206 and relinquish the resource lock 520. In

response, the signaling event manager 206 may again check the table to determine the processor element 524, 526, 528 with the highest priority. In the example illustrated in FIG. 5, at this point processor element 1 526 is the only processor element waiting for a resource lock, and therefore it is also the processor element 1 526 with the highest priority. Thus, the signaling event manager 206 may send a wake up signal 516 triggering processor element 1 526 to wake up from its sleep state and obtain a resource lock 522.

The lock resource access requests 500, 502, 506 may or may not include the information for the criterion for the priority and/or starvation. In an embodiment, the criterion for the priority and/or starvation may not be known to the processor elements 524, 526, 528. In this embodiment, the signaling event manager 206 may retrieve the information for the requesting processor elements 524, 526, 528 by requesting the information from another source, such as a kernel or operating system running on the computing device, retrieving the information from memory, or taking measurements from sensors (e.g., temperature sensors) near the processor elements 524, 526, 528. In other embodiments, the processor elements 524, 526, 528 may know the information, but may not know the information that is required by the signaling event manager 206, which may depend upon the particular heuristic that the signaling event manager has selected. Therefore, in response to receiving a lock resource access request 500, 502, 506, the signaling event manager 206 may send a request for the required information and receive the information from the lock requesting processor element in return. In an embodiment, the request for information by the signaling event manager 206 may be part of the wait-for-event signal 504, 508, and the signaling event manager 206 may receive the requested information prior to the processor elements 524, 526, 528 entering the sleep state. In an embodiment, the required information may be indicated in a special register, a memory location, or an added NOP instruction with special meaning before or after the wait-for-event signal 504, 508.

In an embodiment, rather than the signaling event manager 206 receiving unlock resource access requests 510, 514, and sending wake up signals 512, 516, the signaling event manager 206 may send updated priority information to the processor element that has the resource lock, and in response to relinquishing the resource lock, the relinquishing processor element may send a wake up signal to the processor element with the highest priority. In this manner, the highest priority processor element may receive the wake up signal from the relinquishing processor element, causing it to wake up from its sleep state and obtain a resource lock that is now available.

FIG. 6 illustrates an embodiment method 600 for implementing directed event signaling in a multiprocessor system. The method 600 may be executed in a computing device using software, general purpose or dedicated hardware, such as the processor or signaling event manager, or a combination of software and hardware. In block 602, the signaling event manager may receive a resource access request from a processor element. In an embodiment the resource access request may include information that may aid the signaling event manager to determine a priority of the requesting processor element to obtain a lock on the requested resource as discussed above.

In determination block 604, the signaling event manager may determine whether the resource access request is for unlocking an obtained resource lock or for obtaining a resource lock. For example, the resource access request may specify whether the request is an unlock resource access

request or a lock resource access request. In response to determining that the resource access request is an unlock resource access request (i.e., determination block 604 “Unlock”), the signaling event manager may determine whether there are any pending lock resource access requests for the same resource in determining block 606. In an embodiment, the signaling event manager may check the priorities of the processor elements that have previously sent lock resource access requests but were denied because the requested resource was already locked by another processor element. In response to determining that there are no pending lock resource access requests for the same resource (i.e., determination block 606 “NO”), the signaling event manager may return to block 602 to receive other resource access requests. In response to determining that there are pending lock resource access requests (i.e., determination block 606 “YES”), the signaling event manager may send a wake up signal to the highest priority processor element in block 608, thereby triggering the processor element to wake up from a sleep state and obtain a lock on the available resource. The signaling event manager may receive other resource access requests in block 602.

In response to determining that the resource access request is a lock resource access request (i.e., determination block 604 “Lock”), the signaling event manager may determine whether a lock on the requested resource is owned, or already obtained, by another processor element in determination block 610. As discussed above, the signaling event manager may store and maintain data regarding the lock and the processor elements that may be accessed for this operation, such as in a table. Such a table may be dedicated to particular resources, and the signaling event manager may determine the status of an obtained resource lock by analyzing the data in the table. In an embodiment, the data in a table for a particular resource may indicate the status of a resource lock by the priority value. For example, a certain priority value may indicate that the associated processor element currently owns the resource lock. In another embodiment, data (e.g., a binary flag) may indicate ownership of a resource lock. Other embodiments include a universal table rather than a resource specific table that may include an indication of the resource(s) that the processor elements are requesting or own a resource lock. In response to determining that the requested resource is not locked, the requested resource is available, or the resource lock is not owned (i.e., determination block 610 “NO”), the signaling event manager may confirm acquisition of the resource lock to the requesting processor element in block 612. In block 614, the signaling event manager may update the table, or lock request wait list, to reflect the ownership of the resource lock. In the above described embodiment, the signaling event manager may accomplish updating the table by updating one or more of the indicators of resource lock ownership. The signaling event manager may receive other resource access requests in block 602.

In response to determining that the requested resource is locked, the requested resource is unavailable, the resource lock is owned (i.e., determination block 610 “YES”), the signaling event manager may signal the requesting processor element to wait for an event in block 616. In an embodiment, the wait-for-event signal may trigger the processor element to enter a sleep mode until being notified of the event as described above. In block 618, the signaling event manager may update the table, or lock request wait list, to reflect the denied request for the resource lock. In an embodiment, the signaling event manager may update at least the priority of the requesting processor element in the table, indicating that

the requesting processor element has made a request for a resource lock, and where among the other requests of resource locks, the current request ranks. The signaling event manager may receive other resource access requests in block 602.

In an embodiment, during a session on a computing device, the table may not yet be initialized, or fully or partially populated, prior to a first lock resource access request. A fully or partially populated table may include data, but the data may be irrelevant or incomplete before receiving updated data associated with lock resource access requests. To address this embodiment, the signaling event manager or another processor may implement an embodiment method 700 for determining whether such a table exists, and implementing a lock resource access request table, or wait list, when it does not exist as illustrated in FIG. 7. The method 700 may be executed in a computing device using software, general purpose or dedicated hardware, such as the processor or signaling event manager, or a combination of software and hardware.

When the signaling event manager has a need to access a lock resource access request or wait list table, such as upon receiving a resource request in block 602 or determining whether a requested resource is locked in determination block 604 in method 600, the signaling event manager may determine in determination block 702 whether a lock resource access request table, or wait list, exists, or is populated. In response to determining that the table is does not exist or is not initialized (i.e., determination block 702 “NO”), the signaling event manager may initiate the table, or wait list in block 704. In an embodiment, initiating the table may involve creating and filling the table with data associated with at least one processor element making a lock resource access request. In block 706, the signaling event manager may assign the requesting processor element a priority. As described above, the first processor element of the table may have any priority since there are no other processor elements for comparison. In an embodiment, the first processor element in the table may be assigned a default priority or a random priority.

In response to determining that the table does exist or is initialized (i.e., determination block 702 “YES”), the signaling event manager may determine the priority of the requesting processor elements in block 708. As discussed above, the signaling event manager may receive the relevant data for determining the priority of a processor element by various means, including multiple signals between the signaling event manager and other hardware and software components of the computing device. The data that is relevant for determining the priority of a processor element may be predetermined by software or firmware, or may be selected by the signaling event manager in block 710. The signaling event manager may update the table. Depending on the configuration of the table, the signaling event manager may update the priority of the requesting processor element, and the other processor elements listed in the table based on comparisons of the data associated with the requesting processor element with the data associated with the other processor elements. Updating the table to include or update the priority of the requesting processor element may cause changes to the priorities of the other processor elements in response to the priority of the requesting processor element being higher. In optional block 712, the signaling event manager may send the table to the processor element that owns the resource lock in embodiments in which the processor element owning the resource lock notifies the next waiting processor element when the

resource is unlocked. The operations in optional block 712 enable the processor element owning the resource lock to determine the next waiting processor element that will receive the lock based on the priorities in the table.

FIG. 8 illustrates an embodiment method 800 for implementing prioritization of processor elements for directed event signaling in a multiprocessor system. The method 800 may be executed in a computing device using software, general purpose or dedicated hardware, such as the processor or signaling event manager, or a combination of software and hardware. In block 802, the signaling event manager may determine the criterion or criteria for assigning priorities to the requesting processor elements. The criterion or criteria may include one or more of the processor element temperature, operating frequency, and locality. In an embodiment, the signaling event manager may select the criterion or criteria for assigning priorities, which may depend on the states of the computing device at the time. For example, states of high temperature, low power availability, power saving modes, and low priority processes may lead the signaling event manager to select low processor element temperature and/or low operating frequency as the criteria for higher priority. On the other hand, a fully charged or charging battery, and high performance software may lead the signaling event manager to select high operating frequency and/or high degree of locality as the criteria for higher priority. In other embodiments, the signaling event manager may determine the criterion or criteria for assigning priority by retrieving the criterion or criteria from storage, such as a register integrated in or external to the signaling event manager, or as provided by a software program running on the computing device.

In block 804, the signaling event manager may compare the one or more parameters of the requesting processor element related to the selected criterion in one or more similar parameters of the other processor elements stored in the table to determine the processor element that best satisfies the selected criterion or criteria for assigning priority. Depending on the selected criterion or criteria for determining the priorities of the processor elements, a favorable comparison for the requesting processor element may result from the data of the requesting processor element being higher or lower than that of the comparable data of the other processor elements. A favorable comparison for the requesting processor element may result in a higher priority than at least one of the other processor elements waiting to obtain a resource lock on the same resource as the requesting processor element. In some embodiments, the comparison in block 804 may be a direct comparison of the values of these parameters. In some embodiments, the comparison in block 804 may be an indirect comparison in which one or more algorithms calculations are applied to the values of these parameters and the results of these algorithmic calculations are compared.

In block 806, the signaling event manager may assign a priority to the requesting processor element based on the parameter comparisons. Assigning the priority to the requesting processor element may also result in changes in priority for one or more of the other processor elements waiting to obtain a resource lock on the same resource. For example, in response to the requesting processor element being assigned a higher priority than three other processor elements, the priority of each of the three other processor elements may be decremented accordingly. At the same time, other processor elements having a higher priority than the priority assigned to the requesting processor element may remain unchanged by the priority assignment.

FIG. 9 illustrates an embodiment method 900 for implementing starvation avoidance for directed event signaling in a multiprocessor system. The method 900 may be executed in a computing device using software, general purpose or dedicated hardware, such as the processor or signaling event manager, or a combination of software and hardware. In block 902, the signaling event manager may track wake up statistics for one or more processor elements that have made a lock resource access request. In an embodiment, such wake up statistics may include the number of times the signaling event manager has sent a wake up signal to a processor element, the time at which the processor element was last sent a wake up signal, the elapsed time since the last wake up signal was sent to the processor element, the time in which the processor element made the pending lock resources access request, and/or the elapsed time since the processor element made the pending lock resources access request.

In block 904, the signaling event manager may compare the wake up statistics for a processor element with a pending lock resource access request to a wake up threshold. In an embodiment, the wake up threshold may be a predetermined value, a value based on historical information for the same statistic, or a function of one or more of the values of the same statistic for other processor elements with pending lock resource access requests.

In determination block 906, the signaling event manager may determine whether the wake up statistics of the processor element with the pending lock resource access request exceed the wake up threshold. In response to determining that the wake up statistics of the processor element with the pending lock resource access request do not exceed the wake up threshold (i.e., determination block 906 “NO”), the signaling event manager may maintain the table without changes in block 908.

In response to determining that the wake up statistics of the processor element with the pending lock resource access request do exceed the wake up threshold (i.e., determination block 906 “YES”), the signaling event manager may assign the processor element a higher priority in block 910, and update the table in block 912 as described in block 710 above. The higher priority assigned to the processor element may be determined numerous ways. In an embodiment, the higher priority may be assigned according to a predetermined rule, such as decrementing the priority by a certain value, assigning the processor element a constant priority (e.g., the highest priority), or inverting the position of the processor element in the priority queue. In an embodiment, the higher priority may be assigned according to a severity of starvation of the processor element. The severity may be determined by a function of the wake up statistics of the processor element as compared to the wake up statistics of the other processor elements. The greater the severity of starvation, the higher the priority that may be assigned to the processor element.

FIG. 10 illustrates an embodiment method 1000 for implementing lock-aware scheduling with directed event signaling in a multiprocessor system. The method 1000 may be executed in a computing device using software, general purpose or dedicated hardware, such as the processor or signaling event manager, or a combination of software and hardware. In block 1002, the signaling event manager may detect that multiple threads are requesting the same resource lock, such as when multiple threads use common resources to execute on different processor elements.

In block 1004, the signaling event manager may request the operating system scheduler to migrate some or all of the

threads contending for the same resource to certain processor elements. The signaling event manager may request migrating the threads to certain elements to take advantage of certain efficiencies of different system architectures. In an embodiment, the processor elements may be arranged in a cluster architecture. Selecting certain processor elements of a cluster to execute threads contending for the same resource may maximize performance when the processor elements of the cluster are operating in the same frequency. In an embodiment, processor elements of a cluster tend to be in close proximity. Thermal dissipation may be improved when waiting processor elements of the cluster can be stalled with almost zero power consumption, and the area of the stalled processor elements can be used to dissipate the heat generated by other running processor elements. In an embodiment, the maximum frequency of the running processor elements may exceed the maximum recommended temperature and/or operating frequency when the extra area of the stalled processor elements is used for thermal dissipation. In an embodiment, processor architectures may employ shared portions of memory hierarchy by multiple processor elements, in which case access latency to the memory may be reduced or minimized as priority criteria, and locality increased or maintained as priority criteria in various heuristics. In computing systems that include multiple power sources, priority criteria may be adjusted so that threads may be moved to processor elements using the same power sources while processor elements on other power sources may be reduced in priority so they may not receive threads and therefore powered off. In block 1006, the signaling event manager may receive confirmation of the reallocation of the threads contending for the same resources are reallocated to the requested processor elements. In block 1008, the signaling event manager may update the table as described above for block 710.

FIG. 11 illustrates an embodiment method 1100 for implementing lock stall actions with directed event signaling in a multiprocessor system. The method 1100 may be executed in a computing device using software, general purpose or dedicated hardware, such as the processor or signaling event manager, or a combination of software and hardware. In block 1102, the signaling event manager may estimate the sleep time for a processor element (i.e., the duration that the processor may remain in a low-power state) while in a current wait-for-event state waiting for a requested resource lock to become available. In an embodiment, the signaling event manager may make the estimate based on one or more of historical information, expected execution times for threads on processor elements waiting for the same resource with higher priorities, and the number of higher priority processor elements waiting for the resource. For example, if the processor element has low priority and there are multiple processor elements with higher priority waiting for the same resource, the signaling event manager may estimate that the possessing element will have a relatively long sleep time, especially compared to when the processor element has the highest priority or is the only element on the waiting list and thus is next in line to use the resource.

In block 1104, the signaling event manager may determine a sleep state for the requesting processor element based on the estimated sleep time. Various embodiments may implement numerous different sleep states for the processor elements. Different sleep states may be characterized by the level of power consumption. Higher power consumption sleep states may provide less power savings but be capable of waking up faster, such as by retaining working data in volatile memory. In contrast, lower power consumption

sleep states may save more power by turning off more resources and powering down memory, resulting in the need to reenergize more resources (which requires power) and reload state data into volatile memory, which takes more time in order to ramp up to executing instructions. In an embodiment, the signaling event manager may select higher power consumption sleep states for shorter estimated sleep times because the power savings over a shorter period of time may not be worth the loss in efficiency due to having to reestablish the processor element within the system. Similarly, for longer sleep times the signaling event manager may select lower power consumption sleep states because of the net benefit of the greater power savings. In block 1106, the signaling event manager may notify the requesting processor element of the designated sleep state for the estimated sleep time. In an embodiment, the requesting processor element may implement the designated sleep state.

FIG. 12 illustrates an embodiment method 1200 for implementing lock stall actions with directed event signaling in a multiprocessor system. The method 1200 may be executed in a computing device using software, general purpose or dedicated hardware, such as the processor or signaling event manager, or a combination of software and hardware. In block 1202, the signaling event manager may estimate the sleep time for a processor element (i.e., the duration that the processor may remain in a low-power state) while in a current wait-for-event state waiting for a requested resource lock to become available. In block 1204, the signaling event manager may notify the operating system scheduler of the estimated sleep time. In an embodiment, in optional block 1206, the operating system scheduler may remove the requesting processor element from a scheduling queue for a period of time associated with the estimated sleep time. In an alternative embodiment, in optional block 1208, the operating system scheduler may assign a different thread to the processor element during the estimated sleep time, such as a thread that will not require use of the resource for which the processor element received a wait-for-event signal.

FIG. 13 illustrates an example mobile computing device suitable for use with the various embodiments, for instance, embodiments shown in and/or described with reference to FIGS. 1-12. The mobile computing device 1300 may include a processor 1302 coupled to a touchscreen controller 1304 and an internal memory 1306. The processor 1302 may be one or more multi-core integrated circuits designated for general or specific processing tasks. The internal memory 1306 may be volatile or non-volatile memory, and may also be secure and/or encrypted memory, or unsecure and/or unencrypted memory, or any combination thereof. Examples of memory types which can be leveraged include but are not limited to DDR, LPDDR, GDDR, Wi-Fi, RAM, SRAM, DRAM, PRAM, R-RAM, M-RAM, SLL-RAM, and embedded DRAM. The touchscreen controller 1304 and the processor 1302 may also be coupled to a touchscreen panel 1302, such as a resistive-sensing touchscreen, capacitive-sensing touchscreen, infrared-sensing touchscreen, etc. Additionally, the display of the computing device 1300 need not have touch screen capability.

The mobile computing device 1300 may have one or more radio signal transceivers 1308 (e.g., Peanut, Bluetooth, Zigbee, Wi-Fi, RF, radio) and antenna 1310, for sending and receiving communications, coupled to each other and/or to the processor 1302. The transceivers 1308 and antenna 1310 may be used with the above-mentioned circuitry to implement the various wireless transmission protocol stacks

and interfaces. The mobile computing device 1300 may include a cellular network wireless modem chip 1316 that enables communication via a cellular network and is coupled to the processor.

The mobile computing device 1300 may include a peripheral device connection interface 1318 coupled to the processor 1302. The peripheral device connection interface 1318 may be singularly configured to accept one type of connection, or may be configured to accept various types of physical and communication connections, common or proprietary, such as USB, FireWire, Thunderbolt, or PCIe. The peripheral device connection interface 1318 may also be coupled to a similarly configured peripheral device connector port (not shown).

The mobile computing device 1300 may also include speakers 1314 for providing audio outputs. The mobile computing device 1300 may also include a housing 1320, constructed of a plastic, metal, or a combination of materials, for containing all or some of the components discussed herein. The mobile computing device 1300 may include a power source 1322 coupled to the processor 1302, such as a disposable or rechargeable battery. The rechargeable battery may also be coupled to the peripheral device connection port to receive a charging current from a source external to the mobile computing device 1300. The mobile computing device 1300 may also include a physical button 1324 for receiving user inputs. The mobile computing device 1300 may also include a power button 1326 for turning the mobile computing device 1300 on and off.

The various embodiments, for instance, embodiments shown in and/or described with reference to FIGS. 1-12, may also be implemented within a variety of mobile computing devices, such as a laptop computer 1400 illustrated in FIG. 34. Many laptop computers include a touchpad touch surface 1417 that serves as the computer's pointing device, and thus may receive drag, scroll, and click gestures similar to those implemented on computing devices equipped with a touch screen display and described above. A laptop computer 1400 will typically include a processor 1411 coupled to volatile memory 1412 and a large capacity nonvolatile memory, such as a disk drive 1413 or flash memory. Additionally, the computer 1400 may have one or more antennas 1408 for sending and receiving electromagnetic radiation that may be connected to a wireless data link and/or cellular telephone transceiver 1416 coupled to the processor 1411. The computer 1400 may also include a floppy disc drive 1414 and a compact disc (CD) drive 1415 coupled to the processor 1411. In a notebook configuration, the computer housing includes the touchpad 1417, the keyboard 1418, and the display 1419, all coupled to the processor 1411. Other configurations of the computing device may include a computer mouse or trackball coupled to the processor (e.g., via a USB input) as are well known, which may also be used in conjunction with the various embodiments.

The various embodiments, for instance, embodiments shown in and/or described with reference to FIGS. 1-12, may also be implemented in any of a variety of commercially available servers for compressing data in server cache memory. An example server 1500 is illustrated in FIG. 15. Such a server 1500 typically includes one or more multi-core processor assemblies 1501 coupled to volatile memory 1502 and a large capacity nonvolatile memory, such as a disk drive 1504. As illustrated in FIG. 15, multi-core processor assemblies 1501 may be added to the server 1500 by inserting them into the racks of the assembly. The server 1500 may also include a floppy disc drive, compact disc

(CD or DVD) disc drive 1506 coupled to the processor 1501. The server 1500 may also include network access ports 1503 coupled to the multi-core processor assemblies 1501 for establishing network interface connections with a network 1505, such as a local area network coupled to other broadcast system endpoints and servers, the Internet, the public switched telephone network, and/or a cellular data network (e.g., CDMA, TDMA, (GSM, PCS, 3G, 4G), UMTS, or any other type of cellular data network).

Computer program code or "program code" for executing on a programmable processor for carrying out operations of the various embodiments may be written in a high level programming language such as C, C++, C#, Smalltalk, Java, JavaScript, Visual Basic, a Structured Query Language (e.g., Transact-SQL), Perl, in various other programming languages. Program code or programs stored on a computer readable storage medium as used in this application may refer to machine language code (such as object code) whose format is understandable by a processor.

Many computing devices operating system kernels are organized into a user space (where non-privileged code runs) and a kernel space (where privileged code runs). This separation is of particular importance in Android and other general public license (GPL) environments in which code that is part of the kernel space must be GPL licensed, while code running in the user-space may not be GPL licensed. It should be understood that at least some of the various software components modules discussed here may be implemented in either the kernel space or the user space, unless expressly stated otherwise.

The foregoing method descriptions and the process flow diagrams are provided merely as illustrative examples and are not intended to require or imply that the operations of the various embodiments must be performed in the order presented. As will be appreciated by one of skill in the art, the order of operations in the foregoing embodiments may be performed in any order. Further, any reference to claim elements in the singular, for example, using the articles "a," "an," or "the" is not to be construed as limiting the element to the singular.

The various illustrative logical blocks, modules, circuits, and algorithm operations described in connection with the various embodiments may be implemented as electronic hardware, computer software, or combinations of both. To clearly illustrate this interchangeability of hardware and software, various illustrative components, blocks, modules, circuits, and operations have been described above generally in terms of their functionality. Whether such functionality is implemented as hardware or software depends upon the particular application and design constraints imposed on the overall system. Skilled artisans may implement the described functionality in varying ways for each particular application, but such implementation decisions should not be interpreted as causing a departure from the scope of the present invention.

The hardware used to implement the various illustrative logic, logical blocks, modules, and circuits described in connection with the embodiments disclosed herein may be implemented or performed with a general purpose processor, a digital signal processor (DSP), an application specific integrated circuit (ASIC), a field programmable gate array (FPGA), or other programmable logic device, discrete gate or transistor logic, discrete hardware components, or any combination thereof designed to perform the functions described herein. A general-purpose processor may be a microprocessor, but, in the alternative, the processor may be any conventional processor, controller, microcontroller, or state

machine. A processor may also be implemented as a combination of computing devices, e.g., a combination of a DSP and a microprocessor; a plurality of microprocessors, one or more microprocessors in conjunction with a DSP core, or any other such configuration. Alternatively, some operations or methods may be performed by circuitry that is specific to a given function.

In one or more embodiments, the functions described may be implemented in hardware, software, firmware, or any combination thereof. If implemented in software, the functions may be stored as one or more instructions or code on a non-transitory computer-readable medium or a non-transitory processor-readable medium. The operations of a method or algorithm disclosed herein may be embodied in a processor-executable software module that may reside on a non-transitory computer-readable, processor-readable storage medium. Non-transitory computer-readable processor-readable storage media may be any storage media that may be accessed by a computer or a processor. By way of example but not limitation, such non-transitory computer-readable or processor-readable media may include RAM, ROM, EPROM, FLASH memory, CDRW in other optical disk storage, magnetic disk storage or other magnetic storage devices, or any other medium that may be used to store desired program code in the form of instructions or data structures and that may be accessed by a computer disk and disc, as used herein, includes compact disc (CD), laser disc, optical disc, digital versatile disc (DVD), floppy disk, and Blu-ray disc where disks usually reproduce data magnetically, while discs reproduce data optically with lasers. Combinations of the above are also included within the scope of non-transitory computer-readable and processor-readable media. Additionally, the operations of a method or algorithm may reside as one or any combination or set of codes and/or instructions on a non-transitory processor-readable medium and/or computer-readable medium, which may be incorporated into a computer program product.

The preceding description of the disclosed embodiments is provided to enable any person skilled in the art to make or use the present invention. Various modifications to these embodiments will be readily apparent to those skilled in the art, and the generic principles defined herein may be applied to other embodiments without departing from the spirit or scope of the invention. Thus, the present invention is not intended to be limited to the embodiments shown herein but is to be accorded the widest scope consistent with the following claims and the principles and novel features disclosed herein.

What is claimed is:

1. A method for managing processor elements contending for one or more common resources on a computing device having a plurality of processor elements, comprising:
 - determining a parameter for use in assigning priorities to the plurality of processor elements requesting access to a resource based on an operating state of the computing device;
 - assigning a priority to a processor element for requesting access to the resource, wherein the priority is assigned based on the parameter for the processing element;
 - signaling the processor element requesting access to the resource to wait for the resource to become available in response to determining that the resource is not available; and
 - signaling the processor element to access the resource in response to determining that the resource is available and that the processor element is assigned a highest priority for requesting access to the resource.

2. The method of claim 1, further comprising:
 - receiving a signal indicating availability of the resource; and
 - identifying one of the plurality of processor elements that is assigned the highest priority for accessing the resource in response to the signal indicating availability of the resource.

3. The method of claim 1, wherein:
 - signaling the processor element requesting access to the resource to wait for the resource to become available in response to determining that the resource is not available comprises triggering the processor element to enter a sleep state; and
 - signaling the processor element to access the resource comprises signaling the processor element to wake up.

4. A computing device, comprising:
 - a plurality of processor elements; and
 - a signaling event manager communicatively connected to the plurality of processor elements and configured with signaling event manager-executable instructions to perform operations comprising:

5. The computing device of claim 4, wherein the signaling event manager is further configured with the signaling event manager-executable instructions to perform operations comprising:
 - determining a parameter for use in assigning priorities to the plurality of processor elements requesting access to a resource based on an operating state of the computing device;
 - assigning a priority to a processor element for requesting access to the resource, wherein the priority is assigned based on the parameter for the processing element;
 - signaling the processor element requesting access to the resource to wait for the resource to become available in response to determining that the resource is not available; and
 - signaling the processor element to access the resource in response to determining that the resource is available and that the processor element is assigned a highest priority for requesting access to the resource.

6. The computing device of claim 4, wherein the signaling event manager is further configured with signaling event manager-executable instructions to perform operations such that:
 - signaling the processor element requesting access to the resource to wait for the resource to become available in response to determining that the resource is not available comprises triggering the processor element to enter a sleep state; and
 - signaling the processor element to access the resource comprises signaling the processor element to wake up.

7. A non-transitory processor-readable medium having stored thereon processor-executable software instructions to cause a processor of a computing device having a plurality of processing elements to perform operations comprising:
 - determining a parameter for use in assigning priorities to the plurality of processor elements requesting access to a resource based on an operating state of the computing device;
 - assigning a priority to a processor element for requesting access to the resource, wherein the priority is assigned based on the parameter for the processing element;
 - signaling the processor element requesting access to the resource to wait for the resource to become available in response to determining that the resource is not available; and
 - signaling the processor element to access the resource in response to determining that the resource is available and that the processor element is assigned a highest priority for requesting access to the resource.

assigning a priority to a processor element for requesting access to the resource, wherein the priority is assigned based on the parameter for the processing element; signaling the processor element requesting access to the resource to wait for the resource to become available in response to determining that the resource is not available; and

signaling the processor element to access the resource in response to determining that the resource is available and that the processor element is assigned a highest priority for requesting access to the resource;

8. The non-transitory processor-readable medium of claim 7, wherein the stored processor-executable software instructions are configured to cause the processor to perform operations further comprising:

receiving a signal indicating availability of the resource; and

identifying one of the plurality of processor elements that is assigned the highest priority for accessing the resource in response to the signal indicating availability of the resource;

9. The non-transitory processor-readable medium of claim 7, wherein the stored processor-executable software instructions are configured to cause the processor to perform operations such that:

signaling the processor element requesting access to the resource to wait for the resource to become available in response to determining that the resource is not available comprises triggering the processor element to enter a sleep state; and

signaling the processor element to access the resource comprises signaling the processor element to wake up;

10. A computing device having a plurality of processor elements, comprising:

means for determining a parameter for use in assigning priorities to the plurality of processor elements requesting access to a resource based on an operating state of the computing device;

means for assigning a priority to a processor element for requesting access to the resource, wherein the priority is assigned based on the parameter for the processing element;

means for signaling the processor element requesting access to the resource to wait for the resource to become available in response to determining that the resource is not available; and

means for signaling the processor element to access the resource in response to determining that the resource is available and that the processor element is assigned a highest priority for requesting access to the resource;

11. The computing device of claim 10, further comprising:

means for receiving a signal indicating availability of the resource; and

means for identifying one of the plurality of processor elements that is assigned the highest priority for accessing the resource in response to the signal indicating availability of the resource;

12. The computing device of claim 10, wherein:

means for signaling the processor element requesting access to the resource to wait for the resource to become available in response to determining that the resource is not available comprises means for triggering the processor element to enter a sleep state; and

means for signaling the processor element to access the resource comprises means for signaling the processor element to wake up.

< > < > >



US09733978B2

(12) United States Patent
Suarez Gracia et al.(10) Patent No.: US 9,733,978 B2
(11) Date of Patent: Aug. 15, 2017

(54) DATA MANAGEMENT FOR MULTIPLE PROCESSING UNITS USING DATA TRANSFER COSTS

(71) Applicant: QUALCOMM Incorporated, San Diego, CA (USA)

(72) Inventors: Darío Suarez Gracia, Palo Alto, CA (USA); Tushar Kumar, San Francisco, CA (USA); Aravind Natarajan, Sunnyvale, CA (USA); Ravish Hastaantram, San Jose, CA (USA); Ghenghe Calin Cacevici, Palo Alto, CA (USA); Han Zhao, Santa Clara, CA (USA)

(73) Assignee: QUALCOMM Incorporated, San Diego, CA (USA)

(18) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days

(21) Appl. No.: 14/837,156

(22) Filed: Aug. 27, 2015

(16) Prior Publication Data

US 2017/0060633-A1 Mar 2, 2017

(51) Int. Cl.

G06F 9/26 (2006.01)

G06F 15/173 (2006.01)

(Continued)

(52) U.S. Cl.

CPC G06F 9/48 (2013.01), G06F 9/4806

(2013.01); G06F 9/4837 (2013.01);

(Continued)

(58) Field of Classification Search

None

See application file for complete search history.

(56) References Cited

U.S. PATENT DOCUMENTS

6,180,870 B2 * 11/25/01 Relg G06F 9/5066
6,675,139 B2 * 1/25/04 Relg G06N 9/1005

(10/23)

(Continued)

OTHER PUBLICATIONS

Fisher N., et al., "Thermal-Aware Global Real-Time Scheduling on Multicore Systems," Feb. 3, 2010, 45 pages

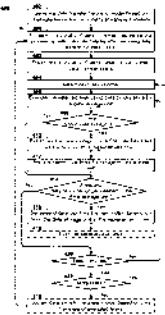
(Continued)

Primary Examiner: Charles Swift
(74) Attorney, Agent or Firm: The Marbury Law Group, PLLC

(57) ABSTRACT

Various embodiments include methods for data management in a computing device utilizing a plurality of processing units. Embodiment methods may include generating a data transfer heuristic model based on measurements from a plurality of sample data transfers between a plurality of data storage units. The generated data transfer heuristic model may be used to calculate data transfer costs for each of a plurality of tasks. The calculated data transfer costs may be used to schedule execution of the plurality of tasks in an execution order on selected ones of the plurality of processing units. The data transfer heuristic model may be updated based on measurements of data transfers occurring during the executions of the plurality of tasks (e.g., time, power consumption, etc.). Code executing on the processing units may indicate to a runtime when certain data blocks are no longer needed and thus may be evicted and/or pre-fetched for others.

26 Claims, 8 Drawing Sheets



US 9,733,978 B2

Page 2

<p>(51) Int. Cl.</p> <p><i>G06F 1/40</i> (2006.01)</p> <p><i>G06F 9/48</i> (2006.01)</p> <p><i>G06F 9/50</i> (2006.01)</p> <p><i>G06F 12/0806</i> (2016.01)</p> <p><i>G06F 12/0862</i> (2016.01)</p> <p><i>G06F 12/12</i> (2016.01)</p> <p><i>G06F 11/34</i> (2006.01)</p>	<p>2015.0085249 A1 * 3.2015. Harinen et al.</p> <p>2015.0109571 A1 * 4.2015. Dubcic</p> <p>2015.0178138 A1 * 6.2015. Saha et al.</p> <p>2015.0199215 A1 * 7.2015. Markovskiy</p>	<p>G06F 9/4891</p> <p>"18.03"</p> <p>G06F 9/5001</p> <p>"18.102"</p>
(52) U.S. CL.		
<p><i>G06F 9/4837</i> (2013.01); <i>G06F 9/4843</i> (2013.01); <i>G06F 9/4881</i> (2013.01); <i>G06F 9/4895</i> (2013.01); <i>G06F 9/50</i> (2013.01); <i>G06F 9/5005</i> (2013.01); <i>G06F 9/5011</i> (2013.01); <i>G06F 9/5016</i> (2013.01); <i>G06F 9/5022</i> (2013.01); <i>G06F 9/5027</i> (2013.01); <i>G06F 9/5033</i> (2013.01); <i>G06F 9/5034</i> (2013.01); <i>G06F 9/5055</i> (2013.01); <i>G06F 9/5074</i> (2013.01); <i>G06F 9/5083</i> (2013.01); <i>G06F 9/5084</i> (2013.01); <i>G06F 9/5093</i> (2013.01); <i>G06F 11/3419</i> (2013.01); <i>G06F 11/3466</i> (2013.01); <i>G06F 12/0806</i> (2013.01); <i>G06F 12/0862</i> (2013.01); <i>G06F 22/2 62</i> (2013.01)</p>	<p>2015.0085249 A1 * 3.2015. Harinen et al.</p> <p>2015.0109571 A1 * 4.2015. Dubcic</p> <p>2015.0178138 A1 * 6.2015. Saha et al.</p> <p>2015.0199215 A1 * 7.2015. Markovskiy</p>	<p>G06F 9/4891</p> <p>"18.03"</p> <p>G06F 9/5001</p> <p>"18.102"</p>
(54) References Cited		
U.S. PATENT DOCUMENTS		
<p>7,886,172 B2 2.2011. Basu et al.</p> <p>8,276,164 B2 9.2012. Mundie et al.</p> <p>8,468,907 B2 6.2013. Agarwal et al.</p> <p>8,510,541 B2 8.2013. Leiferow et al.</p> <p>8,535,716 B2 9.2013. Lippert et al.</p> <p>8,566,570 B2 10.2013. Shi et al.</p> <p>8,566,572 B2 10.2013. Alexander et al.</p> <p>8,707,314 B2 4.2014. Gummadi et al.</p> <p>8,782,635 B2 7.2014. Breitwitz et al.</p> <p>8,829,031 B2 12.2014. Kudlick.</p> <p>8,954,086 B2 2.2015. Rangarajan et al.</p> <p>2010.0294765 A1 * 8.2009. Gupta</p>	<p>2015.12.121</p> <p>211.133</p> <p>4561.17.20</p> <p>769.122</p> <p>1.3293</p> <p>790.391</p> <p>1.3633</p> <p>718.192</p> <p>1.367.2</p> <p>705.7.12</p> <p>4584.1.329</p> <p>700.291</p> <p>6002.20.00</p> <p>18.104</p> <p>6003.9.4863</p> <p>18.103</p> <p>6003.9.4893</p> <p>18.104</p> <p>6003.9.5021</p> <p>729.708</p>	<p>International Search Report and Written Opinion - PCT/US2016/044168, ISA/US, Oct. 6, 2016.</p> <p>Lee J.H. et al., "IPerf: A lightweight Profiler for Task Distribution on GPU+CPU Platforms", Mar. 2, 2015 (Mar. 2, 2015), 14 Pages XIP555401790, Retrieved from the Internet TIRI: https://strataech.github.io/bitstream/handle/1853/3398/GT-CS-15-01.pdf?sequence=1&isAllowed=y [retrieved on Sep. 15, 2016].</p> <p>Lee J.H. et al., "IPerf: A lightweight Profiler for Task Distribution on GPU+CPU Platforms", Mar. 2, 2015 (Mar. 2, 2015), 14 Pages XIP555401814, Retrieved from the Internet TIRI: https://strataech.github.io/bitstream/handle/1853/3398/show/full.pdf?sequence=1&isAllowed=y [retrieved on Sep. 15, 2016].</p> <p>Ploofidmehr P.M. et al., "Portable Performance on Heterogeneous Architectures", PI 10/09 - Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, Jun. 15-20, 2009, Dublin, Ireland, Program Notes: A Monthly Publication of the Special Interest Group on Programming Languages of the ACM, Vol. 48, No. 4, May 16, 2013 (May 16, 2013), pp. 431-444, XIP558010391, DOI: 10.1145/2499368.2451167, ISBN: 978-1-4503-0521-3.</p> <p>Ponam et al., "MDR: Performance Model Driven Runtime for Heterogeneous Parallel Platforms", Supercomputing, ACM, 7 Penn Plaza, Suite 3701, New York, NY, 10114-0001, USA, May 31, 2011 (May 31, 2011), pp. 225-31, XIP558010392, DOI: 10.1145/2499368.2451168, ISBN: 978-1-4503-0522-0.</p> <p>Qualcomm Technologies, et al., "Unimform MDR: Enabling Applications for Heterogeneous Mobile Devices", Apr. 21, 2011 (Apr. 21, 2011), XIP555010666, 1 Pg. http://dev.eceps.qualcomm.com/file/27579/mare-whitepaper.pdf [retrieved on Sep. 15, 2016].</p> <p>Anton Robin, "Efficient Scheduling of Multi-Versioned Tasks", corresponding Qualcomm, unpublished, 8 Appl. No. 14,882,747, filed Sep. 14, 2011, 72 pages includes filing Receipt, Specification and Drawings as filed.</p> <p>Han Zhao et al., "Practical Resource Management for Parallel Work-Stealing Processing Systems", corresponding Qualcomm, unpublished, 8 Appl. No. 14,862,473, filed Sep. 15, 2015, 65 pages includes filing Receipt, Specification and Drawings as filed.</p> <p>Han Zhao et al., "Adaptive Chunk Size Tuning for Data Parallel Processing on Multi-core Architecture", corresponding Qualcomm, unpublished, 8 Appl. No. 14,862,398, filed Sep. 15, 2015, 60 pages includes filing Receipt, Specification and Drawings as filed.</p>
<p>2014.0056998 A1 * 3.2012. Majumdar et al.</p> <p>2014.0136554 A1 * 3.2012. Jain</p> <p>2014.0171591 A1 * 3.2012. Misrahi</p> <p>2014.0192193 A1 * 7.2012. El-Masry</p> <p>2014.0261125 A1 * 10.2012. Ispasov et al.</p> <p>2014.034170 A1 * 3.2013. Prabhakar</p> <p>2014.0365534 A1 * 1.2014. Jain</p> <p>2014.0396666 A1 * 1.2014. Dirakci et al.</p> <p>2014.0367994 A1 * 3.2014. Petraswamy Naga</p> <p>2014.015481 A1 * 7.2014. Patel</p> <p>2014.0282995 A1 * 9.2014. Lee et al.</p> <p>2014.0363347 A1 * 10.2014. Aylesworth</p> <p>2015.0066750 A1 * 3.2015. Kim et al.</p>	<p>10/41.41.0896</p> <p>729.217</p> <p>718.193</p> <p>6560.12.0631</p> <p>718.193</p>	<p>10/41.41.0896</p> <p>729.217</p> <p>718.193</p> <p>6560.12.0631</p> <p>718.193</p>
* cited by examiner		

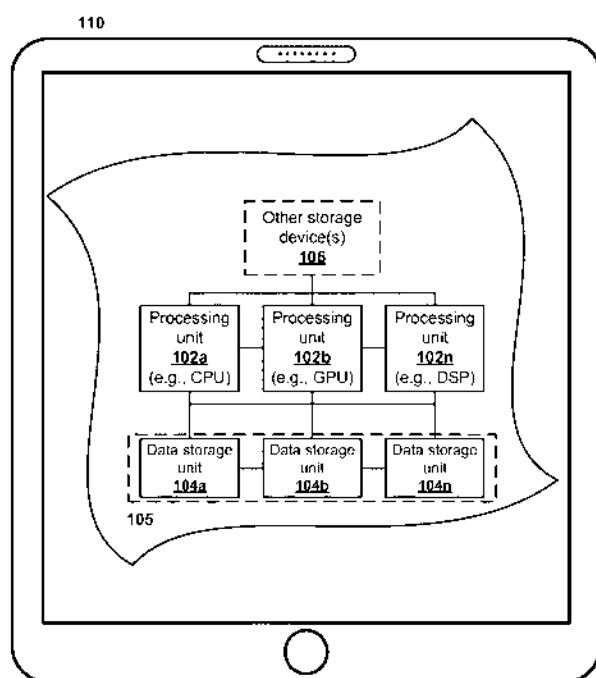


FIG. 1

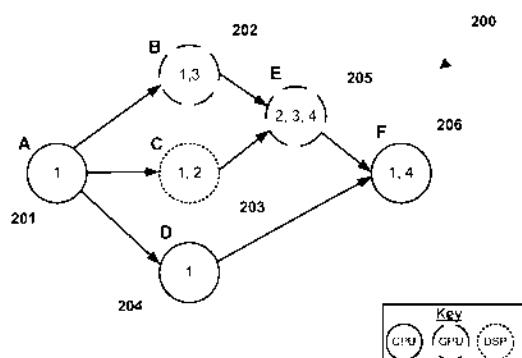


FIG. 2A

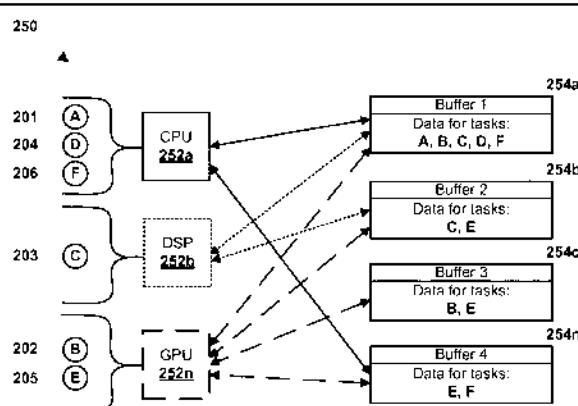


FIG. 2B

300

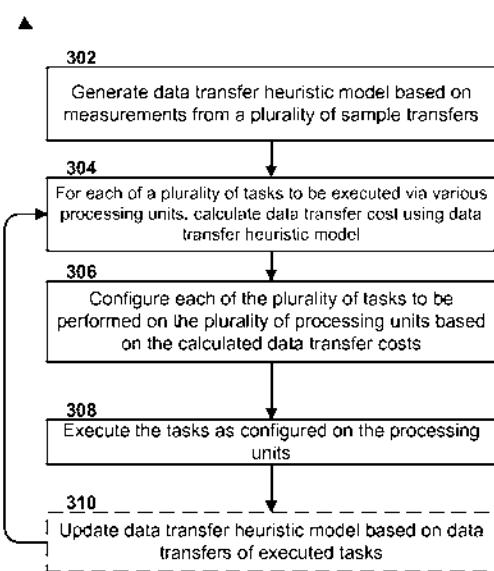


FIG. 3

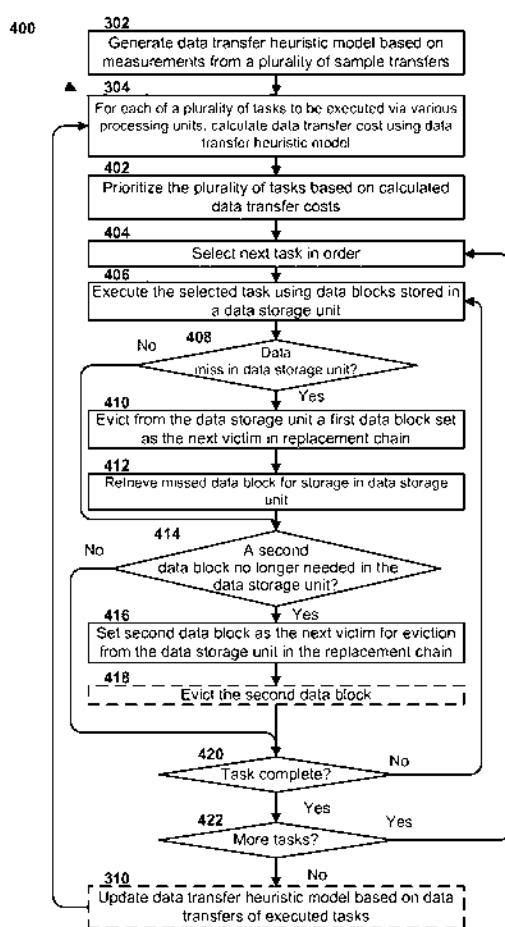


FIG. 4

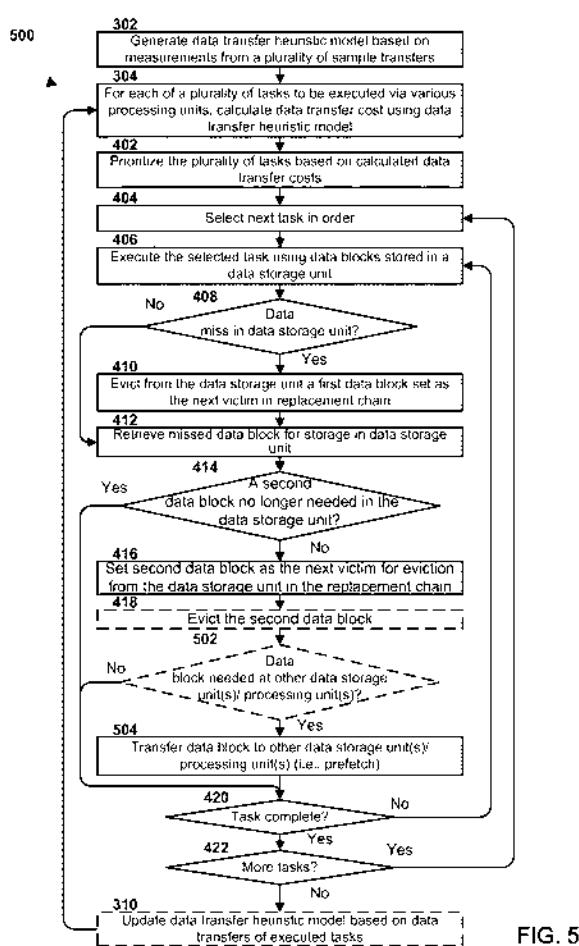


FIG. 5

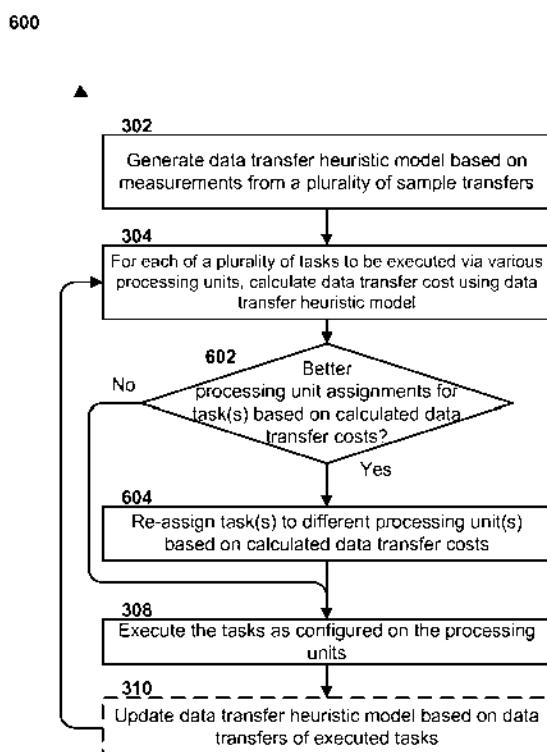


FIG. 6

700

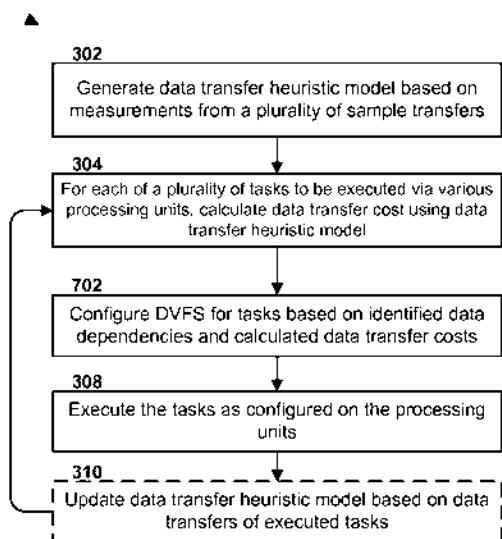


FIG. 7

U.S. Patent

Aug. 15, 2017

Sheet 8 of 8

US 9,733,978 B2

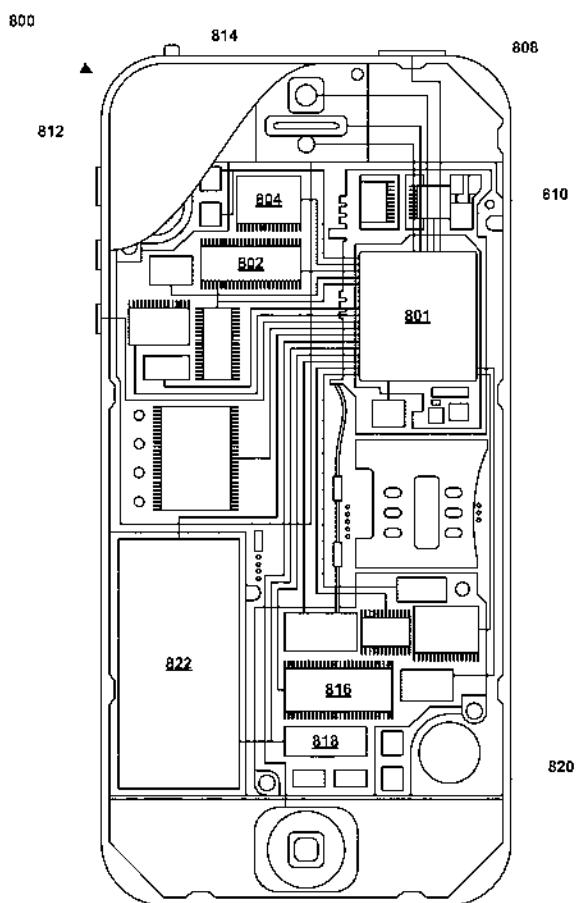


FIG. 8

1

2

DATA MANAGEMENT FOR MULTIPLE PROCESSING UNITS USING DATA TRANSFER COSTS**BACKGROUND**

In an ideal multi-core multi-processor or heterogeneous system, different included processing units are connected to the same cache hierarchy under the same memory space. However, some multi-processor computing systems may provide coherence only in part of the systems, so implementing full coherence schemes is often very costly. For example, in a system having a central processing unit (CPU), a graphical processing unit (GPU), and a digital signal processor (DSP), only the CPU and GPU may utilize coherence. In many typical systems, each processing unit utilizes its own cache hierarchy and memory space. For certain tasks executed by the processing units may be transferred between various memory units in order to enable particular processing units to perform the associated tasks. For example, a GPU may utilize data within a first data store for a first task, a GPU may use data within the first data store and a third data store for a second task, a DSP may use data within the first data store and a second data store for a third task, and the CPU may use data within the first data store and a fourth data store for a fourth task. As each data transfer within a multi-core computing system can require a costly cache flush that includes writes to memory, transferring data for use by different tasks and/or to offload work to various cores often incurs significant overhead costs.

SUMMARY

Various embodiments provide methods, devices, systems, and non-transitory processor-readable storage media for data management in a computing device utilizing a plurality of processing units (i.e., a multi-processor computing device). An embodiment method performed by a multi-processor computing device may include operations for generating a data transfer heuristic model based on measurements from a plurality of sample data transfers between a plurality of data storage units, calculating data transfer costs for each of a plurality of tasks using the generated data transfer heuristic model, and configuring the plurality of tasks to be performed on the plurality of processing units based at least on the calculated data transfer costs. At some embodiments, the measurements may include one of time measurements, power consumption measurements, or time and power consumption measurements.

In some embodiments, generating the data transfer heuristic model based on the measurements from the plurality of sample data transfers between the plurality of data storage units may include adjusting a coefficient of a formula that calculates an estimated cost. In some embodiments, the formula may calculate one of an estimated time cost, an estimated power consumption cost, or both an estimated time cost and an estimated power consumption cost.

In some embodiments, the method may further include operations for executing each of the plurality of tasks as configured on one of the plurality of processing units, and updating the data transfer heuristic model based on measurements of data transfers occurring during executions of the plurality of tasks. In some embodiments, generating the data transfer heuristic model may occur at a boot-up time for the computing device or periodically. In some embodiments,

generating the data transfer heuristic model may include using embedded power monitoring capabilities of the computing device.

In some embodiments, calculating the data transfer costs for each of the plurality of tasks using the generated data transfer heuristic model may include querying the data transfer heuristic model using a data transfer size, a source identity, and a destination identity. In some embodiments, querying the data transfer heuristic model may be performed via an application programming interface (API) call. In some embodiments, calculating the data transfer costs for each of the plurality of tasks using the generated data transfer heuristic model may include identifying data dependencies between the plurality of tasks. In some embodiments, the data dependencies may be provided via a scheduler.

In some embodiments, configuring the plurality of tasks to be performed on the plurality of processing units based at least on the calculated data transfer costs may include prioritizing the plurality of tasks based on the calculated data transfer costs to obtain a scheduling priority order. In some embodiments, the method may further include operations for executing a first task in the scheduling priority order on a first processing unit, determining whether a first data block within a first data storage unit associated with the first processing unit is no longer needed during execution of the first task, and evicting the first data block from the first data storage unit in response to determining that the first data block is no longer needed during the execution of the first task. In some embodiments, determining whether the first data block within the first data storage unit associated with the first processing unit is no longer needed during execution of the first task may be based on compiler information, an application programming interface (API) call within code of the first task, or both.

In some embodiments, evicting the first data block from the first data storage unit in response to determining that the first data block is no longer needed during the execution of the first task may include setting the first data block as a next victim data block in a replacement chain of a cache replacement policy associated with the first processing unit, detecting a data block miss corresponding to a second data block, and evicting the first data block from the first data storage unit in response to detecting the data block miss and based on the cache replacement policy. In some embodiments, the method may further include operations for determining whether the first data block is needed for performing a second task in the scheduling priority order, wherein the second task is performed by a second processing unit associated with a second data storage unit, and transferring the first data block to the second data storage unit in response to determining that the first data block is needed for performing the second task and upon evicting from the first data storage unit.

In some embodiments, evicting the first data block from the first data storage unit in response to determining that the first data block is no longer needed during the execution of the first task may include setting the first data block as a next victim data block in the first data storage unit, identifying that new driver-managed data is needed within the first data storage unit for use with the first task, determining whether there is insufficient storage space in the first data storage unit for the identified new driver-managed data, and evicting the first data block from the first data storage unit in response to determining that there is insufficient storage space for the new driver-managed data in the first data storage unit. In some embodiments, the first data storage unit may be a

driver-managed data structure associated with the first processing unit, a custom software-managed data store, or a custom hardware managed data store.

In some embodiments, configuring the plurality of tasks to be performed on the plurality of processing units based at least on the calculated data transfer costs may include determining whether there is a better processing unit for executing a first task based on the calculated data transfer costs, wherein the first task may be already configured to execute on a first processing unit, and re-assigning the first task to be executed by a second processing unit in response to determining that there is a better processing unit. In some embodiments, configuring the plurality of tasks to be performed on the plurality of processing units may include configuring dynamic voltage and frequency scaling (DVFS) settings for processing units executing each of the plurality of tasks based on the calculated data transfer costs. In some embodiments, the DVFS settings control a use of a processing unit, a use of a bus, or a combination of both.

Further embodiments include a computing device configured with processor-executable instructions for performing operations of the methods described above. Further embodiments include a non-transitory processor-readable medium on which is stored processor-executable instructions configured to cause a computing device to perform operations of the methods described above.

BRIEF DESCRIPTION OF THE DRAWINGS

The accompanying drawings, which are incorporated herein and constitute part of this specification, illustrate exemplary embodiments, and together with the general description given above and the detailed description given below, serve to explain the features of the claims.

FIG. 1 is a component block diagram illustrating a plurality of processing units and data storage units in an exemplary computing device that is suitable for use in various embodiments.

FIG. 2A is a diagram illustrating dependences of tasks configured to utilize data within various data storage units while being executed by various processing units of an exemplary computing device that is suitable for use in various embodiments.

FIG. 2B is a component block diagram illustrating data storage access required to execute a plurality of tasks by a plurality of processing units of an exemplary computing device that is suitable for use in various embodiments.

FIG. 3 is a process flow diagram illustrating an embodiment method performed by a computing device to calculate data transfer costs associated with a plurality of tasks to be executed on a plurality of processing units.

FIG. 4 is a process flow diagram illustrating an embodiment method performed by a computing device to configure a data block for early eviction from data storage.

FIG. 5 is a process flow diagram illustrating an embodiment method performed by a computing device to transfer a data block used in a first task to a data storage for use in a second task to e.g., pre-fetch.

FIG. 6 is a process flow diagram illustrating an embodiment method performed by a computing device to re-assign tasks to various processing units based on calculated data transfer costs.

FIG. 7 is a process flow diagram illustrating an embodiment method performed by a computing device to configure voltage frequency settings used with regard to tasks based at least on calculated data transfer costs.

FIG. 8 is a component block diagram of a computing device suitable for use in some embodiments.

DETAILED DESCRIPTION

The various embodiments will be described in detail with reference to the accompanying drawings. Whenever possible, the same reference numbers will be used throughout the drawings to refer to the same or like parts. References made to particular examples and implementations are for illustrative purposes, and are not intended to limit the scope of the embodiments or the claims.

The word "exemplary" is used herein to mean "serving as an example, instance, or illustration." Any implementation described herein as "exemplary" is not necessarily to be construed as preferred or advantageous over other implementations.

The term "computing device" is used herein to refer to an electronic device equipped with at least a processor. Examples of computing devices may include mobile devices (e.g., cellular telephones, wearable devices, smart-phones, web-pads, tablet computers, Internet enabled cellular telephones, Wi-Fi enabled electronic devices, personal data assistants (PDAs), laptop computers, etc.), personal computers, and server computing devices. In various embodiments, computing devices may be configured with various memory and/or data storage as well as networking capabilities (such as network transceivers) and antenna(s) configured to establish a wide area network (WAN) connection (e.g., a cellular network connection, etc.) and/or a local area network (LAN) connection (e.g., a wired/wireless connection to the Internet via a Wi-Fi router, etc.).

The terms "multi-core computing device", "heterogeneous computing device", and "multi-processor computing device" are used interchangeably herein to refer to a computing device configured to execute various tasks (e.g., routines, instruction sets, etc.) with two or more processing units or devices, such as application processors (e.g., a CPU or specialized processing devices (e.g., a GPU, a DSP, etc.)). For example, a heterogeneous computing device may be a multi-processor computing device (e.g., a system-on-chip (SoC)) with different processing units configured to perform various operations, such as executing routines, tasks, etc. Such multi-processor computing devices may be configured with various data storage (e.g., caches), memory (e.g., storage medium units) connected via wired/wireless connections, etc.). An exemplary multi-processor computing device is illustrated in FIG. 1.

The terms "data storage units" and "data stores" are used herein to refer to various devices, portions, locations of memory, caches, and/or other components of a computing device configured to store data for use by one or more processing units of a multi-processor computing device. For example, data storage units may include distinct volatile or non-volatile storage components that are statically or dynamically associated with a processing unit (e.g., a core, a CPU, etc.). As another example, a data storage unit may correspond to a range of memory addresses that are associated with a particular core of a SoC or alternatively a storage area within system memory. In some embodiments, data storage units may include shared memory accessible by various processing units.

The terms "buffers" and "buffer data" are used herein to refer to abstract program representations of contiguous program data. Buffer data is stored within a data storage unit. Buffer data may be transferred between data storage units so that different processing units accessing the data storage

units may utilize the buffer data. Such buffer data transfers may include transfers of entire buffers or portions of buffers. The terms "buffer" and "buffer data" are used herein for illustrative purposes and are not intended to limit the embodiments or claims to any particular type of data that may be stored within various data storage units of a multi-processor computing device.

To maintain high performance and satisfactory user experience, multi-processor computing systems require fast and low power energy data transfers between data storage used by the various processing units. For example, routines executing on a multi-processor mobile device may require quick data exchanges between any combination of the processing units participating in the execution of the routines.

Various embodiments provide methods, devices, systems, and non-transitory, process-readable storage media for improving data management and buffer utilization in multi-processor computing devices by using programmer and/or runtime data. In particular, a multi-processor computing device may be configured to utilize available information of tasks and the system's capabilities in order to estimate the likely costs for transferring data between data storage units accessed by processing units.

For example, based on scheduler data indicating the dependencies of various tasks to certain buffer data, the multi-processor computing device may calculate an estimated time cost and/or estimated power/energy consumption cost for transferring data of a buffer to the cache of a GPU¹. As another example, via an application programming interface (API) call, the multi-processor computing device may receive user-provided data indicating the data size and/or criticality (or urgency) of tasks to be performed on a first processing unit (e.g., CPU) and determine the data transfer cost for the data of the particular size to be moved from a dedicated memory unit to a processor cache unit. As another example, when the user indicates that a task may only be executed by certain processing units, the multi-processor computing device may calculate required data transfer costs to retrieve the data for use with those processing units.

With such data transfer costs, the multi-processor computing device may also perform various organizational or prioritization operations to improve the efficiency of the execution of the tasks, such as by sorting (or ordering) the tasks and strategically flushing data. For example, based on data transfer costs of a first task and a second task to be performed on a first processing unit, the multi-processor computing device may schedule the first task to execute first due to having a lower estimated time and/or power/energy consumption for associated data transfers. In general, the multi-processor computing device may utilize various scheduling schemes in order to prioritize task executions based on data transfer costs. For example, a scheduler used by the multi-processor computing device may use a sorted queue, distributed queues, scheduling graphs, and/or any other scheduling abstraction configured to organize, sort, and/or otherwise prioritize tasks. As another example, the multi-processor computing device may prioritize by creating multiple independent queues with potential synchronization between the queues (i.e., distributed scheduling). Prioritization based on estimations of data transfer costs required for performing tasks may enable the multi-processor computing device to improve efficiency of task executions with regard to data transfers and thus reduce unnecessary flushing and other drawbacks that may be present in systems lacking coherency.

In various embodiments, the multi-processor computing device may generate heuristic models referred to herein as "data transfer heuristic models," that may be used to calculate costs of transferring data between various data storage units of the system and processing units. Such a heuristic approach may utilize measurements made over time of power/energy consumption and/or time for sample transfers between various data storage units of the system to create data that may be used for querying in real time to estimate costs for data transfers needed to perform tasks on various processing units. For example, using a known buffer size (e.g., provided by programmer via an API call) and embedded power measurement capabilities, the multi-processor computing device may compute a simple model of a transfer by (1) measuring delay and estimated power resulting from transferring data of various sizes (e.g., a minimum size, 11, 12, and 13 cache blocks, and page sizes, etc.) between all possible combinations of storage (e.g., core caches, GPU memory, system memory, etc.) and then (2) adjusting coefficients used in heuristic models. In some embodiments, the data transfer heuristic models may utilize formulas that may be updated over time.

For example, an embodiment linear formula used to estimate latency for transferring a certain data block size in between data storage units (e.g., cost for transferring cache lines) between caches of processing units, etc. may be represented as:

$$\text{latency} = \text{lat}_0 + \text{lat}_1 \cdot \text{size} + \text{lat}_2 \cdot \text{size}^2$$

wherein "size" may represent a data size for transfer, "latency_0" may represent an overhead cost and "latency_cost" may represent a coefficient generated based on sample transfers. As another example, an embodiment formula used to estimate power/energy consumption for transferring a certain data block size in between two data storage units (or processing unit caches) may be represented as:

$$\text{power} = \text{power_0} + \text{power_1} \cdot \text{size} + \text{power_2} \cdot \text{size}^2$$

wherein "size" may represent a data size for transfer, "power_0" may represent an overhead cost and "power_overhead" and "power_coeff_2" may represent a coefficient generated based on sample transfers. In some embodiments, there may be multiple implementations of such formulas in linear and quadratic form, where each formula may apply only to specific ranges of buffer sizes and to a specific pair of source and destination data storage units. Further, there may be a lookup table of coefficients enabling the multi-processor computing device to perform a table lookup based on the range of data transfer sizes involved and the data storage units involved (e.g., source, destination).

In some embodiments, data transfer heuristic models may be computed offline with an external power meter, which may be useful for multi-processor computing devices (e.g., the processing units therein) that do not have embedded power measurements. In some embodiments, such data transfer heuristic models may be calculated when the multi-processor computing device is started booted-up, during application execution for runtime, and/or on a periodic basis (e.g., every other week, etc.). In some embodiments, more complex data transfer heuristic models may be generated that take into the account the temperature and packaging of the multi-processor computing device. In some embodiments, the multi-processor computing device may obtain measurements (e.g., temperatures, etc.) using embedded power monitoring capabilities in order to provide some data used for generating data transfer heuristic models.

In some embodiments, data transfer heuristic inside data may be exposed to a user-space for use by a runtime functionality and/or made available by an operating system (OS) via an application programming interface (API), such as via API calls that take the size of a data set to be sent, a source identity of a source data storage unit, and a destination identity of a destination data storage unit as inputs and then return a value representing the cost of the transfer. For example, user-generated code for a particular task may include an API call for estimating a data transfer cost of moving a data set of a certain size from the cache of a first processing unit (e.g., a CPU) to the cache of a second processing unit (e.g., a GPU), a system memory, and/or other data storage unit.

The multi-processor computing device may configure how and when various tasks are to be performed using the calculated data transfer costs in combination with information from programmers and/or a scheduler indicating the state of data storage units and/or future operations. In particular, the multi-processor computing device may configure the order of execution or priority for each task to improve efficiency with regard to costly flushes that may be required when transferring data. For example, based on calculated data transfer costs of buffer data between data storage units, the multi-processor computing device may determine a prioritized order for performing all tasks on one or more cores such that data transfer costs and flushing operations may be minimized or at least reduced compared to costs incurred with a default scheduling order that does not incorporate data transfer costs.

In some embodiments, based on data transfer cost estimates, the multi-processor computing device may configure (or assign) tasks to execute on particular processing unit such that associated data transfer costs are reduced or minimized. For example, when the code for a task is available for multiple cores, the costs of buffer data transferred between data storage units accessible to the cores may be used to select a certain core for executing the task such that the transfer of buffer data to an associated data storage unit would incur the least time and/or power requirements/consumption first. In some embodiments, when a task graph is known at compile time, a core may be selected at compile time.

Typically, when experiencing a cache miss while executing a first task, a cache may evict blocks associated with another task to make new insertions for the first task. Such actions can be costly, and may impair the runtime of the second task. To improve this scenario, in some embodiments, the multi-processor computing device may update a cache replacement policy when detecting a last use of a data segment within a data storage unit (e.g., a cache of a core, etc.) corresponding to a current task. For example, with compiler support or at the end of task execution, the multi-processor computing device may proactively change the position of a data block in the replacement chain to ensure an early eviction of buffer data (preventive action), as that data block may no longer be needed for subsequent tasks or processing units accessing that data. In other words, compiler analysis may be used to determine use-lifetimes for data blocks used by tasks. This may allow data segments to be flushed early, and this may reduce waiting times while flushing is completed due to data transfers between data storage units. In some embodiments, such proactive flushing operations may require per-block cache block flushing capabilities, which may be provided by specialized hardware. In some embodiments, programmers may provide information, such as via API call within task code, that

indicates of runtime whether a task will be the last to use a data segment stored in a data storage unit, allowing early eviction if so. For example, a programmer may insert API calls into a task's code to indicate when certain ranges of data (or data blocks) are no longer needed by the task (e.g., a call such as "data_block_finished(start, size)", etc.), thus allowing the multi-processor computing device to conduct early eviction of the ranges of data. Such API calls may be provided by a runtime system, such as Multicore Asynchronous Runtime Environment (MARE).

In some embodiments, when the multi-processor computing device has detected the last access of a particular data segment of a data storage unit (e.g., a data block) by a processing unit executing a task, the multi-processor computing device may evict the data segment and transfer the data segment to another data storage unit for use with another task processing unit. In such an "evict and push" manner, the data segment may be allocated at the beginning of the execution of the other task (or "pre-fetched" for the subsequent task processing unit). Based on the type of data storage unit usage (e.g., read-only); and the capabilities of the processing units, the multi-processor computing device may also migrate block data to multiple processing units and/or associated data storage (e.g., caches).

In some embodiments, a multi-processor computing device may use scheduler programmer information to configure tasks for execution by dynamically adjusting voltage frequency scaling (e.g., via dynamic voltage and frequency scaling (DVFS) settings), such as to accommodate the urgency criticality of different tasks. For example, with an urgency criticality of a task (that is based on the number of successive tasks, extra information from the programmer, and/or a calculated data transfer cost), the multi-processor computing device may configure a task with no data dependencies to successor tasks to use a lower frequency to save power, while configuring another task with multiple successors to use a higher frequency. Such configurations may address both core DVFS settings (e.g., adjustments to processing unit frequency used to execute tasks) and bus DVFS settings (e.g., adjustments to the speed of executing a data transfer). For example, bus DVFS settings may be adjusted by the multi-processor computing device based on task scheduler data indicating whether there will be a delay before a task can start execution due to data transfers and/or dependencies. Various conditions may be evaluated for adjusting DVFS settings, such as whether a task is be waiting for a suitable core to execute and/or whether a task is waiting for task dependencies to be satisfied.

The embodiment techniques may be beneficially used with both cache-based memory data stores (e.g., cache blocks, etc.) and driver-managed data stores (e.g., OpenGL buffers, etc.). For example, computing device implementing embodiment methods of the various embodiments may be configured to apply embodiment eviction, pre-fetch, and/or task-scheduling-order operations to supported OpenGL buffers. Non-limiting examples of such driver-managed data stores may include various custom software (SW)-managed data stores and custom hardware (HW)-managed data stores (e.g., OpenGL, OpenGL, ION memory, etc.). Such driver-managed data stores may be accessed by specialized devices, such as GPUs, DSPs and/or custom accelerators that may not utilize caches. Similar to the use of replacement chains with cache-based memory stores, driver-managed data stores may use driver calls and custom software or hardware data store managers when implementing some embodiment techniques.

The embodiment techniques provide detailed calculations for identifying data transfer costs within multi-processor computing systems, such as heterogeneous multi-core systems. For example, embodiment calculations for estimating latency power (energy) consumption for transferring data from a first data storage unit to a second data storage unit may utilize particular coefficient values of a data transfer heuristic model that may be accessed via software, routines, etc. In other words, instead of calculating overall computation time for all computations of tasks, the embodiment calculations may estimate the data transfer-specific costs (e.g., time and/or power/energy) consumption required by processing units when executing the tasks. For example, the embodiment techniques may not provide an overall calculation of the various costs required to execute a task (e.g., cycles, execution time, etc.), but instead may only provide accurate projections for the costs for transferring data between one or more data storage units of the system in order to execute the task.

The embodiment techniques enable software (e.g., runtime functionalities, client applications, schedulers, etc.) access to data transfer cost estimates that may be used to improve the efficiency of task performance. For example, the embodiment techniques may expose information to the operating system (OS) or applications allowing code utilizing API calls to determine the data transfer costs that may likely be required for executing a particular task on any of the available processing units of a multi-processor computing device. Further, unlike some conventional techniques that may utilize a compiler and/or a kernel profiler to match workloads to processes, the embodiment techniques may utilize data-based, heuristic models that provide time and/or power (energy) consumption estimates for data transfers based on limited inputs. For example, the time required for moving data needed by a task may be determined with an API query that includes only a size of data to transfer, a source identity of a source data storage unit, and a destination identity of a destination data storage unit.

In some embodiments, the multi-processor computing device may execute a dedicated runtime functionality (e.g., a runtime service, routine, or other software element, etc.) to perform various operations for generating data transfer cost information or estimates as described herein. For example, the runtime functionality may be configured to take into consideration the topology of the computing device and/or cache properties (e.g., size, associativity, atomic granularity, etc.). In some embodiments, the runtime functionality may be an implementation of Qualcomm's Minimum Asynchronous Runtime Environment (MARE). For example, the runtime functionality may be a MARE functionality that is further configured to identify the power (energy) consumption and/or time likely required to transfer data between data storage units utilized by heterogeneous cores of an SoC.

The embodiment techniques may be implemented in any computing device having more than one processing unit. For example, embodiment methods may be performed by a laptop computer having a CPU and a GPU, a smartphone-type mobile device having multiple cores and/or processors, and/or a coprocessor with a cache interface. For simplicity, the descriptions herein may relate to multi-processor mobile devices; however, the embodiments and claims are not intended to be limited to any particular type of computing device with a plurality of processing units.

FIG. 1 illustrates a plurality of processing units 102a-102n (e.g., CPU, GPU, DSP, etc.) and data storage units 104a-104n in an exemplary heterogeneous computing device (e.g., a multi-processor mobile device 110) suitable

for use with the various embodiments. The mobile device 110 may be configured to execute various applications, threads, software, routines, instruction sets, tasks, and/or other operations via one or more of the processing units 102a-102n. For example, concurrent with the execution of various operations as associated with an operating system, the mobile device 110 may be configured to execute an embodiment runtime functionality via an application processor or CPU (e.g., processing unit 102a). Buffer data used by or otherwise associated with the operations performed on the various processing units 102a-102n may be stored within and retrieved from the various data storage units 104a-104n managed in the mobile device 110. For example, instructions, variable data (e.g., measurements, data, user input data, etc.), and other information used by routines in applications may be stored as buffers (or buffer data stored within the data storage units 104a-104n) for use by tasks executing on the processing units 102a-102n.

In some embodiments, the data storage units 104a-104n may be discrete memory devices or different memory spaces. In some embodiments, the data storage units 104a-104n may be configurations or non-contiguous sections in a shared memory space of a memory unit 105. Such ranges of addresses or blocks within a unified memory device (e.g., a cache, RAM, etc.). In some embodiments, each of the data storage units 104a-104n may be directly associated with one of the processing units 102a-102n. For example, the first data storage unit 104a may be associated with the first processing unit 102a (e.g., CPU), the second data storage unit 104b may be associated with the second processing unit 102b (e.g., GPU), and the third data storage unit 104c may be associated with the third processing unit 102c (e.g., DSP).

In some embodiments, the processing units 102a-102n may access and/or otherwise utilize one or more data storage devices 116 for storing data used in association with executing various tasks. For example, the processing units 102a-102n may retrieve data from a main or system memory device (e.g., RAM), a cache unit(s), and/or an external data storage device (e.g., a memory device connected to the mobile device 110 via a universal serial bus (USB) connection and/or a wireless connection (e.g., Bluetooth®, Wi-Fi®, etc.).

FIG. 2A is a conceptual diagram 200 illustrating dependencies of exemplary tasks 201-206 (i.e., tasks-X-J) that are configured to be executed on various processing units (e.g., CPU, GPU, DSP) of a multi-processor computing device (e.g., mobile device 110 of FIG. 1). In general, tasks executed by the various processing units of a multi-processor computing device may utilize buffer data stored within various data storage units (e.g., buffer data within memory units, etc.). In order for the respective processing units (and thus the tasks) to access required data, the data may need to be transferred in between data storage units requiring time and/or power (energy) consumption. In order to efficiently estimate data transfer costs for transferring data needed for performing the tasks, some embodiment techniques may identify data dependencies of tasks with regard to processing units and/or the locations of data. For example, a scheduler or runtime functionality may identify that a first and second task may both require data of a first buffer stored within a first data storage unit. In some embodiments, data transfer costs (e.g., times, etc.) may indicate costs associated with maintaining cache coherency when multiple tasks are performed in sequence. For example, the cost to transfer data for use by a second task may include not only a data transfer time, but also a time estimate based on the time to complete

the use of the data by a first task. Performing such a time estimation may cause the multi-processor computing device to go through several branches of the task tree, including both communication and computing times.

Each of the different exemplary tasks 201-206 of the diagram 200 are illustrated as configured to utilize data of one or more buffers (i.e., buffers 1-4) stored within various data storage units. For example, a first task 201 ("A") may utilize data of buffer "1", a second task 202 ("B") may utilize data of buffers "1" and "3", a third task 203 ("C") may utilize data of buffers "1" and "5", a fourth task 204 ("D") may utilize data of buffers "1" and "2", a fifth task 205 ("E") may utilize data of buffers "2", "3", and "4", and a sixth task 206 ("F") may utilize data of buffers "1" and "4". Based on information indicating data transfer costs with regard to time and/or power/energy consumption, as well as the overlapping use of buffer data stored within various data storage units, the multi-processor computing device (e.g., via a runtime functionality) may determine estimated data transfer costs for the various tasks and may further perform scheduling in other operations to improve the performance of the tasks.

For example, the multi-processor computing device may determine that buffer "1" is read-only with regard to tasks 202-204 based on runtime data (i.e., tasks 202-204 may not modify data of buffer "1"). Based on this, after the CPU executes the first task 201 that uses data of buffer "1", tasks 202-204 may be launched at the same time on the GPU, DSP, and CPU respectively. As another example, without knowing that buffer "2" is read-only with regard to tasks 202-204, a driver may be configured to flush buffer "1" from use by the CPU in order to enable the second task 202 on the GPU, and the third task 203 on the DSP to potentially modify the data of the buffer "1". As another example, the CPU may be configured to maintain the buffer "1" (e.g., keep in cache of the CPU) due to the multi-processor computing device determining that the sixth task 206 run on the CPU may require the data of the buffer "1".

FIG. 2B is a component diagram 250 illustrating exemplary access of buffer data (i.e., buffers 254a, 254b, 254c, 254d) by a plurality of processing units 252a-252c in order to execute the plurality of exemplary tasks 201-206 as described with reference to FIG. 2A. In particular, FIG. 2B illustrates that a first task 201, fourth task 204, and sixth task 206 may be assigned to execute on a CPU 252a, requiring data of a first buffer 254a and a fourth buffer 254b, a third task 203 may be assigned to execute on a DSP 252b, requiring data of the first buffer 254a and a second buffer 254b, and a second task 202 and a fifth task 205 may be assigned to execute on a GPU 252c, requiring data of the all of the buffers 254a-254d. In various embodiments, the buffers 254a-254d may be stored within various types of data storage units, such as caches directly associated with any of the processing units 252a-252c, portions of shared memory, and/or other data storage devices coupled to the multi-processor computing device.

Embodiment techniques for generating data transfer heuristic models for determining data transfer costs between data storage units may be used in a number of applications to improve the overall functioning of a multi-processor computing device. For example, in a 4-core system, the multi-processor computing device (e.g., via a runtime functionality) may use a data transfer heuristic model to estimate the cost of moving buffer data from data storage units for use by a first core and a second core in order to identify the core on which a certain task might best be executed at a given

time. FIGS. 3-7 provide examples of various implementations that generate and utilize embodiment data transfer heuristic models.

FIG. 3 illustrates an embodiment method 300 performed by a multi-processor computing device to calculate data transfer costs associated with a plurality of tasks to be executed on a plurality of processing units. For each of the tasks available to be executed at a given time, the multi-processor computing device may identify data transfer costs using a pre-generated heuristic model. Further, based on the identified data transfer costs within the computing system (e.g., between caches, memories, SW/HW-managed data stores, etc.), the multi-processor computing device may perform various optimization operations (e.g., configurations of the tasks) to improve the execution efficiency of the tasks, such as prioritizing the tasks for execution in a certain sequence or executing queue or execution on certain processing units. Various operations of the method 300 may be performed by a runtime functionality executing via a processor or processing unit of a multi-processor computing device, such as the CPU 102a of the mobile device 110 described with reference to FIG. 1.

In block 302, the multi-processor computing device may generate a data transfer heuristic model based on measurements from a plurality of sample transfers between a plurality of data storage units. The multi-processor computing device may generate a data transfer heuristic model that may be used to predict or otherwise gauge the time and/or power/energy consumption costs for transferring certain sizes of data between certain data storage units of the computing device. Such measurements may include time measurements and/or power/energy consumption measurements and may be taken by the multi-processor computing device during a training period in which the multi-processor computing device obtains empirical data indicating latency and/or bandwidth, for various memories of the multi-processor computing device. In some embodiments, the measurements may further include overhead information accessible to the OS of the multi-processor computing device (e.g., overhead costs for buffer kernel modules, etc.). In addition to the sample transfers, in some embodiments the data transfer cost model may also be based on measurements taken during real transfers associated with program execution.

The data transfer heuristic model generated in block 302 may be based on simple transfers involving various types of data storage units, such as cache memory locations, system memory, and any other data storage devices associated with the processing units of the computing device. For example, the multi-processor computing device may perform a sample data transfer that simulates a data transfer from system memory to a GPU memory. In some embodiments, if the multi-processor computing device does not have embedded power measurements, the computing device may generate the model offline with an external power meter.

In some embodiments, the data transfer heuristic model may be generated in block 302 when the computing device starts up (or at a boot-up timer), restarts or reloads the operating system, and/or on a periodic basis (e.g., at repeatedly at the expiration of a predefined time interval, such as every other week, etc.). In various embodiments, the data transfer heuristic model may be generated by the operating system (OS) of the multi-processor computing device, the runtime functionality, and/or other functionalities of the multi-processor computing device (e.g., an extensible firmware interface (EFI) between an operating system and firmware, etc.).

In various embodiments, the data transfer heuristic model may include formula, equations, and/or other calculation

frameworks that utilize coefficients that are set based on measurements of the sample transfers. For example, subsequent to the generation of the data transfer heuristic model, the multi-processor computing device may estimate data transfer costs of a certain data transfer (e.g., a certain data size from a certain source data storage unit to a certain destination data storage unit, etc.) based on calculations that use the set coefficients. In some embodiments, the coefficients may be estimated and set in response to and based on transferring a minimum size between all possible combinations of processing units of the computing device, transferring data of several sizes (e.g., 1, 1/2, and 1/3 cache blocks and page sizes, etc.), and storing measurements of the rate and consumed power needed for the sample transfers. In some embodiments, the multi-processor computing device may set and/or update such coefficients each time the data transfer heuristic model is generated (e.g., at startup, periodically, etc.). In some embodiments, the multi-processor computing device may utilize available embedded power measurement capabilities of the computing device to compute a simple model of a data transfer.

In some embodiments, the data transfer heuristic model may utilize a linear heuristic related to the time cost for latency of data transfers represented by the following formula:

$$\text{latency_rate} \times \text{size} + \text{lat_const}$$

wherein "size" represents the size of the data to be transferred, "sender" represents the processing unit (or associated data storage unit) currently using or storing the data to be transferred, "receiver" represents the destination processing unit (or associated data storage unit) that may receive the data to be transferred, and "X" represents a coefficient. The latency of the sender and the receiver may include the associated cost of reading and flushing when required data instructions from the memory hierarchy in order to send/receive the data instructions with the communication network. Such reading (and flushing when required) costs may or may not be fixed and determined by the micro architecture of the sender-receiver.

In some embodiments, the data transfer heuristic model may utilize a quadratic heuristic related to the power (energy) consumption cost of data transfers represented by the following formula:

$$\text{power_cost} \times \text{size}^2 + \text{power_const}$$

wherein "size" represents the size of the data to be transferred, "sender" represents the processing unit (or associated data storage unit) currently using or storing the data to be transferred, "receiver" represents the destination processing unit (or associated data storage unit) that may receive the data to be transferred, "X" represents a first coefficient, and "Y" represents a second coefficient. The quadratic formula for power may utilize a degree-2 polynomial because power is proportional to the square of the capacity that is proportional to the data storage unit size. In some embodiments, the coefficients (e.g., X or Y) may be computed per pair when the sender and receiver units are not similar in architecture (e.g., X sender, receiver). For example, if in a system with several equal cores, the cost of sending and receiving may be the same.

In some embodiments, the data transfer heuristic model may be generated and/or updated using other formulas and techniques, such as by accounting for effects to data transfers in the multi-processor computing device based at least

on various temperature within the multi-processor computing device, packaging/junction structures of the multi-processor computing device, transient failure rate, etc. For example, when the temperature of a receiver node is very high, its power can be multiplied by a factor proportional to the current temperature divided by the desired temperature of operation. In some embodiments, the multi-processor computing device may utilize embedded power monitoring capabilities in order to measure data used for generating the data transfer heuristic model, such as device temperature data.

In some embodiments, the operating system (OS) of the multi-processor computing device may expose the coefficients to a user-space of the multi-processor computing device so that various software, routines, etc. (e.g., the runtime functionality) may be able to directly access the data transfer heuristic model and current values for the coefficients. In some embodiments, the OS may expose a table with the measured values from the different tested data transfer sizes such that the runtime functionality may interpolate values within the table.

In some embodiments, when the coefficients are maintained by the OS but not exposed to user-space, the runtime functionality may be able to utilize various API calls to receive data indicating costs of data transfers based on the data transfer heuristic model (e.g., use an API call that takes the size of a buffer as input and returns a value representing the cost of a data transfer, etc.). Examples API calls may include get_latency_estimation(size, sender, receiver) and get_power_estimation(size, sender, receiver), wherein the "size" parameter may represent a data size to be transferred, "sender" represents the processing unit (or associated data storage unit) currently using or storing the data to be transferred, and "receiver" represents the destination processing unit (or associated data storage unit) that may receive the data to be transferred.

In some embodiments, the data transfer heuristic model may be generated such that particular data storage units, processing units, data sizes, and/or other transfer conditions may be preferred. In other words, certain data transfers may have data transfer costs estimated by the data transfer heuristic model that are more beneficial (e.g., lower in cost) than other data transfers. For example, internally, the data transfer heuristic model via the OS may not provide the actual cost of a particular data transfer from a first data storage unit to a second data storage unit, but instead may perform some adjustment to resulting cost estimates in order to provide estimates that favor (or show a smaller cost) of the data transfer from the first data storage unit to the second data storage unit over other possible transfers (e.g., from the first data storage unit to a third data storage unit, etc.). As another example, the data transfer heuristic model may be configured to always provide lower data transfer cost estimates for data transfers that involve a certain processing unit (e.g., GPU, etc.). Such adjustments to the data transfer heuristic model estimates may be based on scheduler information, multi-processor computing device operating conditions, user preferences, and/or other conditions that may or may not be temporary. For example, when the OS has data indicating that the GPU is scheduled to perform a costly frame render in the near future, estimated data transfer costs that involve the GPU (e.g., as a transfer destination) prior to the frame render may be higher than normal.

At some time after generation of the data transfer heuristic model, the multi-processor computing device may begin estimating data transfer costs at runtime for a plurality of tasks. Thus, in block 304, the multi-processor computing

device may calculate a data transfer cost for each of a plurality of tasks to be executed on certain processing units using the data transfer heuristic model. Such estimated costs may not directly reflect the processing abilities (e.g., clock speeds, etc.) of the processing units executing the various tasks, but instead may indicate the data transfer-related costs, such as measurements related to time delay, power consumed, etc. For example, the multi-processor computing device may estimate or otherwise calculate transfer times and power/energy consumption for each of the tasks based on querying the data transfer heuristic model using an application programming interface (API) call that indicates a source identity (i.e., a source data storage unit for particular data required by the tasks), a destination identity (i.e., a destination data storage unit that may receive the particular data), and the size of the particular data to be transferred (or a data transfer size). In some embodiments, such API calls may be invoked from the tasks' codes. In some embodiments, the multi-processor computing device may also evaluate task dependencies to compute the data transfer costs of the tasks. For example, calculating data transfer costs may include identifying data dependencies between the plurality of tasks, wherein such data dependencies may be provided via a scheduler.

In block 306, the multi-processor computing device may configure each of the plurality of tasks to be performed on the plurality of processing units based on the calculated data transfer costs. In various embodiments, the multi-processor computing device via a scheduler may utilize any number of implementations to prioritize the plurality of tasks based on the calculated data transfer costs, such as sorting tasks in a single queue, creating multiple independent queues that may be synchronized (i.e., distributed scheduling), creating a scheduling graph, etc. For example, such configurations may include assigning specific tasks to particular processing units for execution, adjusting DVFS settings, and/or sorting or ordering tasks into a sorted queue. In block 308, the multi-processor computing device may execute the plurality of tasks in each task configured in block 306 on various processing units.

In optional block 310, the multi-processor computing device may update the data transfer heuristic model based on the data transfers of executed tasks. For example, the data transfer heuristic model may be updated when the effective cost of transferring data between data storage units is reduced based on optimizations and/or prioritizations using the data transfer calculations with the model (e.g., sorting, reassignments to different processing units, evicting and/or pre-fetching, etc.). Such updates may require adjustments to coefficients as described herein.

In some embodiments, the multi-processor computing device may adjust the data transfer heuristic model to utilize the following adjusted latency equation:

$$\text{latency} = \text{latency}(\text{size}, \text{latency}, \text{power}, \text{savings})$$

wherein "latency", "size", "sender", "receiver", "X" and "Y" are described herein, and "savings(sized)" represents a measured time savings.

In some embodiments, latency may include the cost of evicting the data from a data storage unit, such as the cost of evicting from cache memory when data has been modified and the cache line contains the single valid copy for the data in the system. Such an eviction cost may be proportional to the size of the data and may be computed as part of the $(X^* \text{size})$ term. In such a case, the term $\text{savings}(sized)$ may subtract the cost of the eviction that is included by default.

For example, during the computation of heuristics, the eviction cost may be computed and stored for later application via the above-indicated formula. In some embodiments, such savings may be different based on the position of the block within a list recently used (LRU) ordered list of cache blocks (e.g., list based on when cache blocks were last used). As an eviction may not be guaranteed, the heuristic may also take into account the LRU position as input. For example, the lower the position in the LRU (i.e., the more recently used), the bigger the savings (or reduction in latency) may be as the blocks may be expelled much faster than in a conventional case. In some embodiments, when the multi-processor computing device is configured to proactively expel and push data to other data storage units (e.g., pre-fetch), savings may include the reduced latency of the eviction as well as the reduced latency of the push. In some embodiments, when the latency of bringing the data to a recipient data storage unit may be known, that latency cost may be subtracted as part of calculating the cost of evicting the data from a data storage unit.

In some embodiments, the multi-processor computing device may adjust the data transfer heuristic model to utilize the following adjusted power equation:

$$\text{power} = \text{power}(\text{size}, \text{latency}, \text{power}, \text{savings})$$

wherein "power", "size", "sender", "receiver", "X" and "Y" are described herein, and "savings(sized)" represents a measured power (energy) consumption savings. In some embodiments, the savings may also include a square coefficient in the power equation. In some cases, assuming no DVFS is active, savings may correspond to a reduction of idle time in a processor. The power formula may not require the multi-processor computing device to wait until data is evicted or otherwise becomes available to calculate a power cost. In some embodiments, a boot-up power cost may be computed at a boot time and the "savings" term may be subtracted from the boot-up power cost.

The multi-processor computing device may repeat some of the operations of the method 300, such as by again calculating data transfer cost using data transfer heuristic model for each of a plurality of tasks to be executed via various processing units in block 304.

In some embodiments, the multi-processor computing device may re-calculate the data transfer cost for tasks in response to identifying new tasks to be executed by the multi-processor computing device. For example, the runtime functionality may refresh the estimated costs for all ready (or pending) tasks in response to a new task being scheduled for execution on an PLT. Alternatively, a new task order may be calculated in response to changes to the data transfer heuristic model. For example, in response to adjusting coefficients based on recently performed data transfers in association with an executing task, the multi-processor computing device may re-order the tasks for subsequent operations.

In some embodiments, prioritization operations may be used by the multi-processor computing device for setting the execution of various tasks based on data transfer costs (e.g., a scheduling priority order). In some cases, such prioritization (e.g., sorting, etc.) may enable ready (or pending) tasks that may utilize similar data sets to efficiently share data without incurring data coherence problems. For simplicity, FIGS. 4-5 illustrate such prioritization operations (e.g., operations of block 402) in combination with early eviction operations and/or pre-fetching operations for tasks that may utilize common data sets. However, in some

embodiments, such early evicting and/or pre-fetching operations may or may not require prioritization operations as described herein, and vice versa. For example, in some embodiments, the multi-processor computing device may prioritize (e.g., sort, etc.) various tasks executing on various processing units with or without also performing early eviction operations and/or pre-fetching operations.

FIG. 4 illustrates, in embodiment method 400, performed by a multi-processor computing device to configure a data block for early evicting from a data storage unit (e.g., a storage location for a buffer). The operations of the method 400 may be similar to those of the method 300 of FIG. 3, except that the method 400 may include operations for determining whether data blocks are to be evicted from a data storage unit (e.g., cache, etc.) to allow other immediately required data to be loaded in the memory location for use executing a task. In general, in systems that do not utilize coherency support, when a task finishes execution, a cache may need to be flushed to ensure data is accessible to other processing units and tasks. Accordingly, in some embodiments, the multi-processor computing device may be configured to execute an early flush of particular data segments to avoid stalling for other tasks processing units. For example, while still executing a first task on a processing unit, the multi-processor computing device may evict portions of a buffer after use with the first task so that a second task executing on another processing unit may begin using the evicted data without waiting (i.e., without delay due to memory subsystem action(s)). In some embodiments, API calls and/or compiler knowledge of a task may be used by the multi-processor computing device to indicate items that are no longer needed by the task and thus may be proactively flushed. With reference to FIGS. 1-4, various operations of the method 400 may be performed by a runtime functionality executing via processor in processing unit of a multi-processor computing device, such as the CPU 102 of the mobile device 110.

The operations of blocks 312-314, and 310 may be similar to the operations of like numbered blocks described above with reference to FIG. 3. In response to performing the selection operations in block 304, the multi-processor computing device may prioritize the plurality of the tasks based on the calculated data transfer costs in block 402. As described herein, various prioritization or scheduling schemes using data transfer costs may be implemented, such as using a sorted queue, distributed queues, scheduling graphs, and/or any other scheduling abstraction to organize, sort, and/or otherwise prioritize tasks. In particular, the multi-processor computing device, such as via a runtime functionality or a scheduler functionality, may determine an order or sequence (i.e., a scheduling priority order) for executing a plurality of tasks on one or more of the processing units based on the dependences of the tasks, the programmer-supplied data about the tasks (e.g., urgency), the state of the various data storage units, and/or the data transfer costs for moving required data set between various data storage units. For example, the computing device may identify the cost to transfer all needed data for each of a set of tasks ready to be performed, and may sort the tasks based on the identified costs (e.g., data transfer time and/or power/energy consumption but not native abilities of the processing units of the computing device). Then, for each available processing unit, the computing device may assign an unassigned task having the lowest cost based on the sorting to be launched on the processing unit. In this way, the multi-processor computing device may place the plurality of tasks in a sequence for execution on various processing units

and/or a timing order for concurrent execution on different processing units. In some embodiments, the multi-processor computing device may identify a sequence or scheduling priority order for executing various tasks such that tasks having a lower or the lowest estimated cost for data transfers (e.g., lowest time, power/energy consumption) may be performed before those tasks having higher estimated costs. In some embodiments, the multi-processor computing device may identify a sequence or scheduling priority order for executing various tasks such that tasks having a higher or the highest estimated cost for data transfers (e.g., highest time, power/energy consumption) may be performed before those tasks having lower estimated costs.

In block 404, the multi-processor computing device may select a next task in the order (i.e., the scheduling priority order), and in block 406, the multi-processor computing device may execute the selected task in a certain processing unit using data blocks stored in one or more data storage units associated with otherwise accessible by the certain processing unit (e.g., cache, shared memory, other data storage device, etc.). In some embodiments, the processing unit for executing the task may be determined based on data transfer costs, such as described with reference to FIG. 6.

While executing the task, the multi-processor computing device may determine whether there has (or will be) a data block miss with reference to an associated data storage unit in determination block 408. A data block miss may occur when the task requires a data segment that is not currently loaded or otherwise stored within the data storage unit (e.g., the processing unit's cache, etc.). In such a case, the processing unit executing the task may retrieve the missing data block from another data storage unit (e.g., local memory, from disk, from another section of addresses, etc.).

In response to determining that there has (or will be) a data block miss with reference to the associated data storage unit (i.e., determination block 408 "Yes"), the multi-processor computing device may evict from the data storage unit a first data block set as the next victim in replacement chain in block 410. For example, a cache may evict a predefined data block when a new data block from memory is required for an operation of the currently executing task. The replacement chain may be a list, pointer, or other data structure used by the multi-processor computing device for identifying the next data block to be removed. In some embodiments, the replacement chain may be controlled by a cache replacement policy for the processor unit.

In block 412, the multi-processor computing device may retrieve the missed data block for storage in the data storage unit, such as by loading the missed block from system memory, etc.

In response to determining that there has not been (or will not be) a data block miss with reference to the associated data storage unit (i.e., determination block 408 "No"), or in response to performing the retrieving operations in block 412, the multi-processor computing device may determine whether a data block (e.g., a second block) is no longer needed in the data storage unit (e.g., for the current task or other tasks using the data storage unit) in determination block 414. Such a determination may be made based on an analysis of the task's code, runtime data available, and/or by using compiler support. For example, determination of whether a first data block within the first data storage unit associated with a first processing unit is no longer needed during execution of the first task may be based on compiler information, an application programming interface (API) call set by a programmer within code of the first task, or

In response to determining that a second data block is no longer needed in the data storage unit (i.e., determination block 414 “Yes”), the multi-processor computing device may set the second data block as the next victim data block for eviction from the data storage unit in the replacement chain (e.g., of a cache replacement policy) in block 416. For example, with compiler support or at the end of task execution, the multi-processor computing device (e.g., via the runtime functionality) may be configured to change the position of the second data block in the replacement chain to ensure an early eviction. By early evicting the second data block, the multi-processor computing device may provide a preventive action that improves the waiting time another task may incur when a data transfer of the second data block is required. For example, another task may not have to wait for a flush of the second data block prior to a data transfer that provides the second data block for use with the another task. In some embodiments, the multi-processor computing device may include hardware that enables per address cache block flushing.

In some embodiments, when the data storage unit is a driver-managed data structure associated with the first processing unit, a custom software-managed data store, or a custom hardware-managed data store, eviction operations may include determining whether any new driver-managed data is needed within the data storage unit for use with a task. In such a case, the multi-processor computing device may evict a data block from the data storage unit in response to determining that there is insufficient storage space for the new driver-managed data in the data storage unit.

In optional block 418, the multi-processor computing device may evict the second data block, such as by immediately evicting the second data block. In response to determining that a second data block is still needed in the data storage unit (i.e., determination block 414 “No”), or in response to performing the eviction operations in optional block 418, the multi-processor computing device may determine whether the task is complete in determination block 420. In response to determining that the task is not complete (i.e., determination block 420 “No”), the multi-processor computing device may continue the execution operations in block 406.

In response to determining that the task is complete (i.e., determination block 420 “Yes”), the multi-processor computing device may determine whether there are more tasks to process in determination block 422, such as subsequent tasks to perform in the order identified based on the prioritization operations. In response to determining that there are more tasks to process (i.e., determination block 422 “Yes”), the multi-processor computing device may select a next task in order in block 404. In response to determining that there are no more tasks to process (i.e., determination block 422 “No”), the multi-processor computing device may update the transfer heuristic model based on the data transfers of executed tasks in optional block 310, and repeat some of the operations of the method 406, such as by again calculating data transfer cost using data transfer heuristic model for each of a plurality of tasks to be executed via various processing units in block 304.

FIG. 5 illustrates an embodiment method 500 performed by a multi-processor computing device to transmit a data block in between data storage units; and/or processing units) to pre-fetch data used by tasks. The operations of the method 500 may be similar to those of the method 300 (FIG. 3) and method 400 (FIG. 4), except that the method 500 may include operations 505 for performing pre-fetching to enable tasks associated with other data storage units to access data

is an efficient manner. For example, when the multi-processor computing device has decided a second task may be the next task to use a certain data block currently stored in a first data storage unit (e.g., with or without compiler support), the multi-processing computing device may evict the data block from the first data storage unit and send the data block to a second data storage unit so the second task may already have the data block allocated at the beginning of the execution of the second task. In some embodiments, the multi-processor computing device may also trigger the pre-fetching of required instructions since each task may be a “closure.” For example, the multi-processor computing device may also trigger the pre-fetch of required instructions as a task may be a “function closure,” such as with Qualcomm’s MURF, Open Computing Language (OpenCL), and in other APIs. In this way, other processing units executing tasks may avoid having to wait to populate associated data and instructions storage units, and instead may simply benefit from having pre-fetched data and instructions transferred once used by another processing unit or from memory. Various operations of the method 500 may be performed by a runtime functionality executing via a processor or processing unit of a multi-processor computing device, such as the CPU 102a of the mobile device 110 described with reference to FIG. 1.

The operations of blocks 302-304, 310 may be similar to the operations of like numbered blocks described above with reference to FIG. 3 and the operations of blocks 402-422 may be similar to the operations of like numbered blocks described above with reference to FIG. 4. In response to performing the eviction operations of optional block 418, the multi-processor computing device may determine whether the data block is needed at another data storage unit and/or processing unit in determination block 502. Such a determination may be made based on an analysis of the task’s code, using compiler support, and/or runtime data availability.

In response to determining that the data block is needed at other data storage units; and/or processing units (i.e., determination block 502 “Yes”), the multi-processor computing device may transfer data block to the other data storage unit(s) and/or processing unit(s) in block 504. In other words, the data block may be pre-fetched for use via the other data storage unit(s)/processing unit(s). In some embodiments, the multi-processor computing device may multicast the data block to more than one destination data storage unit. For example, based on the type of buffer usage for other processing units (e.g., read-only and/or the capabilities of the multi-processor computing device, the multi-processor computing device may multicast a particular data block to one or more data storage units for use with other tasks. In some embodiments, the multi-processor computing device may evaluate a task graph (e.g., the task graph of FIG. 2A) to identify eligible multicast recipients.

In response to determining that the data block is not needed at other data storage units and/or processing units (i.e., determination block 502 “No”), or in response to performing the operations of block 504, the multi-processor computing device may repeat the operations of blocks 404 through 422 until all tasks are complete (i.e., determination block 422 “No”) at which point the multi-processor computing device may update the transfer heuristic model based on the data transfers of executed tasks in optional block 310, and repeat some of the operations of the method 500, such as by again calculating data transfer cost using data transfer heuristic model for each of a plurality of tasks to be executed via various processing units in block 304.

FIG. 6 illustrates an embodiment method 600 performed by a multi-processor computing device to re-assign tasks to various processing units based on calculated data transfer costs. The operations of the method 600 may be similar to those of the method 300 of FIG. 3, except that the method 600 may include operations for re-assigning tasks originally associated with a first processing unit to another processing unit based on data transfer calculations. In general, some tasks may be general processing tasks configured to be executed on one of a plurality of the processing units of the multi-processor computing device (e.g., polymorphic tasks). At runtime for such a task, the multi-processor computing device may use the estimated data transfer costs for performing the tasks on all applicable processing units, such that the processing unit associated with the lowest calculated data transfer cost may be assigned the task. Various operations of the method 600 may be performed by a runtime functionality executing via a processor or processing unit of a multi-processor computing device, such as the CPU 102a of the mobile device 110 described with reference to FIG. 1.

The operations of blocks 302-304, 308-310 may be similar to the operations of like-numbered blocks described above with reference to FIG. 3. Using the data transfer costs calculated in block 304, the multi-processor computing device may determine whether there are better processing unit(s) for better processing unit assignments (or performing tasks) based on calculated data transfer costs in determination block 602.

In response to determining that there are better processing unit(s) for performing the task(s) based on the calculated data transfer costs (i.e., determination block 602 “Yes”), the multi-processor computing device may re-assign the task(s) to the different processing unit(s) based on the calculated data transfer costs in block 604. For example, the multi-processor computing device may move the task to another task queue. In some embodiments, device re-assignment may be performed statically at compile time or dynamically at runtime.

In response to determining that there are not better processing unit(s) for performing the task(s) based on the calculated data transfer costs (i.e., determination block 602 “No”), in using the re-assigned tasks determine in block 604, the multi-processor computing device may execute the tasks as configured on the processing units in block 608. The multi-processor computing device may update the transfer heuristic model based on the data transfers of executed tasks in optional block 310, and repeat some of the operations of the method 600, such as by again calculating data transfer cost using data transfer heuristic model for each of a plurality of tasks to be executed via various processing units in block 304.

In some embodiments, a task graph of polymorphic tasks may be known at compile time, and therefore the multi-processor computing device may select a processing unit to execute the task at compile time. In some embodiments, the multi-processor computing device may be configured to traverse a task queue of polymorphic tasks to select the order that may minimize the data transfer for the current tasks. The multi-processor computing device may be configured to support parallel processing environments or scenarios in which performance may be improved by dividing work items across multiple processing units and/or tasks in order to overlap computations. For example, multiple processing units/tasks may be used in performing parallel loop executions on different data simultaneously. Accordingly, in some embodiments, the multi-processor computing device may perform the operations in blocks 602-604 of the method 600

in order to select a processing unit for processing a task, configured for use in parallel processing of a cooperative processing effort by more than one task processing unit (e.g., a parallel loop).

In some embodiments, when the multi-processor computing device is configured to support work-stealing protocols in parallel processing efforts, tasks that have completed individually-assigned workloads may be configured to opportunistically steal work items from other tasks processing units. In such cases, the multi-processor computing device may further determine victim tasks from which another tasks may “steal” work items for processing based on the data transfer cost calculations. For example, the multi-processor computing device (e.g., via the runtime functionality) may calculate the costs for transferring data from various processing units participating in a cooperative processing effort in order to identify the victim task that may lose work items to another task processing unit. In some embodiments, the multi-processor computing device may also use the data transfer heuristic model to calculate how much data to steal from victim tasks processing units. For example, once a victim task is identified based on a first data transfer cost calculation, the multi-processor computing device may perform various other cost calculations for different amounts of work to be transferred to determine the maximum amount of work to steal without exceeding a data transfer cost threshold.

FIG. 7 is a process flow diagram illustrating an embodiment method performed by a multi-processor computing device to configure voltage/frequency settings used with regard to tasks based on calculated data transfer costs and task dependencies. The operations of the method 700 may be similar to those of the method 300 of FIG. 3, except that the method 700 may include operations for adjusting various DVFS settings associated with individual tasks. For example, based on the urgency/criticality of a task, the multi-processor computing device may configure the execution of the task to use a voltage or frequency that is lower or higher than a default setting. Various operations of the method 700 may be performed by a runtime functionality executing via a processor or processing unit of a multi-processor computing device, such as the CPU 102a of the mobile device 110 as described with reference to FIG. 1.

The operations of blocks 302 and 304 may be similar to the operations of like-numbered blocks of the method 300 described above with reference to FIG. 3. Using the data transfer costs calculated in block 304, the multi-processor computing device may configure DVFS settings for tasks based on identified data dependencies and calculated data transfer costs in block 702. For example, when a particular task has no dependencies (i.e., no other tasks are reliant upon the data currently associated with, stored at, being processed by the task), the multi-processor computing device may configure DVFS settings for the task to be low, allowing data to be transmitted to a data storage unit used by the task to utilize a slower frequency that may save power/resource consumption. As another example, when a particular task has one or more dependencies (i.e., successor tasks), the multi-processor computing device may configure DVFS settings for the task to be high, allowing the task to receive data faster, enabling faster turnaround for the successor tasks. DVFS settings may control bus and/or core use (e.g., speed of task execution on a core, etc.) based on data transfer costs.

As conventional DVFS algorithms/protocols may be “blind” with regard to predictive data indicating future requirements of processors, in some embodiments, the

urgency and/or criticality of each task may be determined based on an identified number of successes for the task and/or additional information from the programmer, such as provided within code of the task, an API call, etc. In some embodiments, the multi-processor computing device may select the DVIIS settings for a plurality of tasks such that data transfer times are normalized for each. For example, if several state sets have to be emulated by several processing units/devices and there is a global synchronization element, the synchronization element may prevent any processing units/devices from continuing until all processing units/devices have reached the element, and so the multi-processor computing device (e.g., via the naming functionality) may calculate DVIIS settings for each that provide equalized arrival times to all, minimizing waiting time and power usage. In some embodiments, the generation of data transfer heuristic model data in block 302 may include estimations of voltage and/or frequency, such as provided as inputs for sample transfers, etc.

With the DVFS configured to tasks in block 702, multi-processor computing device may execute the tasks as configured on the processing units listed in block 306. The multi-processor computing device may update the transfer heuristic model based on the data transfers of executed tasks in optional block 310, and repeat some of the operations of the method 600, such as by again calculating data transfer cost using data transfer heuristic model for each of a plurality of tasks to be executed via various processing units in block 304.

Various forms of computing devices, including personal computers, mobile devices, and laptop computers, may be used to implement the various embodiments. Such computing devices may typically include the components illustrated in FIG. 8 which illustrates an example multi-processor mobile device 800. In various embodiments, the mobile device 800 may include a processor 801 coupled to a touch screen controller 804 and an internal memory 802. The processor 801 may be one or more multi-core if designated for general or specific processing tasks. In some embodiments, other processing units may also be included and coupled to the processor 801. The internal memory 802 may be volatile and/or non-volatile memory, and may also be secure and/or encrypted memory, or insecure and/or unencrypted memory, or any combination thereof. The touch screen controller 804 and the processor 801 may also be coupled to a touch screen panel 812, such as a resistive-sensing touch screen, capacitive-sensing touch screen, infrared sensing touch screen, etc. The mobile device 800 may have one or more radio signal transceivers 808 (e.g., Bluetooth®, ZigBee®, Wi-Fi®, radio frequency (RF) radio, etc.) and antennae 810, for sending and receiving, coupled to each other and to the processor 801. The transceivers 808 and antennae 810 may be used with the above-mentioned circuitry to implement the various wireless transmission protocol stacks and interfaces. The mobile device 800 may include a cellular network wireless module chip 816 that enables communication via a cellular network and is coupled to the processor. The mobile device 800 may include a peripheral device connection interface 818 coupled to the processor 801. The peripheral device connection interface 818 may be singularly configured to accept one type of connection, or multiply configured to accept various types of physical and communication connections, common or proprietary, such as USB, FireWire, Thunderbolt, or PCIe. The peripheral device connection interface 818 may also be coupled to a similarly configured peripheral device connection port (not shown). The mobile device 800 may also

include speakers 814 for providing audio outputs. The mobile device 800 may also include a housing 820, constructed of a plastic, metal, or a combination of materials, for containing all or some of the components discussed herein. The mobile device 800 may include a power source 822 coupled to the processor 801, such as a disposable or rechargeable battery. The rechargeable battery may also be coupled to the peripheral device connection port to receive a charging current from a source external to the mobile device 800.

The various processors described herein may be any programmable microprocessor, microcomputer or multiple processor chip or chips that may be configured by software instructions (routines) to perform a variety of functions, including the functions of the various embodiments described herein. In the various devices, multiple processors may be provided, such as one processor dedicated to wireless communication functions and one processor dedicated to running other applications. Typically, software applications may be stored in internal memory before they are accessed and loaded into the processors. The processors may include internal memory sufficient to store the application software instructions. In many devices the internal memory may be a volatile or nonvolatile memory, such as flash memory, or a mixture of both. For the purposes of this description, a general reference to memory refers to memory accessible by the processors including internal memory or removable memory plugged into the various devices and memory within the processors.

The foregoing method descriptions and the process flow diagrams are provided merely as illustrative examples and are not intended to require or imply that the operations of the various embodiments must be performed in the order presented. As will be appreciated by one of skill in the art the order of operations in the foregoing embodiments may be performed in any order. Words such as "thereafter," "then," "next," etc. are not intended to limit the order of the operations; these words are simply used to guide the reader through the description of the methods. Further, any reference to claim elements in the singular, for example, using the articles "a," "an" or "the" is not to be construed as limiting the element to the singular.

The various illustrative logical blocks, modules, circuits, and algorithm operations described in connection with the embodiments disclosed herein may be implemented as electronic hardware, computer software, or combinations of both. To clearly illustrate this interchangeability of hardware and software, various illustrative components, blocks, modules, circuits, and operations have been described above generally in terms of their functionality. Whether such functionality is implemented as hardware or software depends upon the particular application and design constraints imposed by the overall system. Skilled artisans may implement the described functionality in varying ways for each particular application, but such implementation decisions should not be interpreted as causing a departure from the scope of the present claims.

The hardware used to implement the various illustrative logics, logical blocks, modules, and circuits described in connection with the embodiments disclosed herein may be implemented or performed with a general purpose processor, a digital signal processor (DSP), an application specific integrated circuit (ASIC), a field programmable gate array (FPGA), or other programmable logic device, discrete gate or transistor logic, discrete hardware components, or any combination thereof designed to perform the functions described herein. A general-purpose processor may be a microprocess-

or, but, in the alternative, the processor may be any conventional processor, controller, microcontroller, or state machine. A processor may also be implemented as a combination of computing devices, e.g., a combination of a DSP and a microprocessor, a plurality of microprocessors, one or more microprocessors in conjunction with a DSP core, or any other such configuration. Alternatively, some operations or methods may be performed by circuitry that is specific to a given function.

In one or more exemplary embodiments, the functions described may be implemented in hardware, software, firmware, or any combination thereof. If implemented in software, the functions may be stored on or transmitted over as one or more instructions or code on a non-transitory processor-readable, computer-readable, or server-readable medium or non-transitory processor-executable storage medium. The operations of a method or algorithm disclosed herein may be embodied in a processor-executable software module or processor-executable software instructions which may reside on a non-transitory computer-readable storage medium, a non-transitory server-readable storage medium, and/or a non-transitory processor-readable storage medium. In various embodiments, such instructions may be stored processor-executable instructions or stored processor-executable software instructions tangible, non-transitory computer-readable storage media may be any available media that may be accessed by a computer. By way of example, and not limitation, such non-transitory computer-readable media may comprise RAM, ROM, EPROM, CD-ROM or other optical disk storage, magnetic disk storage or other magnetic data storage devices, or any other medium that may be used to store desired program code in the form of instructions or data structures and that may be accessed by a computer. Disk and disc, as used herein, includes compact disc (CD), laser disc, optical disc, digital versatile disc (DVD), floppy disk, and Blu-ray Disc, where disks usually reproduce data magnetically, while discs reproduce data optically with lasers. Combinations of the above should also be included within the scope of non-transitory computer-readable media. Additionally, the operations of a method or algorithm may reside as one or any combination of a set of codes and/or instructions on a tangible, non-transitory processor-readable storage medium and/or computer-readable medium, which may be incorporated into a computer program product.

The preceding description of the disclosed embodiments is provided to enable any person skilled in the art to make or use the embodiment techniques of the claims. Various modifications to these embodiments will be readily apparent to those skilled in the art, and the generic principles defined herein may be applied to other embodiments without departing from the spirit or scope of the claims. Thus, the present disclosure is not intended to be limited to the embodiments shown herein but is to be accorded the widest scope consistent with the following claims and the principles and novel features disclosed herein.

What is claimed is:

1. A method for data management in a computing device utilizing a plurality of processing units, comprising: generating a data transfer heuristic model based on measurements from a plurality of sample data transfers between a plurality of data storage units; calculating data transfer costs for each of a plurality of tasks using the generated data transfer heuristic model; configuring the plurality of tasks to be performed on the plurality of processing units based at least on the

calculated data transfer cost by prioritizing the plurality of tasks to obtain a scheduling priority order; executing a first task in the scheduling priority order on a first processing unit;

determining whether a first data block within a first data storage unit associated with the first processing unit is no longer needed during execution of the first task; and evicting the first data block from the first data storage unit in response to determining that the first data block is no longer needed during the execution of the first task.

2. The method of claim 1, wherein the measurements include one or more measurements, power consumption measurements, or time and power consumption measurements.

3. The method of claim 1, wherein generating the data transfer heuristic model based on the measurements from the plurality of sample data transfers between the plurality of data storage units comprises: adjusting a coefficient of a formula that calculates an estimated cost.

4. The method of claim 3, wherein the formula calculates one of an estimated time cost, an estimated power consumption cost, or both an estimated time cost and an estimated power consumption cost.

5. The method of claim 4, further comprising: executing each of the plurality of tasks as configured on one of the plurality of processing units; and updating the data transfer heuristic model based on measurements of data transfers occurring during execution of the plurality of tasks.

6. The method of claim 1, wherein generating the data transfer heuristic model occurs at a boot-up time for the computing device or periodically.

7. The method of claim 1, wherein generating the data transfer heuristic model includes using embedded power monitoring capabilities of the computing device.

8. The method of claim 1, wherein calculating the data transfer costs for each of the plurality of tasks using the generated data transfer heuristic model comprises querying the data transfer heuristic model using a data transfer size, a source identity, and a destination identity.

9. The method of claim 8, wherein querying the data transfer heuristic model is performed via an application programming interface (API) call.

10. The method of claim 1, wherein calculating the data transfer costs for each of the plurality of tasks using the generated data transfer heuristic model comprises identifying data dependencies between the plurality of tasks.

11. The method of claim 10, wherein the data dependencies are provided via a scheduler.

12. The method of claim 1, wherein determining whether the first data block within the first data storage unit associated with the first processing unit is no longer needed during execution of the first task is based on computer information, an application programming interface (API) call within code of the first task, or both.

13. The method of claim 1, wherein evicting the first data block from the first data storage unit in response to determining that the first data block is no longer needed during the execution of the first task comprises:

- setting the first data block as a next victim data block in a replacement chain of a cache replacement policy associated with the first processing unit;
- detecting a data block miss corresponding to a second data block; and
- evicting the first data block from the first data storage unit in response to detecting the data block miss and based on the cache replacement policy.

14. The method of claim 13, further comprising:
determining whether the first data block is needed for
performing a second task in the scheduling priority
order, wherein the second task is performed by a second
processing unit associated with a second data storage
unit; and
transferring the first data block to the second data storage
unit in response to determining that the first data block
is needed for performing the second task and upon
eviction from the first data storage unit.

15. The method of claim 1, wherein evicting the first data
block from the first data storage unit in response to
determining that the first data block is no longer needed during
the execution of the first task comprises:

setting the first data block as a next victim data block in
the first data storage unit;
identifying that new driver-managed data is needed without
the first data storage unit for use with the first task;
determining whether there is insufficient storage space in
the first data storage unit for the identified new driver-
managed data; and
evicting the first data block from the first data storage unit
in response to determining that there is insufficient
storage space for the new driver-managed data in the
first data storage unit.

16. The method of claim 15, where the first data storage
unit is a driver-managed data structure associated with the
first processing unit; a custom software-managed data store;
or a custom hardware-managed data store.

17. The method of claim 1, wherein configuring the
plurality of tasks to be performed on the plurality of pro-
cessing units based at least on the calculated data transfer
costs further comprises:

determining whether there is a better processing unit for
executing a second task based on the calculated data
transfer costs, wherein the second task is already con-
figured to execute on a first processing unit; and
re-assigning the second task to be executed by a second
processing unit in response to determining that there is
a better processing unit.

18. The method of claim 1, wherein configuring the
plurality of tasks to be performed on the plurality of pro-
cessing units further comprises configuring dynamic voltage
and frequency scaling (DVFS) settings for processing units
executing each of the plurality of tasks based on the cal-
culated data transfer costs.

19. The method of claim 18, wherein the DVFS settings
control a use of a processing unit, a use of a bus, or a
combination of both.

20. A computing device, comprising:
a memory; and
a processor of a plurality of processing units, wherein the
processor is coupled to the memory and is configured
with processor-executable instructions to perform
operations comprising:

generating a data transfer heuristic model based on
measurements from a plurality of sample data trans-
fers between a plurality of data storage units;
calculating data transfer costs for each of a plurality of
tasks using the generated data transfer heuristic
model;
configuring the plurality of tasks to be performed on the
plurality of processing units based at least on the
calculated data transfer costs, wherein configures
the plurality of tasks comprises prioritizing the plu-
rality of tasks to obtain a scheduling priority order;

executing a first task in the scheduling priority order on
a first processing unit; determining whether a first
data block within a first data storage unit associated
with the first processing unit is no longer needed
during execution of the first task; and
evicting the first data block from the first data storage
unit in response to
determining that the first data block is no longer needed
during the execution of the first task.

21. The computing device of claim 20, wherein the
measurements include one of time measurements, power
consumption measurements, or time and power consump-
tion measurements.

22. The computing device of claim 20, wherein the
processor is configured with processor-executable instruc-
tions to perform operations such that generating the data
transfer heuristic model based on the measurements from the
plurality of sample data transfers between the plurality of
data storage units comprises adjusting a coefficient of a
formula that calculates an estimated cost.

23. The computing device of claim 20, wherein the
processor is configured with processor-executable instruc-
tions to perform operations further comprising:
executing each of the plurality of tasks as configured on
one of the plurality of processing units; and
updating the data transfer heuristic model based on mea-
surements of data transfers occurring during execu-
tions of the plurality of tasks.

24. The computing device of claim 20, wherein the
processor is configured with processor-executable instruc-
tions to perform operations such that calculating the data
transfer costs for each of the plurality of tasks using the
generated data transfer heuristic model comprises querying
the data transfer heuristic model using a data transfer size,
a source identity, and a destination identity.

25. A computing device, comprising:
means for generating a data transfer heuristic model based
on measurements from a plurality of sample data
transfers between a plurality of data storage units;
means for calculating data transfer costs for each of a
plurality of tasks using the generated data transfer
heuristic model;

means for configuring the plurality of tasks to be per-
formed on a plurality of processing units based at least
on the calculated data transfer costs, wherein the means
for configuring the plurality of tasks comprises means
for prioritizing the plurality of tasks to obtain a sched-
uling priority order;

means for executing a first task in the scheduling priority
order on a first processing unit;

means for determining whether a first data block within a
first data storage unit associated with the first process-
ing unit is no longer needed during execution of the first
task; and

means for evicting the first data block from the first data
storage unit in response to determining that the first
data block is no longer needed during the execution of
the first task.

26. A non-transitory processor-readable storage medium
having stored thereon processor-executable instructions
configured to cause a processor of a computing device to
perform operations comprising:
generating a data transfer heuristic model based on mea-
surements from a plurality of sample data transfers
between a plurality of data storage units;
calculating data transfer costs for each of a plurality of
tasks using the generated data transfer heuristic model;

US 9,733,978 B2

29

configuring the plurality of tasks to be performed on a plurality of processing units based at least on the calculated data transfer costs, wherein configuring the plurality of tasks comprises prioritizing the plurality of tasks to obtain a scheduling priority order;
executing a first task in the scheduling priority order on a first processing unit,
determining whether a first data block within a first data storage unit associated with the first processing unit is no longer needed during execution of the first task, and
evicting the first data block from the first data storage unit in response to determining that the first data block is no longer needed during the execution of the first task.

30



US 20170206035A1

(19) United States

(21) Patent Application Publication (10) Pub. No.: US 2017/0206035 A1
(22) Kumar et al. (143) Pub. Date: Jul. 20, 2017

(54) RANDOM-ACCESS DISJOINT CONCURRENT SPARSE WRITES TO HETEROGENEOUS BUFFERS

(71) Applicant: QUALCOMM Incorporated, San Diego, CA

(72) Inventors: Tushar Kumar, San Francisco, CA (US); Aravind Narasajan, Sunnyvale, CA (US); Darío Suárez Gracia, Madrid (ES)

(21) Appl. No.: 15/600,667

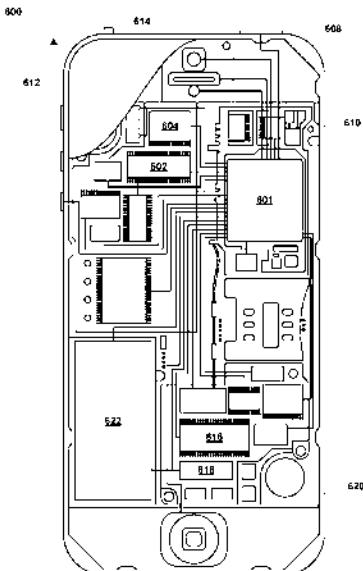
(22) Filed: Jun. 19, 2016

Publication Classification

(51) Int. Cl.
G06F 3/46 (2006.01)
G06F 12/10 (2006.01)(52) U.S. Cl.
C10C G06F 3/4656 (2013.01); G06F 3/467 (2013.01); G06F 3/4673 (2013.01); G06F 12/102 (2013.01); G06F 2212.65 (2013.01)

(57) ABSTRACT

Methods, devices, and non-transitory processor-readable storage media for a computing device to merge concurrent writes from a plurality of processing units to a buffer associated with an application. An embodiment method executed by a processor may include identifying a plurality of concurrent requests to access the buffer that are sparse, disjoint, and write-only; configuring a write-set for each of the plurality of processing units; executing the plurality of concurrent requests to access the buffer using the write-sets; determining whether each of the plurality of concurrent requests to access the buffer is complete; obtaining a buffer index and data via the write-set of each of the plurality of processing units; and writing to the buffer using the received buffer index and data via the write-set of each of the plurality of processing units in response to determining that each of the plurality of concurrent requests to access the buffer is complete.



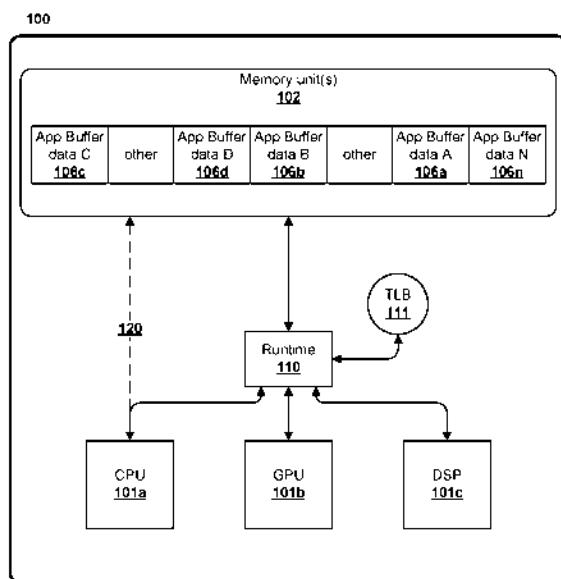


FIG. 1

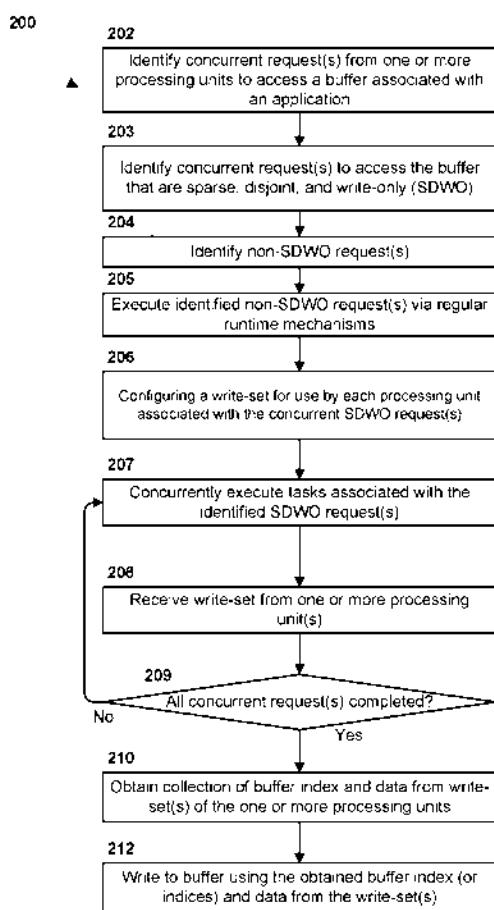


FIG. 2

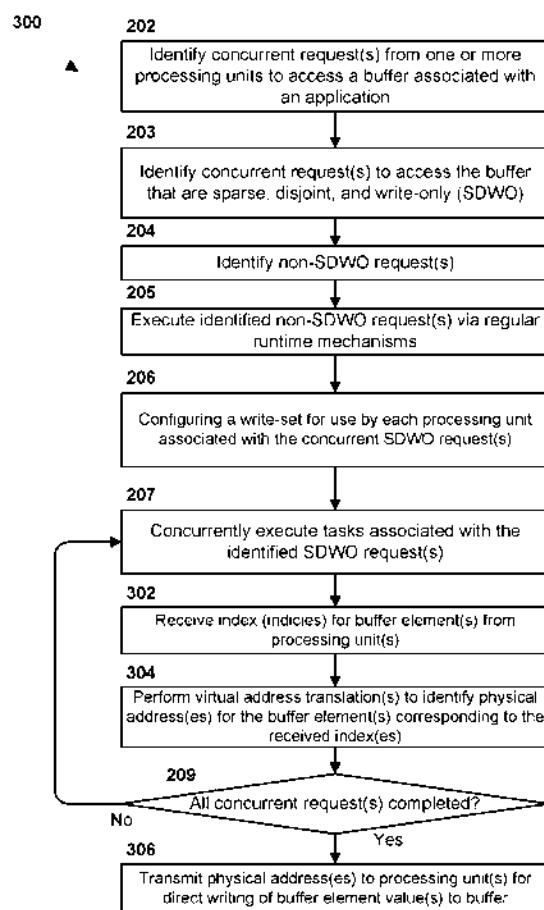


FIG. 3A

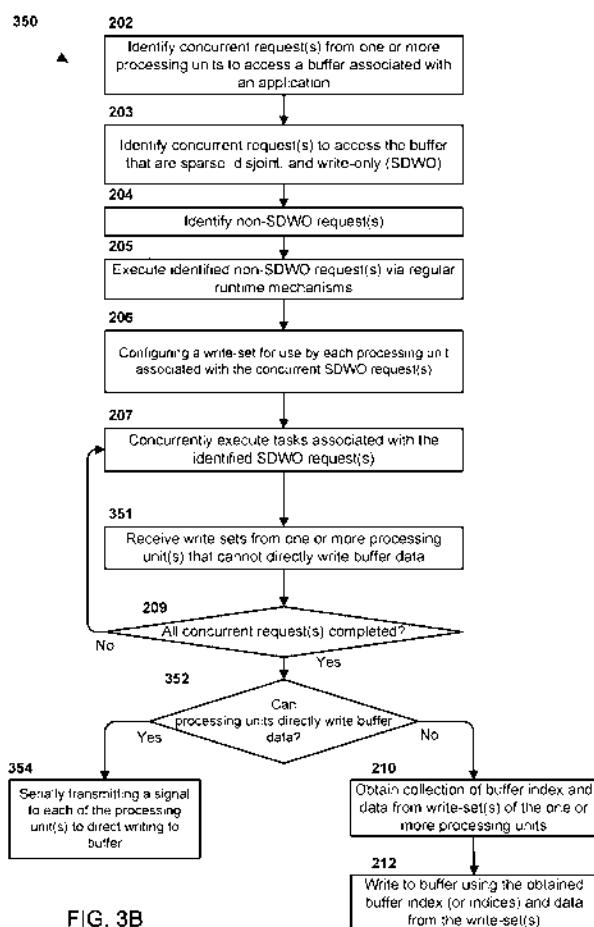


FIG. 3B

400

4

Pseudocode

//create a buffer with 100000 entries buffer<float> b(100000);	402
//launch tasks on different processing units //buffers a,c,d,e get transferred to processing units at launch //buffer b does not get transferred as it is declared SDWO t1 = launch_task(cpu_function, a, sdwo(b), ..); t2 = launch_task(gpu_kernel, c, sdwo(b), ..); t3 = launch_task(dsp_kernel, d, sdwo(b), ..); t4 = launch_task(custom_hw_acc1, e, sdwo(b), ..);	404
//wait for all concurrent tasks to complete SDWO access to buffer b wait_for(t1); wait_for(t2); wait_for(t3); wait_for(t4);	406

FIG. 4

500

Kernel Pseudocode for SDWO

```
void f(global float* c, global float* b_sdwo)
{
    ...
    sdwo_write(b_sdwo, i, value);
}
```

FIG. 5A

550

Kernel Pseudocode with SDWO specialized HW

```
void f(global float* c, global float* b)
{
    ...
    b[i] = value;
}
```

FIG. 5B

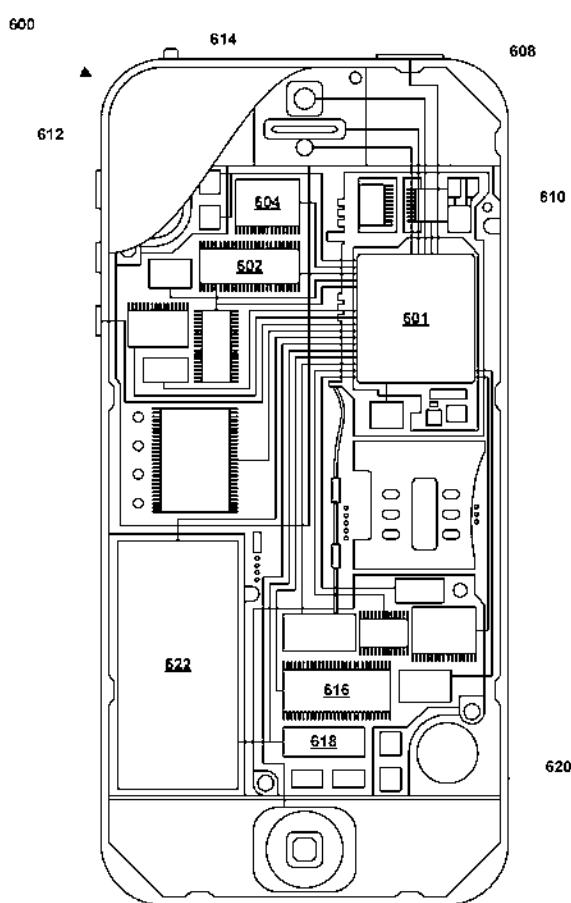


FIG. 6

RANDOM-ACCESS DISJOINT CONCURRENT SPARSE WRITES TO HETEROGENEOUS BUFFERS

BACKGROUND

[0001] Computing devices often utilize runtime systems to create abstractions of data that are accessible by various processing units or functionalities, such as central processing units (CPUs), digital signal processors (DSPs), graphical processing units (GPUs), camera pipelines, etc. Some runtime systems can identify and manage access to data for an application that is stored in various memory units of a computing device. In particular, some runtime systems may manage the application data and provide an abstract representation, such as a "buffer" to abstract over array data. Buffer data may reside on various types of memory units accessible by various processing units of a computing device. Buffer data may be accessed via a programmatic buffer handle that may be used by application code executing on the various processing units. In response to buffer access requests (e.g., using programmatic buffer handles), runtime systems can move buffer data to memory units associated with requesting processing units.

[0002] Due to the different capabilities and structures of the various processing units of a computing device, such runtime systems often make the data of buffers available via different manners for different processing units. In particular, copies of buffer data may be individually stored and maintained as "backing stores" on suitable memory units accessible by particular processing units. For example, a backing store may be an allocated block of main memory for a CPU, a specialized driver-managed data store generated using OpenGL drivers for a GPU, an allocated block in a specialized region of main memory (such as RDN memory on Advanced RISC Machines (ARM) architectures) for a DSP or a camera pipeline, and/or internal registers and local memory banks for a custom accelerator.

[0003] Some runtime systems internally allocate and synchronize various backing stores across processing units. Additionally, for some processing units, such runtime systems may provide abstract programming mechanisms by which application code executing on a processing unit may be able to read and write the buffer data. In such cases, the application code may access the buffer data contents (e.g., via a runtime system) without knowledge of which backing stores currently hold the buffer data. Some runtime systems may hide the layout of buffer data within a backing store from application code. Such abstract interfaces for data access typically allow runtime systems to perform various data movement, data storage, and/or synchronization optimizations.

SUMMARY

[0004] Various embodiments provide methods, devices, and non-transitory process-readable storage media for a multi-processor computing device to merge concurrent writes from a plurality of processing units to a buffer associated with an application. An embodiment method performed by the multi-processor computing device may include identifying a plurality of concurrent requests to access the buffer that are sparse, disjoint, and write-only (SDWO), configuring a write-set for each of a plurality of processing units, executing the plurality of concurrent

requests to access the buffer using the write-sets, determining whether each of the plurality of concurrent requests to access the buffer is complete, obtaining a buffer index and data via the write-set of each of the plurality of processing units, and writing to the buffer using the obtained buffer index and data via the write-set of each of the plurality of processing units in response to determining that each of the plurality of concurrent requests to access the buffer is complete. In some embodiments, identifying the plurality of concurrent requests to access the buffer that are sparse, disjoint, and write-only (SDWO) may include identifying an SDWO application programming interface (API) call of a task executing on each of the plurality of processing units.

[0005] Some embodiments may further include receiving an index for a buffer element from one of the plurality of processing units, and performing a virtual address translation to identify a physical address for the buffer element corresponding to the received index. Such embodiments may further include transmitting the physical address to one of the plurality of processing units for direct writing of a buffer element value to the buffer in response to performing the virtual address translation.

[0006] In some embodiments, determining whether each of the plurality of concurrent requests to access the buffer is complete may include waiting until all of the plurality of processing units have completed performing respective tasks for writing to associated write-sets. In some embodiments, determining whether each of the plurality of concurrent requests to access the buffer is complete may include identifying a marker within code of the application indicating completion of requested buffer accesses by the plurality of processing units. Some embodiments may further include determining whether the plurality of processing units are configured to directly write to the buffer and serially transmitting a signal to each of the plurality of processing units in response to determining that the plurality of processing units are configured to directly write to the buffer, in which the signal indicates when each of the plurality of processing units can directly write to the buffer.

[0007] Some embodiments may further include identifying another plurality of concurrent requests to access the buffer that are not sparse, disjoint, and write-only, and serially transferring the buffer to each processing unit associated with the another plurality of concurrent requests such that the another plurality of concurrent requests are executed serially. Some embodiments may further include identifying a coordinator processor as having a backing store with valid data for the buffer, and transmitting data of the write-sets to the coordinator processor for writing to the backing store in response to determining that each of the plurality of concurrent requests to access the buffer is complete. In some embodiments, writing to the buffer using the obtained buffer index and data via the write-set of each of the plurality of processing units in response to determining that each of the plurality of concurrent requests to access the buffer is complete may include writing to the backing store associated with the coordinator processor.

[0008] In some embodiments, each of the plurality of concurrent requests to access the buffer may correspond to an access of a separate data structure that uses the buffer as a memory pool, in which the data structure may be one of a linked list or a tree. In some embodiments, the application may be a step simulation program. In some embodiments, the processor may be executing a runtime system. In some

embodiments, identifying the plurality of concurrent requests to access the buffer that are sparse, disjoint, and write-only (SDWO) may include determining that the plurality of concurrent requests are requests to write the same value to the same buffer index.

[0019] Further embodiments include a computing device configured with processor-executable instructions for performing operations of the methods described above. Further embodiments include a computing device having means for performing functions of the methods described above. Further embodiments include a non-transitory processor-readable medium on which is stored processor-executable instructions configured to cause a computing device to perform operations of the methods described above.

III. DESCRIPTION OF THE DRAWINGS

[0020] The accompanying drawings, which are incorporated herein and constitute part of this specification, illustrate various embodiments, and together with the general description given above and the detailed description given below, serve to explain the features of the claims.

[0021] FIG. 1 is a component block diagram of a multi-processor computing device configured to support sparse, disjoint, write-only operations according to various embodiments.

[0022] FIG. 2 is a process flow diagram illustrating a method for a multi-processor computing device to synchronize concurrent buffer accesses that are sparse, disjoint, and write-only according to some embodiments.

[0023] FIGS. 3A-B are process flow diagrams illustrating methods for a multi-processor computing device to synchronize concurrent buffer accesses that are sparse, disjoint, write-only according to some embodiments.

[0024] FIG. 4 is a listing of pseudocode for a runtime functionality that enables concurrent buffer accesses that are sparse, disjoint, and write-only according to some embodiments.

[0025] FIGS. 5A-5B are listings of pseudocode that may be performed by processing units of a multi-processor computing device according to some embodiments.

[0026] FIG. 6 is a component block diagram of a computing device suitable for use in some embodiments.

DETAILED DESCRIPTION

[0027] The various embodiments will be described in detail with reference to the accompanying drawings. Wherever possible, the same reference numerals will be used throughout the drawings to refer to the same or like parts. References made to particular examples and implementations are for illustrative purposes, and are not intended to limit the scope of the embodiments or the claims.

[0028] This disclosure includes illustrative descriptions of various implementations or embodiments. However, any implementation described herein should not necessarily be construed as preferred or advantageous over other implementations.

[0029] The term "computing device" is used herein to refer to an electronic device equipped with at least a processor. Examples of computing devices may include mobile devices (e.g., cellular telephones, wearable devices, smartphones, web-pads, tablet computers, Internet enabled cellular telephones, Wi-Fi® enabled electronic devices, personal

data assistants (PDAs), laptop computers, etc.), personal computers, and server computing devices.

[0030] The term "multi-processor computing device" is used herein to refer to a computing device configured to execute various tasks (e.g., routines, instruction sets, etc.) with two or more processing units. For example, a multi-processor computing device may be a heterogeneous computing device (e.g., a system-on-chip (SoC)) with different processing units, each configured to perform specialized and/or general-purpose workloads. Such processing units may include various processor devices, a core, a plurality of cores, etc. For example, processing units of a multi-processor computing device may include an application processor (AP) (e.g., a central processing unit (CPU)) and one or more specialized processing devices, such as a graphics processing unit (GPU), a digital signal processor (DSP), and a modem processor.

[0031] In general, processing units of a multi-processor computing device may execute various software routines, kernels, threads, instructions, and/or code associated with an application (referred to herein simply as "tasks"). Some tasks executing on various processing units of a multi-processor computing device may be configured to access a common buffer. Conventional techniques for enabling and managing buffer access by various tasks often result in sub-optimal performance. For example, in some cases, tasks may be configured to manage coherent access to the buffer data, such as via the use of critical sections, atomic variables, and other application-managed access techniques. However, such application-managed access techniques can fail when processing units that rely on different backing stores of buffer data execute concurrent buffer accesses. For example, a CPU task can read and write to a buffer by accessing a backing store in main memory, while a concurrently-executing GPU task can access an OpenCL-managed backing store, often in GPU local memory. The concurrent CPU and GPU tasks may not be aware of updates to the buffer performed other tasks, causing the buffer data to become inconsistent.

[0032] In many cases, runtime systems managing buffers on behalf of applications may be forced to serialize the execution of a plurality of processing units that would concurrently write to a buffer, thereby slowing application execution. In addition to the performance penalty incurred when runtime systems serialize execution of tasks writing buffer data, there may be additional costs incurred due to necessary synchronization of data between the backing stores in between the execution of the serialized tasks. Such synchronization may be the transfer of data from a first backing store (accessible by some of the processing units) to a second backing store (accessible by some other processing units) required so that updates to the buffer by the earlier executing tasks become visible to the later executing tasks.

[0033] In some cases, a first backing store and a second backing store may be tied together such that a runtime system need only change some memory configuration and perform driver flushes to ensure the second backing store has up-to-date data (i.e., a low cost synchronization). However, in many common situations, the entire contents of the buffer data must be copied from the first backing store to the second backing store (referred to as a "full-buffer copy"), even if only a small portion of the buffer data has been updated in the first backing store compared to the data

already present in the second backing store. This type of synchronization can impact device performance by requiring significant processing time and consuming battery power.

[0024] Not all backing stores of a computing device may be configured for coherency, and thus flushing may be required. For example, some processing units may access a backing store in shared virtual memory (SVM) allowing concurrent writes in a consistent manner. Other processing units may only support partial SVM in which accesses to a backing store originating from different processing units are inconsistent due to a lack of cache-coherence mechanisms. In such cases, explicit driver-managed flushes may be required to synchronize with the backing store.

[0025] In other cases, multiple tasks executing on various processing units may concurrently write to the same buffer without creating inconsistency, such as writes to distinct elements of the buffer. However, the involvement of multiple backing stores and/or the lack of coherency mechanisms on backing stores may result in performance penalties (e.g., task serialization, full-buffer copies, etc.). Even if concurrent tasks accessed an SVM backing store, there may be the possibility of "false sharing" in which different tasks write to different buffer elements co-located on the same cache line, causing cache thrashing and performance degradation.

[0026] Techniques are needed that enable concurrent execution of tasks despite multiple backing stores and avoid full-buffer copies between backing stores that would otherwise need full-buffer copy synchronization.

[0027] The various embodiments include techniques for improving operations in a multi-processor computing device during concurrent buffer accesses that are "sparse, disjoint, and write-only" (SDWO). As referred to herein, SDWO buffer accesses may correspond to special-case accesses by one or more processing units (or tasks executing on the same or more processing units) of a multi-processor computing device to a buffer assigned to an executing application (e.g., a program, etc.). Concurrent SDWO buffer accesses may be random (e.g., indices accessed are determined only at runtime by each processing unit), sparse (e.g., only a small fraction of the buffer is likely to be accessed), disjoint (e.g., various processing units access distinct buffer indices), and write-only (e.g., none of the processing units participating in concurrent SDWO buffer accesses may read any of the buffer data).

[0028] Various embodiments include methods, as well as multi-processor computing devices executing such methods, and non-transitory processor-readable storage media storing instructions of such methods, to merge concurrent SDWO writes to an application's buffer. In general, a multi-processor computing device may be configured to identify concurrent requests by various processing units to perform SDWO accesses of a buffer for an application (e.g., a program, etc.). For example, the multi-processor computing device, via a runtime functionality, may identify the invocation in task code of special application programming interface (API) calls associated with SDWO buffer accesses (or access requests). When concurrent SDWO buffer access requests occur, the multi-processor computing device may set up each processing unit with a per-device write-set instead of copying the buffer data across backing stores used by each of the processing units. For example, the multi-processor computing device may configure a write-set com-

prised of a set of buffer index, value pairs for each processing unit. The individual processing units may use the respective write-sets to indicate buffer locations (e.g., index test corresponding to data to be written to the buffer). When the concurrent SDWO buffer accesses are complete and all write-set information is received from the processing units, the multi-processor computing device may perform writes to the buffer as defined by the various per-device write-sets. For example, the write-sets may be used to update a current backing store maintained by an individual processing unit (e.g., CPU) that the multi-processor computing device uses to update the buffer. The various embodiments enable the multi-processor computing device to execute power-efficient and time-efficient writes to a buffer that do not require processing units to have the same coherency abilities. This capability avoids the serialization and/or costly buffer copying allocations.

[0029] In various embodiments, SDWO buffer accesses may be identified by application programmers based on the semantics of applications. In particular, an application programmer may explicitly declare a buffer access as an SDWO access within code of a task. For example, a CPU task and a GPU task may be structured (or coded) to update distinct elements of a buffer, not read any part of the buffer, and may likely write only to a small fraction of the elements of the buffer. Based on these semantics or characteristics of the tasks of the application, an application programmer may be able to determine that the CPU and GPU tasks may execute SDWO buffer accesses. The application programmer may designate accesses by the CPU task and GPU task as SDWO for that buffer, such as by programming the respective tasks to utilize a suitable SDWO API call. When the tasks are executed and invoke the SDWO API call, the multi-processor computing device (e.g., via a runtime functionality) may perform embodiment SDWO optimizations, allowing the tasks for the CPU and the GPU to execute concurrently while avoiding full-buffer copies.

[0030] As another example, separate buffer entries may act as "data payloads" for nodes of linked-list data-structures used by an application. A first and a second linked-list may have associated buffer entries interspersed over the buffer for the application. However, accessing the nodes of the first linked list would access buffer entries distinct from the buffer entries associated with nodes of the second linked list. A kernel configured to run on a DSP may "walk" (or traverse) the first linked list and write the corresponding buffer entries for the corresponding node data, whereas a kernel configured to run on a GPU may walk the second linked list and write the corresponding buffer entries. As each kernel addresses a different data element/data object of the buffer, the corresponding write requests would only address distinct, though possibly interspersed elements of the buffer. As there are no producer-consumer relationships on buffer entries across processing units, disjointness of concurrent write requests is guaranteed, and therefore the application programmer may declare such write requests SDWO within the kernels' code. As another example, a DSP kernel and a GPU kernel working on two sibling subtrees of a tree data-structure may be declared SDWO over the writes to a buffer whose entries act as the data payload for the tree nodes.

[0031] In various embodiments, the multi-processor computing device may be configured to identify tasks or processing units that will make concurrent buffer accesses that

are SDWO. For example, the multi-processor computing device may evaluate application code or instruction sets to detect when kernels for two processing units (e.g., CPU, GPU, etc.) are configured to submit requests to write data to distinct buffer indexes. In various embodiments, compiler analysis of kernel code may indicate that buffer indices written to by a first kernel are distinct from the buffer indices written to by a second kernel. In some embodiments, the multi-processor computing device, via a runtime functionality, may identify application programming interface (API) calls within code of tasks to be executed via one or more processing units (e.g., tasks for a GPU, CPU, and DSP that correspond to a particular physics engine program). The runtime functionality may be informed that the tasks make SDWO buffer access requests before the corresponding processing units (e.g., GPU, CPU and DSP) launch and execute the tasks. For example, SDWO buffer accesses may be declared within code beforehand. An example of such an API that may be called is an “sdwoInit” command, wherein ‘h’ is a buffer handle passed as an argument to a task invoking the API command.

[0032] In various embodiments, in response to detecting concurrent requests by a plurality of processing units (and respective tasks) to perform SDWO buffer accesses of the buffer, the multi-processor computing device may set-up or configure each of the processing units with per-device write-sets. For example, instead of buffering tracking stores and transferring data to the various processing units, the multi-processor computing device may simply instruct the processing units to set up write-sets for a given buffer identified as having SDWO access. With the write-sets, the code executing on the individual processing units (e.g., kernel code) may provide indices of the buffer that should be updated with new data, such as via an API call. For example, a kernel executing on a DSP may execute an API call “sdwo_write(...);” that includes the index of the buffer and a data value of the type stored by the buffer (e.g., with a “float” buffer, the processing unit may invoke “sdwo_write(index, 3.0f” to write the floating-point value “3.0” at the specified buffer index, etc.).

[0033] In general, there may be various device-specific mechanisms for creating write sets within the local memory of each processing unit (or processing device) of the multi-processor computing device. In some embodiments, a write-set of a processing unit may be implemented as an array of index-data pairs with a new index-data pair appended by the processing unit in response to each sdwo_write call on that processing unit. In some embodiments, hash-maps may be used to record the index-data pairs of write-sets. Use of the write-sets enables new buffer data to be communicated between processing units without time and power-consuming data transfers of the entire contents of the buffer back and forth between storage units (e.g., device caches, etc.). In some embodiments, when all the various processing units of the multi-processor computing device utilize full or partial SVM support, write-sets may be enabled via pointers provided to the processing units.

[0034] In some embodiments, the multi-processor computing device may transfer write-sets from various processing units to a coordinator processor. Such a coordinator processor may be identified as designated based on the processor being associated with a valid backing store of the buffer. For example, a CPU having a memory backing store with the latest data of the buffer may be identified as the

coordinator processor and thus may receive the write-sets for inserting data from the processing units into the backing store. In some embodiments, the coordinator processor may be any processing unit within the multi-processor computing device that has a valid backing store of the buffer. In some embodiments, the multi-processor computing device may perform operations to rank in alternative sort in order of efficiency each processing unit that may function as a coordinator processor in order to use the fastest and/or most efficient processing unit and backing store.

[0035] The following is an illustration of an implementation of SDWO buffer accesses and a coordinator processor. After concurrently executing processing units have updated respective write-sets for SDWO buffer accesses, the various write-sets may be sent to a designated coordinator processor (e.g., at P1) that has access to a backing store for the buffer. Such a coordinator processor may update the backing store using the received write-sets. Since the writes by the coordinated processor are sparse, the total size of all write-sets may be small compared to the size of the buffer. Therefore, the backing store may be updated with the writes in a duration of time much shorter than the time required to transfer the entire contents of the buffer, as would have been necessary without SDWO.

[0036] In various embodiments, the processing units may transfer write-sets (e.g., to the coordinator processor) either when a processing unit has completed (e.g., completed a task), or intermittently during the execution of a task on the processing unit. In some embodiments, the coordinator processor may determine when to update a backing store for the buffer based on determining that all processing units associated with concurrent SDWO buffer accesses have completed operations and, thus, have transferred write-set data. In some embodiments, a programmer may simply indicate a completion point after which any updates of kernels processing units should occur.

[0037] When concurrent SDWO buffer accesses are complete (e.g., all write-sets have been received from processing units), the multi-processor computing device may update a backing store that includes valid buffer data. For example, the multi-processor computing device may overwrite data in an up-to-date copy of the buffer backing store accessible by a CPU.

[0038] Even using write-sets can be costly when validated entries in a buffer are very large. For example, transfer of a large data value in a coordinator processor via a write-set may then the writing of the data value to a backing store by the coordinator processor may entail two transfers of the large data value over the memory system. To address this, in some embodiments, the multi-processor computing device may be configured to identify processing units that can perform write-backs directly to memory using buffer indices. In other words, some processing units may be configured to directly access a backing store being updated by the coordinator processor in order to avoid an extra data value transfer. Such a technique may not require coherent access to the backing store. For example, the backing store of a coordinator processor may be in GPU local memory, allowing a write-set created by a GPU kernel to be directly updated to the backing store under the supervision of the coordinator processor.

[0039] In some cases, processing units that have direct coherent access to a backing store, such as via SVM, may not need to use write-sets, but other units without coherent

access capabilities may use write-sets. In this way, embodiment techniques enable the mixing of the two types of processing units and may reduce the number of write-sets involved. For example, after ensuring the completion of concurrently executing processing units that make SDWO accesses to a buffer, the multi-processor computing device may signal various processing units with SVM timetables to serialize buffer write-backs, punctuated by appropriate driver flushes. Such embodiments may enable the multi-processor computing device to minimize write-back of the write-sets by some processing units (e.g., CPU) with write-set updates performed by the coordinator processor for other processing units devices (e.g., DSP, GPU, accelerators).

[0040] For example, the multi-processor computing device may identify that some processing units may be capable of directly accessing memory such that backing stores with the latest buffer data may be directly accessible by these processing units, though perhaps not in a coherent manner. Thus, associated backing stores may not be required to be allocated or synchronized to hold the latest buffer data before these processing units perform SDWO operations. Such processing units may not need to send write-sets to a coordinator processor, but instead may write data associated with SDWO buffer accesses directly after execution of the concurrent SDWO buffer accesses. The coordinator processor may signal these processing units to indicate when each unit safely applies write-set data directly to the backing store. The coordinator processor may ensure correctness by serializing the write-set writes by each processing unit and/or executing memory barrier instruction to ensure coherence before signaling the next processing unit.

[0041] In some embodiments, when processing units can handle physically addressed memory but have no virtual address capabilities, the processing units may send write-set indices to the multi-processor computing device (e.g., the coordinator processor) for virtual address translations. For example, when a DSP is not configured to utilize a translation lookaside buffer (TLB) for virtual address translations, the multi-processor computing device, via a runtime functionality, may perform conversion operations to translate buffer indices from the DSP into physical addresses. Based on such translations, the individual processing units may use the physical address to directly write in memory.

[0042] In some embodiments, the multi-processor computing device may be configured with dedicated hardware that enables SDWO buffer accesses without special API calls. For example, hardware may be configured to enable routine recognition of the address ranges of regular accesses to a buffer in kernel code and translate those automatically into a write-set for that buffer. Such embodiment multi-processor computing devices may enable seamless write-set use that is invisible to programmers. With such hardware implementations, individual kernels executing on processing units may simply use regular API calls and/or buffer accesses. Such implementations allow the hardware improvements to identify SDWO buffer accesses and act without additional programmer effort.

[0043] In various embodiments, the multi-processor computing device may be configured to support reads of a buffer while maintaining coherency between various processing units. For example, the multi-processor computing device may provide heterogeneous processing units with a previous version of a buffer (e.g., a time 't' version of a buffer used in a physics simulation) for reading purposes and configure

write-sets for writing to the buffer (e.g., to capture simulation-state updates for time 't+1' by concurrently executing processing units). Subsequent SDWO write calls by the processing units may generate the write-sets (e.g., data to be written), which may be applied to a backing store holding the time 't' version of the buffer data to generate the updated time 't+1' version of the buffer backing store.

[0044] In some embodiments, the multi-processor computing device may be configured to enable buffer accesses that are not necessarily disjoint. In other words, the multi-processor computing device may implement "approximation-tolerant" concurrent SDWO buffer accesses. For example, when concurrent updates to a certain buffer index (e.g., b[1]) are deemed valid (e.g., the concurrently written values are all considered "approximately correct"), or there is an expectation that each processing unit will attempt to write the same value, etc., the multi-processor computing device may apply both write-sets to the buffer. In other words, when identifying that a plurality of concurrent requests to access the buffer are SDWO, the multi-processor computing device may determine that the plurality of concurrent requests are requests to write a same value to a same buffer index. In some embodiments, such relaxed circumstances for disjointness may require new API calls, such as "sparse, approximate, write-only," (e.g., *sawn* writes).

[0045] Methods according to the various embodiments may be performed by a routine functionality supported by the multi-processor computing device. For example, a processing unit (e.g., CPU) may be configured to execute or otherwise utilize a service, software, instructions, circuitry, or any combination thereof that identifies concurrent SDWO buffer accesses (e.g., identified SDWO API calls in task code, etc.) and provides write-sets to various processing units to facilitate concurrent SDWO requests.

[0046] Various embodiments provide methods for processing units (e.g., GPU, CPU, DSP, etc.) of a multi-processor computing device to concurrently perform sparse, disjoint and write-only operations corresponding to an application's buffer. Specifically addressing SDWO buffer accesses, the methods of various embodiments may be used to configure a multi-processor computing device to synchronize a plurality of sparse and disjoint writes to a buffer, only writing to a single buffer representation (e.g., a backing store memory, etc.) when all concurrent writing units have provided respective data. This enables operations that typically would be serialized (e.g., executed in sequence) to be performed concurrently. Such embodiment methods may reduce the amount of redundant buffer data that is required to be copied and/or transferred in order for multiple devices to perform such writes. For example, full-buffer-copies may be eliminated as tasks on different processing units may use the write-sets to update buffers. As another example, a processing units can be enabled to directly write to a buffer in embodiments that utilize signaling and/or virtual address translations from a coordinator processor; double-copying of updated buffer elements may also be eliminated, such as when a buffer element is transferred from the processing unit to the coordinator processor to a backing store. These improvements can be especially beneficial when buffers or buffer elements are large.

[0047] The various embodiments may also improve power usage of multi-processor computing devices. For example, power may be saved by deactivating cache-coherence protocols (e.g., full S3 MI) and/or by reducing the amount of bus

activity (e.g., minimizing data transfer overheads by avoiding buffer copying, etc.). Various embodiments may further improve the operations of a multi-processor computing device by decreasing the potential thrashing that could otherwise occur due to cache accesses of a plurality of devices.

[0048] Applications or programs that utilize linked lists may benefit from the various embodiments. In some cases, multiple linked lists may be allocated from a single pool. If the pool is a routine-managed buffer, the linked list nodes from the multiple linked lists may be dispersed throughout the buffer. If a plurality of tasks concurrently executing via a plurality of processing units process distinct linked lists allocated from the same buffer pool, the accesses from the concurrent tasks will be disjoint. Instances in which the linked lists' nodes are allocated outside the buffer but the buffer holds the "payload" for the nodes, buffer accesses to the payloads may be write-only because reading to traverse the linked list (e.g., reading next pointers of nodes) may be done with the data outside of the buffer. In other cases, if the linked list nodes are allocated from within the buffer, the processing units may be provided (e.g., via a runtime functionality) with a read-only copy of the buffer data, while any writes to the contents of a linked list node may be recorded in write-sets. After the concurrently executing processing units complete, the write-sets may be applied to a single backing store, which will represent the updated buffer data. While the cost of using read-only copies of the backing stores available to the processing units may not be avoided, the costs of serialization and full buffer copies of the updated buffer contents are avoided.

[0049] Applications that utilize tree data-structures may also benefit from the various embodiments. In some cases, tree data-structures may be efficiently stored as elements of an array, or alternatively tree node "payloads" may be pool-allocated within a buffer. For example, a programmer may recognize code configurations of an application that include tasks operating on sibling sub-trees under a root. In such a case, the task accesses may be identified as SDWO, in a manner similar to linked-list embodiments.

[0050] Time-step simulations for applications may benefit from embodiment techniques. For example, hacking stores of processing units may read such data for time-step 't', possibly requiring one buffer copy in the backing store for each processing unit. However, write operations for the various tasks may only be for time-step 't+1'. Write-sets for the various processing units may be updated at the end of the concurrent tasks to one chosen backing store, avoiding a second buffer copy from each updated backing store. Hence, concurrent task execution may be supported using the SDWO technology despite the presence of certain patterns of buffer reads often seen in time-step simulations, and thus a computing device implementing various embodiments may avoid copying the updated backing stores again and again after each serialized task completes updates.

[0051] As described, embodiment methods may be particularly beneficial with applications that require multiple, potentially diverse processing units to perform writes to a large buffer, such as step simulations or other programs that support large systems (e.g., video game physics engines, linked lists, computer vision applications, etc.). For

example, when executing a game physics engine, the multi-processor computing device may employ SDWO buffer access as described herein to implement an efficient collision detection algorithm that concurrently updates small portions of large world-state data per compute step, requiring little copying or buffer transfers.

[0052] Unlike other typical systems, the embodiment methods may be used to improve concurrent writing to an application buffer used by processing units that may or may not utilize cache coherence technologies. For example, various embodiments may be used to conduct concurrent writes from processing units that may or may not be configured to perform virtual address translations. Typical cache coherence methods do not provide optimizations for the special use case of concurrent sparse, disjoint, write-only accesses to a single application buffer. For example, typical methods may not decrease system traffic when a processing unit is required to write multiple times to the same buffer index; instead may require multiple cache line reads in order to conduct the multiple writes. The embodiment methods do not require "dirty bit" methods or resiling entire cache lines in order to write a portion of the cache lines, as loads or reads may be skipped with SDWO buffer accesses specifically addressed herein.

[0053] Conventional multi-processor computing devices may require special and/or modified architectures, and/or schemes that require segregation between memory ranges or address spaces. For example, some typical schemes require partitioned storage memory units for different applications. As another example, some typical schemes utilize particular address ranges of memory that are pre-associated with particular applications. The embodiment methods do not require such associations of architectures or multiple address spaces. Instead, embodiment methods address SDWO buffer accesses that correspond to a single buffer for an individual application (or program). For example, concurrent SDWO buffer accesses may relate to a buffer that is stored at various locations through one or more memory units of a computing device. The embodiment methods address concurrent writes that are disjoint, such as different data elements of one program that are stored within the same address space. Thus, various embodiment methods do not utilize partitioned or restricted memory spaces. Further, various embodiment methods require synchronization that is not required in various range-based partitioning schemes.

[0054] The descriptions of various embodiments refer to scenarios in which there are concurrent requests to perform SDWO buffer accesses. However, the various embodiments may also provide performance improvements when only one processing unit requests to perform SDWO buffer accesses with regard to an application's buffer. In other words, even without a plurality of concurrent writes to an application's buffer, the multi-processor computing device may set-up a write-set for a processing unit that may in fact avoid unnecessary data transfers. For example, an application executing on the multi-processor computing device may be associated with only a single kernel executing on a GPU. The kernel may be configured to update a few data objects or values within a large buffer associated with the application. Thus, the GPU, via the kernel, may invoke SDWO buffer necessary in which a runtime functionality may configure a write-set on the GPU that does not require a buffer transfer or copy. Once the kernel executing on the GPU identifies buffer indexes and data to be written to the

buffer, the multi-processor computing device may use a CPU coordinator processor to update a valid backing store of the buffer within a memory accessible by the CPU without requiring buffer transfers to or from memory accessible by the GPU.

[0055] FIG. 1 illustrates a multi-processor computing device 100 configured to support SIMD buffer accesses suitable for use with various embodiments. The multi-processor computing device 100 may include various hardware and software components, such as found in some typical computing devices (e.g., smartphones, laptop computers, etc.). For example, the multi-processor computing device 100 may include at least a plurality of processing units 101a-101c (e.g., a CPU 101a, a GPU 101b, and a DSP 101c, etc.).

[0056] The multi-processor computing device 100 may also include one or more memory units 102 of various types, such as main memory, GPU local memory, custom accelerator local memory banks, etc. The memory unit(s) 102 may be configured to store various buffer data 106a-106n associated with one or more applications. For example, the memory unit(s) 102 may store a plurality of data elements (e.g., items in a linked list, subtrees, etc.) for a physics engine, a step simulation program, etc. The buffer data 106a-106n for the application may be distributed throughout the memory unit(s) 102.

[0057] In various embodiments, each processing unit 101a-101c may be connected to only one memory unit and may be configured to access a respective backing store. Processing units 101a-101c may utilize various types of backing stores. For example, one type of backing store may be a block in main memory, another backing store may be an OpenCL-managed data block, and another backing store may be spread over multiple static scratchpad memory banks attached in a custom accelerator using separate memory interfaces. Each backing store may hold the contents of a buffer.

[0058] In various embodiments, the multi-processor computing device 100 may support a runtime functionality 110 (e.g., software routines, services, logic, circuitry, etc.) that is configured to manage buffer access by the various processing units 101a-101c with regard to the application. The runtime functionality 110 may detect one or more concurrent requests to perform SIMD buffer accesses to the buffer data 106a-106n and otherwise perform functions as described herein with reference to FIGS. 2-3B. For example, in response to determining that one or more of the CPU 101a, GPU 101b, and DSP 101c has requested to concurrently write to the buffer data 106a-106n, the runtime functionality 110 may set-up write-sets for the requesting processing units 101a-101c, and identify when the concurrent writes by the processing units 101a-101c have completed (e.g., the conclusion of a wait period). The runtime functionality 110 may further cause various data from the processing units 101a-101c to be written to a valid backing store of the buffer, such as by designating a coordinator processor to receive the write-set data for updating a respective up-to-date backing store. In some embodiments, the runtime functionality 110 may be executed by a main or application processor (e.g., the CPU 101a).

[0059] In some embodiments, the runtime functionality 110 may utilize a translation look aside buffer (TLB) that is used to convert virtual addresses to physical addresses of the memory unit(s) 102. For example, in response to receiving a buffer index from one of the processing units 101a-101c, the runtime functionality 110 may perform a lookup using the TLB 111 to identify the physical address in the memory unit(s) 102 of a first segment of the buffer data 106a. Such physical addresses may be used to write directly to the memory unit(s) 102, such as when the processing units 101a-101c are not able to access memory through virtual addresses. In some embodiments, the runtime functionality 110 may be configured to provide physical addresses to the processing units 101a-101c. For example, when the DSP 101c is only capable of accessing memory via physical addresses, the runtime functionality 110 may provide physical addresses to the DSP 101c, that may be used to conduct direct accesses 120 of the buffer data 106a-106n within the memory unit(s) 102.

[0060] FIGS. 2, 3A-3B illustrate methods 200, 300 and 350, respectively, for a multi-processor computing device to synchronize concurrent buffer accesses that are sparse, disjoint, write-only according to various embodiments. In some embodiments, the multi-processor computing device (e.g., multi-processor computing device 100 in FIG. 1) may execute or otherwise support a runtime functionality (e.g., runtime functionality 110 in FIG. 1) to perform one or more of the operations of the methods. In various embodiments, the operations of the methods 200, 300 and 350 may be performed via one of the processing units of the multi-processor computing device, such as an application processor or CPU (e.g., CPU 101a in FIG. 1). Further, in some embodiments, various operations from the methods 200, 300, 350 may be performed in various orders or sequences than as depicted in the FIGS. 2, 3A-3B.

[0061] FIG. 2 illustrates a method 200 for a multi-processor computing device to synchronize concurrent SIMD buffer accesses to buffer data for an application according to various embodiments. In various embodiments, the application may be a game, physics engine, a step simulation program, or other complex application as described herein [0062].

[0062] In block 202, the processor of the multi-processor computing device may identify (or detect) concurrent requests from one or more processing units to access a buffer associated with an application. The processing units (e.g., a CPU, DSP, GPU, etc.) may each execute software associated with the application (e.g., routines, tasks, kernels, etc.). During the course of execution of the individual tasks, one or more of the processing units may make requests to read and/or write data in the buffer data of the application. For example, the processing units may make API calls to load data objects of a linked list or sub-tree into local backing stores. The multi-processor computing device may detect any such access requests based on an analysis of task code. For example, the multi-processor computing device, via a runtime functionality, may identify invocations of API calls within the code for various tasks associated with processing units. In various embodiments, each of the plurality of concurrent requests to access the buffer may correspond to an access of a separate data structure that uses the buffer as a memory pool, wherein the data structure may be one of a linked list or a tree. For example, each concurrent request may be made by different processing units accessing separate linked lists in separate sub-trees of a tree, all allocated over the same buffer in memory pool.

[0063] In block 203, the processor of the multi-processor computing device may identify one of the identified concurrent requests to access the buffer that are sparse, des-

joint, and write-only (SDWO). For example, the multi-processor computing device may identify concurrent write requests that correspond to data objects in a linked list or a subtree. In some embodiments, the multi-processor computing device may make the identification based on the type of API call employed by the tasks for task code executing on the processing units. For example, the multi-processor computing device may determine whether the concurrent requests correspond to a predefined “special” SDWO API call for SDWO buffer accesses (e.g., a SDWO buffer access request). As another example, the multi-processor computing device may determine whether task code on processing units invoke an “sdwo_writer” API call. As described, such SDWO API calls may be declared before launch or execution of application code for task code, such as API calls declared by an application programmer.

[1064] In some embodiments, the identification in block 203 may be based on whether the concurrent requests are associated with predefined buffer indices. For example, the multi-processor computing device may configure hardware of the multi-processor computing device to detect write operations to the buffer by the plurality of processing units via typical API calls. In such a case, the multi-processor computing device may convert such write operations to the buffer to write operations to write-sets.

[1065] In block 204, the processor of the multi-processor computing device may identify any of the identified concurrent requests to access the buffer that are not SDWO (i.e., non-SDWO requests). For example, the multi-processor computing device may determine whether any of the concurrent requests relate to API calls that are not known as SDWO-related.

[1066] For the non-SDWO requests, the multi-processor computing device may conduct regular operations to provide buffer access to the associated processing unit(s) and/or task(s). Accordingly, in block 205, the processor of the multi-processor computing device may execute the identified non-SDWO requests via regular routine mechanisms. For example, in response to determining that certain concurrent requests to access the buffer are not SDWO, the processor of the multi-processor computing device may serially execute such buffer requests by performing typical operations that may potentially cause or utilize task serialization, cache thrashing, and/or full-buffer copies. Tasks running on processing units that do not utilize SDWO operations may execute as normal, with the usual penalties (e.g., full-buffer-copies, etc.). In some cases, the multi-processor computing device (e.g., via a routine functionality) may serially transfer the buffer to each of the one or more processing units making the concurrent request(s) to access the buffer. In other words, the multi-processor computing device may perform typical serialization operations to enable the processing units to access the buffer individually in sequence. Such typical operations may create redundant copies and transfers of the buffer data (e.g., transfer to various backing stores of the processing units). Due to the potential differences in the processing units, the transfers of the buffer may be serial in order to ensure coherency.

[1067] In response to identifying concurrent request(s) to access the buffer that are SDWO, the processor of the multi-processor computing device may configure (or set-up) a write-set for each of the one or more processing units associated with the identified, concurrent SDWO request(s) in block 206. For example, the multi-processor computing

device may configure a per-device write-set on each of the one or more processing units executing tasks that requested SDWO buffer accesses. As described, each write-set may be a set of {index, data_value} pairs. Setting-up such write-sets for each of the processing units may result in less system traffic than transferring copies of the buffer to each of the processing units.

[1068] The operations in blocks 207-209 may constitute a loop or period during which the multi-processor may be configured to wait a predefined period or otherwise until all concurrent SDWO operations have been addressed by the various processing units with regard to the respective write-sets. In block 207, the multi-processor computing device may concurrently execute tasks associated with the identified SDWO request(s).

[1069] In block 208, the processor of the multi-processor computing device may receive a write-set from one or more of the plurality of processing units. For example, the multi-processor computing device may receive a set of buffer index, data value pairs from one or more processing units, wherein each write-set may be a set of buffer index, value pairs. Write-set data may be received as tasks on processing units complete. However, in some iterations of the operational loop of blocks 207-209, no write-set data may be received as processing units may still be performing task operations to generate buffer index and data information for transmission.

[1070] In determination block 209, the processor of the multi-processor computing device may determine whether all concurrent SDWO requests have completed. In other words, the multi-processor computing device may wait until write-set data is received from all processing units associated with the identified concurrent SDWO requests and/or until a period for executing the tasks associated with the identified SDWO requests has elapsed in otherwise finished. For example, the multi-processor computing device may wait until all of the plurality of processing units have completed performing respective tasks for writing to associated write-sets. As another example, the multi-processor computing device may determine the period is complete in response to identifying a marker within code of the application indicating completion of requested buffer accesses by the plurality of processing units.

[1071] In some embodiments, the multi-processor computing device may determine that all concurrent SDWO requests have completed (or that the period has elapsed) in response to detecting that each of the plurality of processing units associated with the identified SDWO requests has made a specialized API call associated with the SDWO operation (e.g., *sdwo_writer*). Such an API call by an individual processing unit may not indicate that the period has ended for all other processing units associated with other identified SDWO requests. Instead, an API call such as “*sdwo_waitbl*” may indicate that the multi-processor computing device has access to updated write-set data after a program point corresponding to the invocation of “*SDWO_writer*”. In some embodiments, the multi-processor computing device may wait until the period has elapsed, and the various write-sets have respective updated results.

[1072] In some embodiments, there may be alternative ways for the multi-processor computing device to utilize data from write-sets of corresponding processing units. For example, the multi-processor computing device may wait for all tasks accessing write-sets to complete with respective

operations (e.g., `hold unit wait (unit)`, `wait (unit)`, etc., is called by tasks 11-12, etc.). As another example, the multi-processor computing device may wait directly on the buffer, blocking until all tasks accessing write-sets that were launched prior to a `swo_wait()` call have completed. In some embodiments, the multi-processor computing device may determine that the period has completed in response to identifying a marker within code of the application that indicates the completion of requested buffer accesses by the plurality of processing units (i.e., a programmer-defined end to a concurrent write period).

[0073] In response to determining that all the concurrent SDWO request(s) have not completed (i.e., determination block 209 “No”), the multi-processor computing device may continue executing the tasks in block 207. In some cases, the multi-processor computing device may receive buffer index and data from write-sets of various processing units during subsequent operations of block 208. In other words, the various executing tasks of the processing units may intermittently send respective write-sets with data to be written to the buffer.

[0074] In response to determining that all the concurrent SDWO request(s) have completed (i.e., determination block 209 “Yes”) the processor of the multi-processor computing device may obtain a collection of buffer index and data from the write-sets of one or more of the plurality of processing units in block 210. In other words, the multi-processor computing device may obtain and merge all write-set data received in response to setting up the various write-sets. For example, the multi-processor computing device may obtain one or a plurality of pairs of data from each of the processing units that indicates data for updating the buffer (e.g., a collection of `(index, element-value)` pairs). As described above, the write-sets from each of the processing units associated with SDWO requests (and data therein) may become available to the multi-processor computing device at various times, such as intermittently during the operations of blocks 207-209.

[0075] In block 212, the processor of the multi-processor computing device may write to the buffer using the obtained buffer index(es) and data from the write-set(s) of the one or more processing units. For example, the multi-processor computing device may write data to a backing store (including the latest version of the buffer data (e.g., write directly to memory locations associated with the buffer of the application).

[0076] In some embodiments, the operations of blocks 210-212 may be performed via a coordinator processor. Accordingly, in some embodiments, the multi-processor computing device may perform operations to identify a coordinator processor. As described, such a central processor may be a device or other functionality that has a backing store that is storing valid data for the buffer at the time of the concurrent write requests. For example, such a coordinator processor may be identified as a CPU, that corresponds to a backing store with a valid representation of the buffer data. In such instances, the multi-processor computing device may transmit any information regarding the write-sets to the coordinator processor for writing to the backing store in response to determining that the synchronization period is complete. In response, the coordinator processor may send write-set information to write to the backing store. For example, a designated CPU coordinator processor may receive the write-sets (e.g., collections of index, element-

value pairs) from each processing unit when associated tasks complete and then the coordinator processor may update a corresponding backing store with the data from the received write-sets, designating the backing store as storing the most up-to-date data.

[0077] FIGS. 3A-3B illustrate embodiment methods 300, 350 for enabling processing units to directly address buffer data when possible. For example, when the buffer consists of large elements (e.g., each buffer element is a large “string” data object), the multi-processor computing device may incur significant costs (e.g., increases in power demand and/or increases in processing overheads when copying the written data elements as part of transmitting the write sets to a designated processing unit 208) that will update the buffer with the write sets 210, 212. Therefore, the multi-processor computing device may leverage any capabilities of the processing units to directly access memory units directly (e.g., SVM capabilities) in order to avoid unnecessary buffer copying. The goal is to avoid the double transmission of large data elements over the system bus (i.e., first when the write sets are received, and second when write set data is written to the buffer) while preserving the benefits of SDWO. Instead, various embodiments allow the received write sets to carry alternative information that does not include the written buffer element data.

[0078] FIG. 3A illustrates a method 300 for a multi-processor computing device to synchronize concurrent SDWO buffer accesses to buffer data for an application according to various embodiments. The method 300 is similar to the method 200 described with reference to FIG. 2, except that the method 300 includes operations for performing virtual address look-ups for processing units that may be limited to operate on physical addresses. When a concurrent processing unit completes execution, only the buffer indices of the buffer elements written are received from the processing unit (302), without the buffer element data as was the case in 208. The multi-processor computing device may look up the virtual address of a received buffer index in a backing store holding the latest contents of the buffer, translate the virtual address to a physical address, and transmit the physical address back to processing unit. The processing unit may use the physical address to write the corresponding buffer element directly to the backing store, thereby avoiding the double transmission of a large buffer element. In another embodiment, in order to improve on the efficiency of writing to the buffer of the application, the multi-processor computing device may utilize any memory capabilities of the various processing units to enable direct writing to the buffer data. For example, as shown in FIG. 3B, some processing units may have SVM capabilities, while other processing units may not. Therefore, the multi-processor computing device may automatically determine that regular mechanisms should be used for accessing the buffer directly for devices with SVM capabilities, but determine that non-SVM processing units should use write-sets to perform SDWO buffer accesses for an application. In this manner, writing to the buffer may be done by some of the processing units themselves, in contrast to the writing being accomplished exclusively by a coordinator processor as described herein.

[0079] Referring to FIG. 3A, the operations of blocks 202-207, 209 may be similar to the operations of like numbered blocks of the method 200 described with reference to FIG. 2. In block 302, the processor of the multi-

processor computing device may receive index-set for buffer elements of the buffer from the processing unit(s). In block 304, the processor of the multi-processor computing device may perform a virtual address translation to identify physical addresses(s) for the buffer elements corresponding to the received index-set.

[0080] After determining that all concurrent SDWO request(s) have completed (or a period for executing the concurrent SDWO requests ends) (e.g., determination block 219 “Yes”), the processor of the multi-processor computing device may transmit the physical addresses(s) to the processing unit(s) for direct writing to the buffer (e.g., direct writing of the buffer element value to the buffer) in block 306. Such physical address(es) may then be employed by processing unit(s) to perform SDWO buffer access(es) without requiring the double transmission of possibly large buffer elements.

[0081] FIG. 3B illustrates a method 350 for a multi-processor computing device to synchronize concurrent SDWO buffer accesses to buffer slots for an application according to various embodiments. The method 350 is similar to the method 200 described with reference to FIG. 2, except that the method 350 includes operations for enabling direct writing to a buffer by processing units configured with the appropriate functionalities (e.g., SVM functionality). The multi-processor computing device (e.g., via a coordinator processor as described herein) may serially indicate to the processing units associated with concurrent SDWO requests when these processing units can directly write respective write-set(s) in a single backing store with up-to-date buffer data. For example, the multi-processor computing device (e.g., via a coordinator processor) may monitor for reception of a message from a processing unit that indicates when the processing unit has completed execution of a SDWO buffer access. The processing unit may wait to receive a response from the multi-processor computing device (e.g., via the coordinator processor) indicating that the processing unit may directly update a backing store with a respective write-set. In response to receiving such a signal (e.g., from the coordinator processor), the processing unit may perform driver-managed flushes, explicit cache flushes, and/or configure a translation lookaside buffer (TLB) to consistently update the backing store. In this way, write-sets for some processing units may not be transmitted (e.g., to a coordinator processor) and some processing units may write directly to memory when all of the concurrent processing units complete execution.

[0082] Referring to FIG. 3B, the operations of blocks 202-207 may be similar to the operations of like numbered blocks of the method 200 described with reference to FIG. 2. In block 351, the processor of the multi-processor computing device may receive write sets from one or more processing units that cannot directly write buffer data.

[0083] Once all concurrent requests are completed (i.e., determination block 209 “Yes”), the processor of the multi-processor computing device may determine whether any of the processing units that requested the concurrent SDWO buffer access(es) (or writes) are capable of directly writing buffer data (or addressing the buffer directly) in determination block 352. For example, the multi-processor computing device may determine whether the processing units can address virtual addresses directly but may lack full SVM cache coherence. Any such processing units may be directed

to perform write-backs directly to memory in order to save redundant copying of the buffer.

[0084] In response to determining that processing unit(s) that requested concurrent SDWO writes are not capable of addressing the buffer directly (i.e., determination block 352 “No”), the multi-processor computing device may perform the operations of blocks 210-212 as described with reference to FIG. 2 with regard to the write-sets of those processing unit(s).

[0085] In response to determining that processing unit(s) that requested concurrent writes are capable of addressing the buffer directly (i.e., determination block 352 “Yes”), the processor of the multi-processor computing device may serially transmit a signal to each of those processing units(s) for direct writing to the buffer in block 354. In other words, these processing unit(s) may be configured to concurrently execute (in the process constituting their respective write-set), and upon completion or execution wait for a signal to serially write their write-set directly to the buffer (e.g., a backing store with up-to-date data of the buffer). The signal may cause each of these processing unit(s) to write respective write-set data to a single backing store (e.g., a backing store of the buffer data associated with a designated coordinator processor). For example, a coordinator processor (e.g., CPU having a local backing store of the buffer) may signal the processing units to serialize the device write-backs, punctuated by appropriate driver flushes.

[0086] In some embodiments, the multi-processor computing device may perform operations of the method 350 to handle concurrent SDWO requests by processing unit(s) eligible to write directly to a buffer (or backing store) and/or processing unit(s) capable of providing write-set data to a coordinator processor for direct writing to the backing store [0087]. The embodiment operations described with reference to FIGS. 3A-3B provide operations that may help the multi-processor computing device avoid “double copying” of updated data entries (e.g., avoid both a first copy to a coordinator processor and then a second copy to a backing store by the coordinator processor). With the methods 300 and 350, the multi-processor computing device may instead enable processing units to write data directly to a buffer. Such embodiment methods may be useful for situations in which individual buffer entries are very large.

[0088] In some cases, cache-coherence may not be necessary to ensure consistency. Accordingly, dividing cache-coherence functionalities policies may save considerable power when not necessary for consistency.

[0089] FIG. 4 illustrates a non-limiting example of pseudocode 400 that may be executed by a processor of a multi-processor computing device to enable concurrent buffer accesses that are sparse, disjoint, and write-only according to some embodiments. For example, the pseudocode 400 may be performed by a runtime functionality to manage one or more tasks or other executable elements configured to cause SDWO buffer accesses corresponding to a particular application’s buffer.

[0090] The pseudocode 400 may include a first section 402 that creates a buffer for the application (e.g., a buffer if that includes 200000 data objects or entries for a game physics engine, etc.).

[0091] The pseudocode 400 may include a second section 404 for launching tasks, kernels, and/or other operations to be performed by various processing units of the multi-processor computing device (e.g., CPU, GPU, DSP, custom

hardware accelerating, etc.). In launching each task, the multi-processor computing device may provide copies of data for reading and/or writing read buffers "a", "c", "d", "e", etc.) and configure write-sets associated with the sdwo buffer "b". In other words, the multi-processor computing device may provide some copies of data to the processing units, but may only configure write-sets associated with the sdwo buffer.

[0092] The pseudocode 400 may include a third section 406 that causes the multi-processor computing device to wait until each of the processing units has provided at least a buffer index and data based on executions of respective code (e.g., tasks). For example, the multi-processor computing device may wait until the CPU has returned a buffer index and data after executing a "cpu function" task, the GPU has returned a buffer index and data after executing a "gpu kernel" kernel, the DSP has returned a buffer index and data after executing a "dsp_kernel" kernel, and a custom hardware accelerating has returned a buffer index and data after executing a "custom_hw_func" task.

[0093] When all tasks complete, the multi-processor computing device may have received (e.g., intermittently, etc.) all associated write-sets from the various processing units and respective tasks and may then update a backing store to update the buffer. In some embodiments, the operations in the third section 406 may include providing received write-set data to a coordinator processor device as well as operations to update a backing store (e.g., updating a common memory for the buffer).

[0094] FIGS. 5A-5B illustrate non-limiting examples of pseudocode 500, 550 that may be performed by processing units of a multi-processor computing device according to some embodiments. The pseudocode 500, 550 may be called by at least a CPU, DSP, or GPU in response to the multi-processor computing device performing the operations of the sections 402-404 of the pseudocode 400 illustrated in FIG. 4. In some embodiments, the pseudocode 500, 550 may be implemented for various routine systems, such as OpenCL.

[0095] In FIG. 5A, the pseudocode 500 may include a function ("f") to be performed by a processing unit (e.g., GPU, etc.). The function may utilize input parameters of a normal buffer "c" and a write-set for SDWO buffer "b" (e.g., a pointer to a write-set). While performing the function, the processing unit may eventually make a special API call using the write-set (e.g., sdwo_write...). Such an API call may include providing parameters that define the write-set (e.g., b, sdwo), the index of the buffer to write to (e.g., "c"), and data to write in that buffer index (e.g., values). In other words, sdwo_write... may be an API call used from within tasks to write a buffer element.

[0096] The pseudocode 550 in FIG. 5B resembles a typical function that may be performed by a processing unit. For example, the pseudocode 550 may not include any special API call to invoke or otherwise access a write-set as described herein. However, when the multi-processor computing device is modified to include specialized hardware that is capable of identifying requests for SDWO buffer accesses by processing units, the pseudocode 550 may be used to invoke SDWO optimization according to various embodiments.

[0097] Various forms of computing devices, including personal computers and laptop computers, may be used to implement the various embodiments. An example of a

multi-processor computing device 600 suitable for implementing the various embodiments is illustrated in FIG. 6. In various embodiments, the multi-processor computing device 600 may include a processor 601 coupled to a touch screen controller 604 and an internal memory 602. The processor 601 may be one or more multicore ICs designated for general or specific processing tasks. The internal memory 602 may be volatile and/or non-volatile memory, and may also be secure and/or encrypted memory, or unsecure and/or unencrypted memory, or any combination thereof. The touch screen controller 604 and the processor 601 may also be coupled to a touch screen panel 612, such as a resistive-sensing touch screen, capacitive-sensing touch screen, infrared sensing touch screen, etc. The multi-processor computing device 600 may have one or more radio signal transceivers 608 (e.g., Bluetooth®, Zigbee®, Wi-Fi®, radio frequency (RF) radio) and antennae 610, for sending and receiving, coupled to each other and/or to the processor 601. The transceivers 608 and antennae 610 may be used with the above-mentioned circuitry to implement the various wireless transmission protocol stacks and interfaces. The multi-processor computing device 600 may include a cellular network wireless modem chip 616 that enables communication via a cellular network and is coupled to the processor. The multi-processor computing device 600 may include a peripheral device connection interface 618 coupled to the processor 601. The peripheral device connection interface 618 may be singularly configured to accept one type of connection, or multiply configured to accept various types of physical and communication connections, common or proprietary, such as USB, FireWire, Thunderbolt, or PCIe. The peripheral device connection interface 618 may also be coupled to a similarly configured peripheral device connection port (not shown). The multi-processor computing device 600 may also include speakers 614 for providing audio outputs. The multi-processor computing device 600 may also include a housing 620, constructed of a plastic, metal, or a combination of materials, for containing all or some of the components discussed herein. The multi-processor computing device 600 may include a power source 622 coupled to the processor 601, such as a disposable or rechargeable battery. The rechargeable battery may also be coupled to the peripheral device connection port to receive a charging current from a source external to the multi-processor computing device 600.

[0098] The various embodiments illustrated and described are provided merely as examples to illustrate various features of the claims. However, features shown and described with respect to any given embodiment are not necessarily limited to the associated embodiment and may be used or combined with other embodiments that are known and described further. Further, the claims are not intended to be limited by any one example embodiment.

[0099] The various processors described herein may be any programmable microprocessor, micro-computer or multiple processor chip or chips that can be configured by software instructions (applications) to perform a variety of functions, including the functions of the various embodiments described herein. In the various devices, multiple processors may be provided, such as one processor dedicated to wireless communication functions and one processor dedicated to running other applications. Typically, software applications may be stored in internal memory before being accessed and loaded into the processors. The process-

sors may include internal memory sufficient to store the application software instructions. In many devices the internal memory may be a volatile or nonvolatile memory, such as flash memory, or a mixture of both. For the purposes of this description, a general reference to memory refers to memory accessible by the processors including internal memory or removable memory plugged into the various devices and memory within the processors.

[0100] The foregoing method descriptions and the process flow diagrams are provided merely as illustrative examples and are not intended to require or imply that the operations of the various embodiments must be performed in the order presented. As will be appreciated by one of skill in the art the order of operations in the foregoing embodiments may be performed in any order. Words such as "then," "then," "next," etc., are not intended to limit the order of the operations; these words are simply used to guide the reader through the description of the methods. Further, any reference to claim elements in the singular, for example, using the articles "a," "an" or "the," is not to be construed as limiting the element to the singular.

[0101] The various illustrative logical blocks, modules, circuits, and algorithm operations described in connection with the embodiments disclosed herein may be implemented as electronic hardware, computer software, or combinations thereof. To clearly illustrate this interchangeability of hardware and software, various illustrative components, blocks, modules, circuits, and operations have been described above, generally in terms of their functionality. Whether such functionality is implemented as hardware or software depends upon the particular application and design constraints imposed on the overall system. Skilled artisans may implement the described functionality in varying ways for each particular application, but such implementation decisions should not be interpreted as causing a departure from the scope of the present claims.

[0102] The hardware used to implement the various illustrative logics, logical blocks, modules, and circuits described in connection with the embodiments disclosed herein may be implemented or performed with a general purpose processor, a digital signal processor (DSP), an application specific integrated circuit (ASIC), a field programmable gate array (FPGA) or other programmable logic device, discrete gate or transistor logic, discrete hardware components, or any combination thereof designed to perform the functions described herein. A general-purpose processor may be a microprocessor, but, in the alternative, the processor may be any conventional processor, controller, microcontroller, or state machine. A processor may also be implemented as a combination of computing devices, e.g., a combination of a DSP and a microprocessor; a plurality of microprocessors, one or more microprocessors in conjunction with a DSP core, or any other such configuration. Alternatively, some operations or methods may be performed by circuitry that is specific to a given function.

[0103] In one or more embodiments, the functions described may be implemented in hardware, software, firmware, or any combination thereof. If implemented in software, the functions may be stored on or transmitted over as one or more instructions or code on a non-transitory processor-readable, computer-readable, or server-readable medium or a non-transitory processor-readable storage medium. The operations of a method or algorithm disclosed herein may be embodied in a processor-executable software

module or processor-executable software instructions, which may reside on a non-transitory computer-readable storage medium, a non-transitory server-readable storage medium, and/or a non-transitory processor-readable storage medium. In various embodiments, such instructions may be stored processor-executable instructions or stored processor-executable software instructions. Tangible, non-transitory computer-readable storage media may be any available media that may be accessed by a computer. By way of example, and not limitation, such non-transitory computer-readable media may comprise RAM, ROM, EEPROM, CD-ROM or other optical disk storage, magnetic disk storage or other magnetic storage devices, or any other medium that may be used to store desired program code in the form of instructions or data structures and that may be accessed by a computer. Disk and disc, as used herein, includes compact disc (CD), laser disc, optical disc, digital versatile disc (DVD), floppy disk, and Blu-ray disc, where disks usually reproduce data magnetically, while discs reproduce data optically with lasers. Contributions of the above should also be included within the scope of non-transitory computer-readable media. Additionally, the operations of a method or algorithm may reside as one or any combination or set of codes and/or instructions on a tangible, non-transitory processor-readable storage medium and/or computer-readable medium, which may be incorporated into a computer program product.

[0104] The preceding description of the disclosed embodiments is provided to enable any person skilled in the art to make or use the embodiment methods of the claims. Various modifications to these embodiments will be readily apparent to those skilled in the art, and the generic principles defined herein may be applied to other embodiments without departing from the spirit or scope of the claims. Thus, the present disclosure is not intended to be limited to the embodiments shown herein but is to be accorded the widest scope consistent with the following claims and the principles and novel features disclosed herein.

What is claimed is:

1. A method for a multi-processor computing device to merge concurrent writes from a plurality of processing units to a buffer associated with an application, comprising:
identifying, by a processor of the multi-processor computing device, a plurality of concurrent requests to access the buffer that are sparse, disjoint, and write-only (SDWO);
consuming, by the processor, a write-set for each of the plurality of processing units;
executing, by the plurality of processing units, the plurality of concurrent requests to access the buffer using the write-set;
determining, by the processor, whether each of the plurality of concurrent requests to access the buffer is complete;
obtaining, by the processor, a buffer index and data via the write-set of each of the plurality of processing units; and
writing, by the processor, to the buffer using the obtained buffer index and data via the write-set of each of the plurality of processing units in response to determining that each of the plurality of concurrent requests to access the buffer is complete.
2. The method of claim 1, wherein identifying, by the processor of the multi-processor computing device, the

- Plurality of concurrent requests to access the buffer that are sparse, disjoint, and write-only (SDWO) comprises:
- identifying, by the processor, an SDWO application programming interface (API) call of a task executing on each of the plurality of processing units;
 - The method of claim 1, further comprising:
 - receiving, by the processor, an index for a buffer element from one of the plurality of processing units; and
 - performing, by the processor, a virtual address translation to identify a physical address for the buffer element corresponding to the received index;
 - The method of claim 1, further comprising transmitting, by the processor, the physical address to the one of the plurality of processing units for direct writing of a buffer element value to the buffer in response to performing the virtual address translation;
 - The method of claim 1, wherein determining, by the processor, whether each of the plurality of concurrent requests to access the buffer is complete comprises:
 - waiting, by the processor, until all of the plurality of processing units have completed performing respective tasks for writing to associated write-sets;
 - The method of claim 1, wherein determining, by the processor, whether each of the plurality of concurrent requests to access the buffer is complete comprises:
 - identifying, by the processor, a marker within code of the application indicating completion of requested buffer accesses by the plurality of processing units;
 - The method of claim 1, further comprising:
 - determining, by the processor, whether the plurality of processing units are configured to directly write to the buffer, and
 - serially transmitting, by the processor, a signal to each of the plurality of processing units in response to determining that the plurality of processing units are configured to directly write to the buffer, wherein the signal indicates when each of the plurality of processing units can directly write to the buffer;
 - The method of claim 1, further comprising:
 - identifying, by the processor, another plurality of concurrent requests to access the buffer that are not sparse, disjoint, and write-only; and
 - serially transferring, by the processor, the buffer to each processing unit associated with the another plurality of concurrent requests such that the another plurality of concurrent requests are executed serially;
 - The method of claim 1, further comprising:
 - identifying, by the processor, a coordinator processor as having a backing store with valid data for the buffer, and
 - transmitting, via the processor, data of the write-sets to the coordinator processor for writing to the backing store in response to determining that each of the plurality of concurrent requests to access the buffer is complete;
 - The method of claim 8, wherein writing, by the processor, to the buffer using the obtained buffer index and data via the write-set of each of the plurality of processing units in response to determining that each of the plurality of concurrent requests to access the buffer is complete comprises:
 - writing, by the coordinator processor, to the backing store associated with the coordinator processor;
 - The method of claim 1, wherein each of the plurality of concurrent requests to access the buffer corresponds to an access of a separate data structure that uses the buffer as a memory pool, wherein the data structure is one of a linked list or a tree:
11. The method of claim 1, wherein the application is a step simulation program;
 12. The method of claim 1, wherein the processor is executing a runtime system;
 13. The method of claim 1, wherein identifying, by the processor, of the multi-processor computing device, the plurality of concurrent requests to access the buffer that are sparse, disjoint, and write-only (SDWO) comprises:
 - determining, by the processor, that the plurality of concurrent requests are requests to write the same value to the same buffer index;
 14. A computing device, comprising:
 - a memory; and
 - a plurality of processing units including a processor coupled to the memory and configured with processor-executable instructions to perform operations comprising:
 - identifying a plurality of concurrent requests to access a buffer that are sparse, disjoint, and write-only (SDWO), wherein the buffer is associated with an application;
 - configuring a write-set for each of the plurality of processing units;
 - executing the plurality of concurrent requests to access the buffer using the write-sets;
 - determining whether each of the plurality of concurrent requests to access the buffer is complete;
 - obtaining a buffer index and data via the write-set of each of the plurality of processing units, and
 - writing to the buffer using the obtained buffer index and data via the write-set of each of the plurality of processing units in response to determining that each of the plurality of concurrent requests to access the buffer is complete;
 15. The computing device of claim 14, wherein the processor is configured with processor-executable instructions to perform operations such that identifying the plurality of concurrent requests to access the buffer that are sparse, disjoint, and write-only (SDWO) comprises:
 - identifying an SDWO application programming interface (API) call of a task executing on each of the plurality of processing units;
 16. The computing device of claim 14, wherein the processor is configured with processor-executable instructions to perform operations further comprising:
 - receiving an index for a buffer element from one of the plurality of processing units;
 - performing a virtual address translation to identify a physical address for the buffer element corresponding to the received index; and
 - transmitting the physical address to the one of the plurality of processing units for direct writing of a buffer element value to the buffer in response to performing the virtual address translation;
 17. The computing device of claim 14, wherein the processor is configured with processor-executable instructions to perform operations such that determining whether each of the plurality of concurrent requests to access the buffer is complete comprises:

- waiting until all of the plurality of processing units have completed performing respective tasks for writing to associated write-sets;
18. The computing device of claim 14, wherein the processor is configured with processor-executable instructions to perform operations such that determining whether each of the plurality of concurrent requests to access the buffer is complete comprises:
- identifying a marker within code of the application indicating completion of requested buffer accesses by the plurality of processing units;
19. The computing device of claim 14, wherein the processor is configured with processor-executable instructions to perform operations further comprising:
- determining whether the plurality of processing units are configured to directly write to the buffer, and serially transmitting a signal to each of the plurality of processing units in response to determining that the plurality of processing units are configured to directly write to the buffer, wherein the signal indicates when each of the plurality of processing units can directly write to the buffer;
20. The computing device of claim 14, wherein the processor is configured with processor-executable instructions to perform operations further comprising:
- identifying another plurality of concurrent requests to access the buffer that are not sparse, disjoint, and write-only, and serially transferring the buffer to each processing unit associated with the another plurality of concurrent requests such that the another plurality of concurrent requests are executed serially;
21. The computing device of claim 14, wherein the processor is configured with processor-executable instructions to perform operations further comprising:
- identifying a coordinator processor as having a backing store with valid data for the buffer; and transmitting data of the write-sets to the coordinator processor for writing to the backing store in response to determining that each of the plurality of concurrent requests to access the buffer is complete;
22. The computing device of claim 21, wherein the processor is configured with processor-executable instructions to perform operations such that writing to the buffer using the obtained buffer index and data via the write-set of each of the plurality of processing units in response to determining that each of the plurality of concurrent requests to access the buffer is complete comprises:
- writing to the backing store associated with the coordinator processor;
23. The computing device of claim 14, wherein each of the plurality of concurrent requests to access the buffer corresponds to an access of a separate data structure that uses the buffer as a memory pool, wherein the data structure is one of a linked list or a tree;
24. The computing device of claim 14, wherein the application is a step simulation program;
25. The computing device of claim 14, wherein the processor is executing a runtime system;
26. The computing device of claim 14, wherein the processor is configured with processor-executable instructions to perform operations such that identifying the plurality of concurrent requests to access the buffer that are sparse, disjoint, and write-only (SDWO) comprises:
- determining that the plurality of concurrent requests are requests to write the same value to the same buffer index;
27. A non-transitory processor-readable storage medium having stored thereto processor-executable instructions configured to cause a processor of a computing device to perform operations comprising:
- identifying a plurality of concurrent requests to access a buffer that are sparse, disjoint, and write-only (SDWO), wherein the buffer is associated with an application;
 - configuring a write-set for each of a plurality of processing units within the computing device;
 - executing the plurality of concurrent requests to access the buffer using the write-sets;
 - determining whether each of the plurality of concurrent requests to access the buffer is complete;
 - obtaining a buffer index and data via the write-set of each of the plurality of processing units; and
 - writing to the buffer using the obtained buffer index and data via the write-set of each of the plurality of processing units in response to determining that each of the plurality of concurrent requests to access the buffer is complete;
28. The non-transitory processor-readable storage medium of claim 27, wherein the stored processor-executable instructions are configured to cause the processor of the computing device to perform operations such that identifying the plurality of concurrent requests to access the buffer that are sparse, disjoint, and write-only (SDWO) comprises:
- identifying an SDWO application programming interface (API) call of a task executing on each of the plurality of processing units;
29. The non-transitory processor-readable storage medium of claim 27, wherein the stored processor-executable instructions are configured to cause the processor of the computing device to perform operations further comprising:
- receiving an index for a buffer element from one of the plurality of processing units;
 - performing a virtual address translation to identify a physical address for the buffer element corresponding to the received index; and
 - transmitting the physical address to one of the plurality of processing units for direct writing of a buffer element value to the buffer in response to performing the virtual address translation;
30. A computing device, comprising:
- means for identifying a plurality of concurrent requests to access a buffer that are sparse, disjoint, and write-only (SDWO), wherein the buffer is associated with an application;
 - means for configuring a write-set for each of a plurality of processing units within the computing device;
 - means for executing the plurality of concurrent requests to access the buffer using the write-sets;
 - means for determining whether each of the plurality of concurrent requests to access the buffer is complete;
 - means for obtaining a buffer index and data via the write-set of each of the plurality of processing units; and
 - means for writing to the buffer using the obtained buffer index and data via the write-set of each of the plurality of processing units.

of processing units in response to determining that each of the plurality of concurrent requests to access the buffer is complete.

* * *



US 20160216969A1

(19) United States

(21) Patent Application Publication

Suarez Gracia et al.

(10) Pub. No.: US 2016/0216969 A1

(14) Pub. Date:

Jul. 28, 2016

(54) SYSTEM AND METHOD FOR ADAPTIVELY
MANAGING REGISTERS IN AN
INSTRUCTION PROCESSOR(52) U.S. CL.
CPC G06F 9/30109 (2013.01), G06F 9/30141
(2013.01)(71) Applicant: QUALCOMM Incorporated, San
Diego, CA (US)

(57) ABSTRACT

(72) Inventors: Dario Suarez Gracia, Palo Alto, CA
(US); Behnam Rahmati, San Jose, CA
(US)

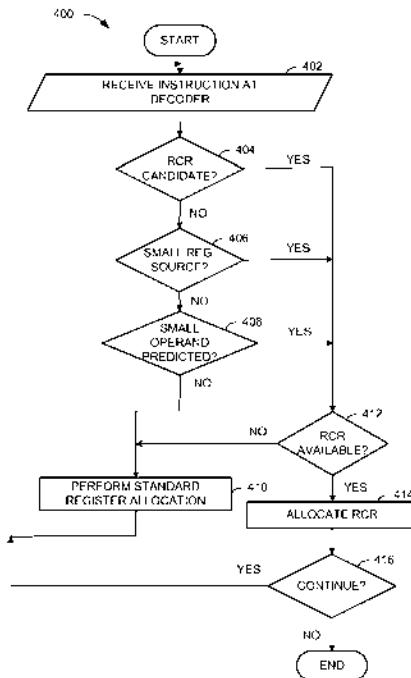
(21) Appl. No.: 14/607,270

(22) Filed: Jan. 28, 2015

Publication Classification

(51) Int. Cl.
G06F 9/30
(2006.01)

Systems and methods for adaptively managing registers in an instruction processor are disclosed. The system identifies one or more registers with operable cells. An operand manager identifies a set of operable cells within the one or more registers with inoperable cells and determines if a present instruction will use an operand that can be supported by the set of operable cells. When the set of operable cells can support the operand, the operand manager generates an assignment which is communicated to a register file manager.



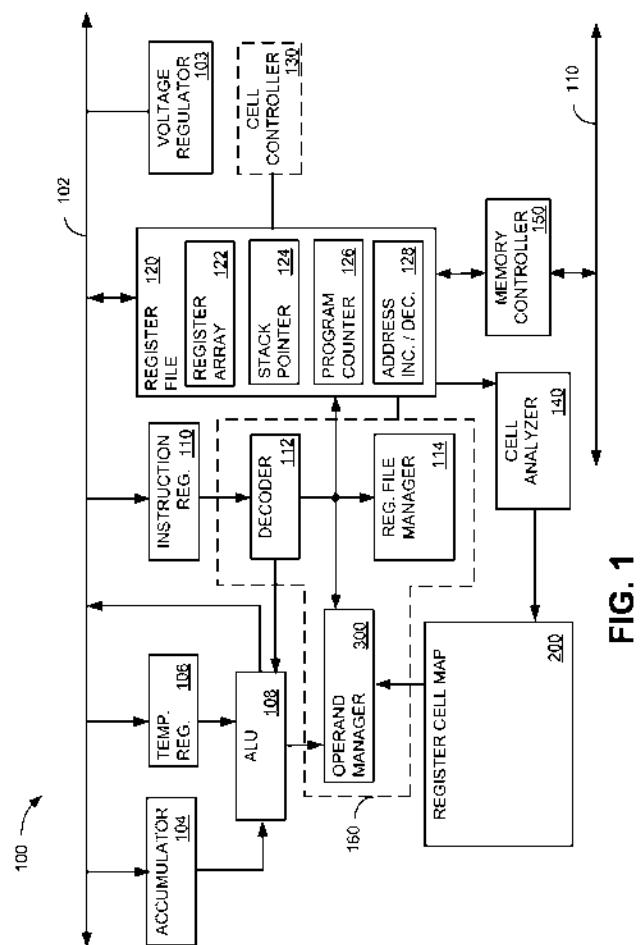


FIG. 1

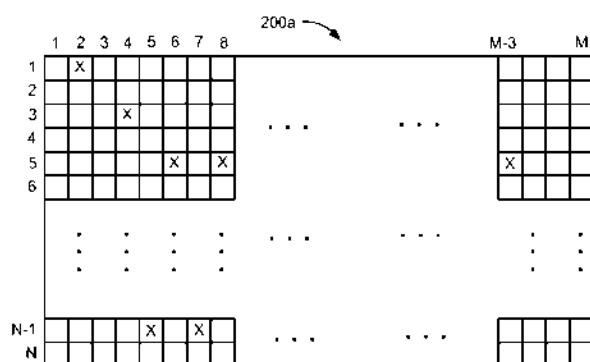


FIG. 2A

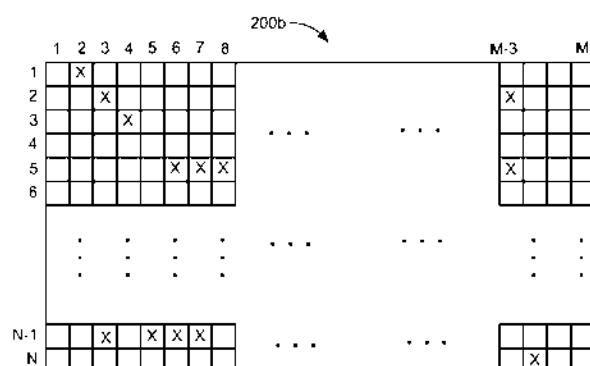


FIG. 2B

1	2	3	4	5	6	7	8								
1	X	X	X	X	X	X	X								
2	X	X	X	X	X	X	X								
3	X	X	X	X	X	X	X								
4	X	X	X	X	X	X	X								
5	X	X	X	X	X	X	X								
6	X	X	X	X	X	X	X								
⋮								⋮			⋮			⋮	
N-1	X	X	X	X	X	X	X								
N	X	X	X	X	X	X	X								

FIG. 2C

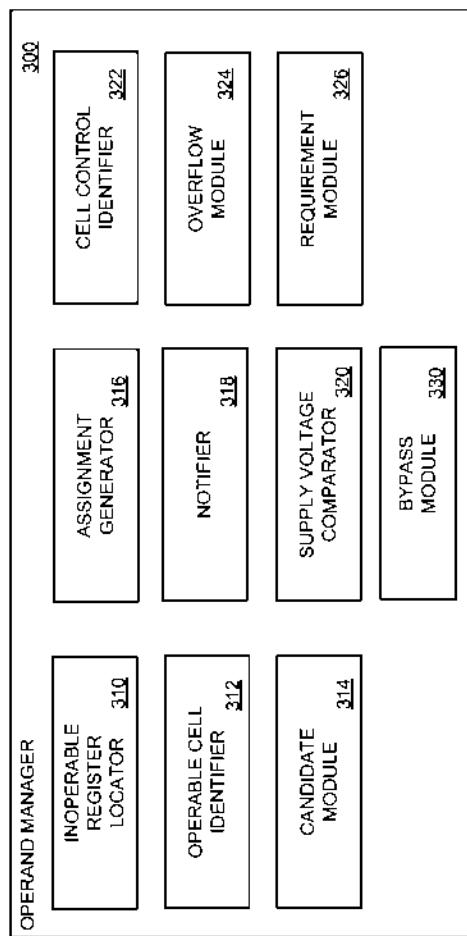


FIG. 3

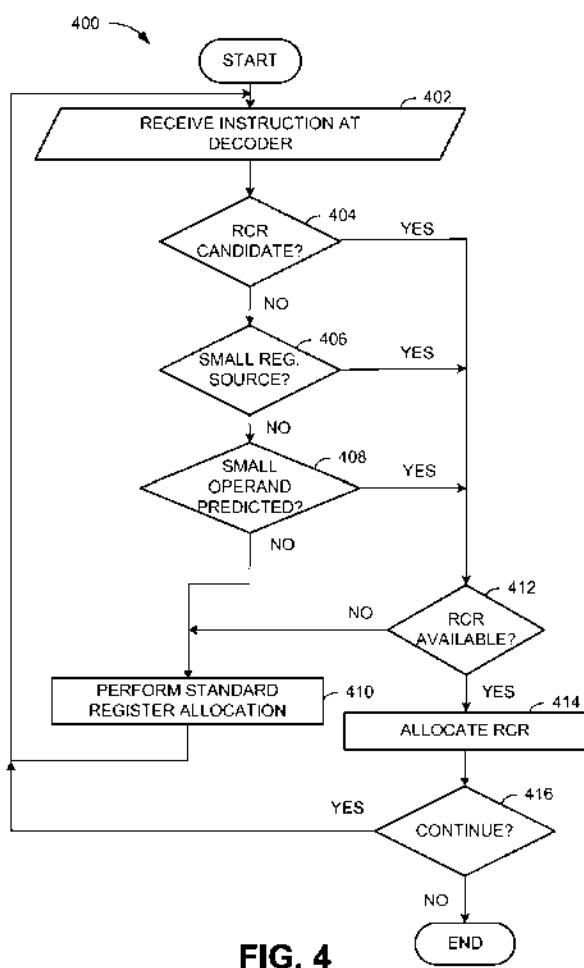


FIG. 4

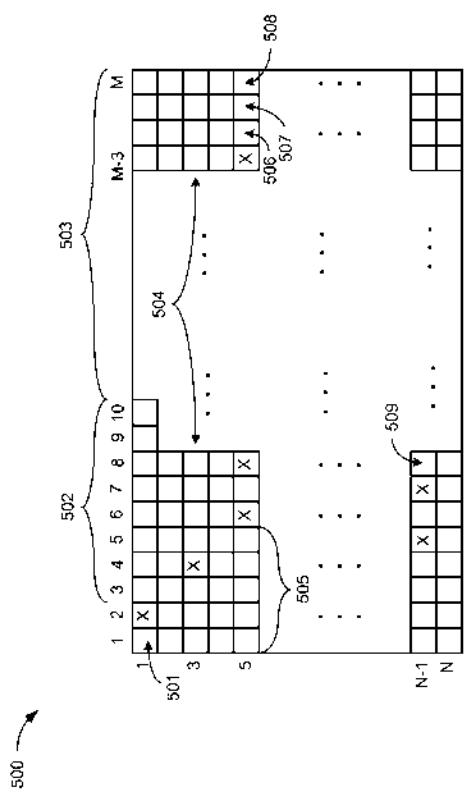
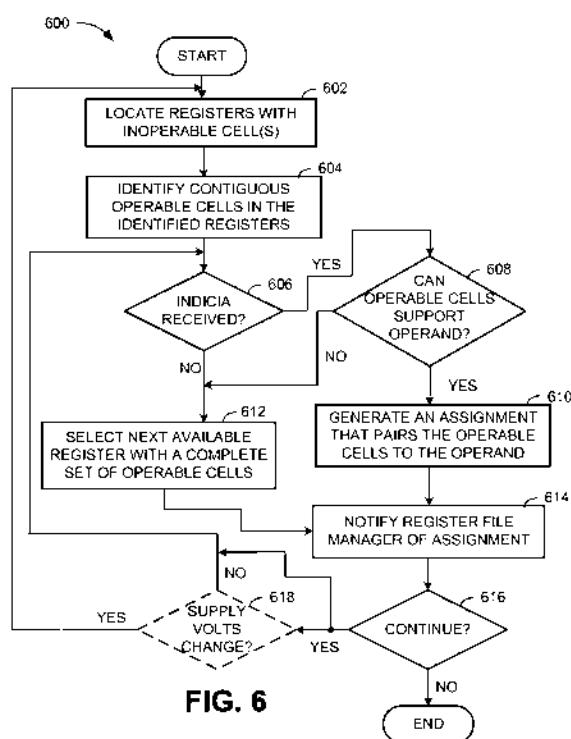


FIG. 5



**SYSTEM AND METHOD FOR ADAPTIVELY
MANAGING REGISTERS IN AN
INSTRUCTION PROCESSOR**

[0001] CLINICAL FIELD

[0001] The present disclosure relates generally to systems and methods for managing registers in instruction processing systems.

DESCRIPTION OF THE RELATED ART

[0002] As early as 1965, Gordon E. Moore observed that the number of circuit devices (e.g., transistor, resistor, capacitor) in a integrated circuit doubles approximately every two years. Moore attributed this to the log-linear relationship between circuit complexity and time. Moore's analysis was directed at the density of transistors at which cost is minimized.

[0003] In 1974, Robert H. Dennard observed that as transistors were made smaller through an photolithography, their power density remains constant, so power usage is proportional to integrated circuit area. According to Dennard, as transistor dimensions are reduced and with both voltage and current being proportional to length for transistors, performance per Watt would grow at roughly the same rate as transistor density. As transistor dimensions are reduced by 30%, input voltages are reduced to keep the electric field constant; power consumption is reduced by 50%. Therefore, with each generation of processor, the transistor density doubles, the circuit becomes 40% faster, while power consumption remains the same.

[0004] However, since about 2002, Dennard scaling no longer appears to be applicable as for smaller transistor sizes, current leakage and temperature become problematic. Both current leakage and heat contribute to a reduction in energy efficiency. In addition, operating voltages can no longer be reduced without increasing permanent and transient errors in integrated circuits.

[0005] The breakdown in Dennard scaling has led a number of integrated circuit designers to develop multi-core processors, the addition of more processors certainly increases chip performance. However, unless the power consumed per instruction is reduced, there will still be an increase in power density. In addition, multicore processors have proved to be difficult to program and have failed to reach their utilization potential.

SUMMARY

[0006] An embodiment of an instruction processing system that adaptively manages operable storage cells includes a set of available registers and an operand manager. The set of available registers includes N members of M cells, where N and M are positive integers, and where at least one of the M cells of an identified member register of the set of N registers is operable. The operand manager identifies a set of operable cells in the identified member register and determines if the set of operable cells in the identified member register can support an operand. In response to determining that the set of operable cells in the identified member register can support the operand, the operand manager generates an assignment that logically couples the set of operable cells to the operand and notifies a register file manager of the assignment. Otherwise, when the set of operable cells cannot support the oper-

and, the system selects the next available register with a complete set of operable cells.

[0007] An embodiment of a method for adaptively managing registers in a processor includes locating at least one member register from a set of available registers having at least one operable cell among the cells forming the at least one member register, identifying a set of operable cells in the at least one member register that has at least one operable cell, in response to an indicator that an operand can be supported by less than a nominal number of cells, determining if the set of operable cells in the at least one member register that has at least one operable cell can support the operand; and in response to determining that the set of operable cells can support the operand, generating an assignment that logically couples the set of operable cells to the operand, and notifying a register file manager of the assignment. Otherwise, when the set of operable cells cannot support the operand, selecting the next available register from the set of available registers having all operable cells.

[0008] These and other features and advantages of an improved instruction processing system and a method for adaptively managing registers in the instruction processing system will become apparent from the following description, drawings, and claims.

BRIEF DESCRIPTION OF THE DRAWINGS

[0009] In the figures, like reference numerals refer to like parts throughout the various views, unless otherwise indicated. For reference numerals with letter character designations such as "102a" or "102b", the letter character designations may differentiate two like parts or elements present in the same figure. Letter character designations for reference numerals may be omitted when it is intended that a reference numeral encompass all parts having the same reference numeral in all figures.

[0010] FIG. 1 is a schematic diagram illustrating an embodiment of an instruction processing system.

[0011] FIGS. 2A-2C each includes respective schematic diagrams illustrating the operability of individual cells in the set of available registers of the instruction processing system of FIG. 1.

[0012] FIG. 3 is a schematic diagram illustrating an embodiment of the operand manager of FIG. 1.

[0013] FIG. 4 is a flow chart illustrating an embodiment of a method for allocating operands to sets of operable cells in the operable registers of FIGS. 2A-2C.

[0014] FIG. 5 is a schematic diagram illustrating a collection of sets of operable cells available to support modified or short operands with the operand manager of FIG. 3.

[0015] FIG. 6 is a flow chart illustrating an embodiment of a method for adaptively managing registers in the instruction processing system of FIG. 1.

DEFINITION OF TERMS

[0016] The term "exemplary" is used herein to mean "serving as an example, instance, or illustration." Any aspect described herein as "exemplary" is not necessarily to be construed as preferred or advantageous over other aspects.

[0017] In this description, the phrase "instruction processing system" is used to describe a hardware element capable of accessing and executing machine level instructions. Non-limiting examples of instruction processing systems include a central processing unit ("CPU"), a digital signal processor,

(“DSP”), “graphical processing unit” (“GPU”), an application specific integrated circuit (“ASIC”), etc. Moreover, a CPU, DSP, GPU, ASIC, etc. may be comprised of one or more distinct processing components generally referred to as “core(s).”

[0018] In this description, “machine level instructions” are fetched and executed directly by a processing unit such as a CPU, GPU, etc. Each instruction performs a very specific task such as a load, a jump, or an arithmetic logic unit operation (e.g., add, subtract, multiply, increment, decrement, OR, AND, NOT, NOR, . . .) on a unit of data in register or in an addressable memory.

[0019] The term “contiguous cells” is used to refer to a series of adjacent bit storage elements in an addressable register.

[0020] The term “operable” is used to refer to data or an address.

[0021] The term “inoperable cell” is used to refer to a separate storage element in a physical register that is incapable of being controllably set to a condition or unable to maintain a condition indicative of a logical value. An inoperable cell may be energized or de-energized.

[0022] In contrast, an “operable cell” is a storage element that is energized, capable of being controllably set to a desired condition, and maintains the desired condition indicative of a logical value as long as the element remains energized.

[0023] The term “register file” is used to refer to an array of physical registers that contains the general-purpose registers of the instruction processing system.

[0024] The term “register file manager” is used to refer to remaining logic and/or additional interfaces that manipulate operands in the general-purpose registers of the instruction processing system.

[0025] In addition, “content” when associated with an operable cell, is used to identify a logical value (e.g., a two-state cell supports a logical value of “1” or “0”). The term “content” may also include groups of contiguous cells that form a digital word. As is known, digital words can be further combined to form instructions, and a series of instructions used to generate executables, such as: object code, scripts, byte code, markup language files, etc.

[0026] As used in this description, the terms “logic,” “map,” “element,” “imodule,” and the like are intended to refer to items enabled in hardware via circuits and combinations of circuits.

[0027] Such hardware implementations may include any or a combination of the following technologies, which are all well known in the art: discrete electronic components, discrete logic circuit(s) having logic gates for implementing logic functions upon data signals, an application specific integrated circuit having appropriate logic gates, a programmable gate array(s) (PGA), a field programmable gate array (FPGA), etc.

[0028] The improved instruction processing system may be integrated in an embedded system or controller that operates under the direction of firmware or may be integrated at a computing device in any of a number of various form factors. However arranged, the computing device may be enabled in operational modes using hardware, a combination of hardware and firmware, a combination of hardware and software in execution, or a combination of hardware, firmware and software in execution.

[0029] Consequently, an improved instruction processing system may be operated in conjunction with configuration

information and an executable, a thread of execution, a program, and/or multiple programs. These components may execute from various computer-readable media having various data structures stored thereon. The components may communicate by way of local and/or remote processes such as in accordance with a signal having one or more data packets (e.g., data frames) one component interacting with another component in a local system, distributed system, and/or across a network such as the Internet with other systems.

[0030] Improvements in the management of registers in an instruction processing system are illustrated and described. A register file (e.g., data stored in circuit(s) and register file manager (e.g., circuit(s) or hardware)) are fundamental components in instruction processing systems. An instruction set or instruction set architecture defines the basic steps related to programming. An instruction set includes or defines native data types, commands, registers, addressing modes for memory, interrupt and exception handling, and external input/output. A significant portion of the overall power consumed by an instruction processing system is related to the storage and management of values in registers. The size in number of bits of each register is defined by the architecture of the processing system. For example, an Intel i5 processor has a set of 32-bit integer registers, a second set of 80-bit floating point registers, and a third set of 128-bit vector registers. By way of further example, an ARM v7 compatible processor has a set 32-bit integer registers and a set of 64-bit floating point registers.

[0031] When an instruction is decoded, remaining logic in the register file manager pairs a logical output register from the instruction with a physical register. The remaining logic tracks the physical register until another instruction directs the instruction processing system to overwrite the contents in the physical register or the register is released back to a pool of available physical registers.

[0032] However, most stored values are significantly smaller than what can be stored in the assigned or paired register. This variability in the size in bits required to support stored values in registers of an instruction processing system provides an opportunity to reduce the power consumed by such a system.

[0033] A number of embodiments are possible and envisioned. In a first embodiment, a voltage regulator can be used to reduce the voltage applied to a set of available registers. This can be accomplished in a controlled manner until a desired voltage level is reached. As a result, some of the cells within a register or registers in the set of available registers will no longer store or hold an appropriate voltage. These particular cells are deemed inoperable and if used by an instruction processing system will lead to errors. Stated another way, one or more inoperable cells within a register render the collection of cells unfit for use as a conventional register with a full compliment of the designated number of bits. However, some portion of the cells will likely continue to operate nominally at this lower operating voltage. A map of the inoperable cell locations can be provided to an operable manager. The operable manager can use one or more external sources to detectly operate that can be supported by less than a full compliment of operable cells. In response, and in accordance with a desired or expected number of cells to support the operable, the operable manager pairs a set of operable cells selected from the map with the operable and notifies the remaining logic of the assignment.

[0034] In an alternative embodiment, the voltage regulator may be directed to lower the operating voltage provided to the set of available registers until a desired number of inoperable cells is detected. The supply voltage adjustment in this case would be made without additional information.

[0035] In other alternative embodiments, the voltage regulator may be controllably adjusted to provide an increase or a decrease in the operating voltage (e.g., a variable supply voltage) to the registers based on the number of contiguous operable cells of various lengths available to support modified or short operands. Additionally, such adjustments might also be considered in light of analysis of a particular application and the likelihood of a need for shortened operand support.

[0036] In still another alternative embodiment, a controller may be deployed to de-energize a selected number of cells in a desired number of registers to achieve a desired power savings. In this arrangement, the controller may provide configuration information or signals indicative of an array of cells that were de-energized (by the operand manager). Such an array can define a set of operable inoperable cells in a single physical register or a set of similarly positioned operable inoperable cells in two or more of the available physical registers.

[0037] Each of the described embodiments can achieve reductions in power consumption during various stages of a processor pipeline. For a processor pipeline that includes fetch, decode, rename, issue, execute, write-back and commit stages, the disclosed operand manager may be arranged with circuitry responsive to a number of sources that determine at the decode stage whether a present instruction will produce an operand with a small value. When the determination is affirmative, the operand manager may mark the instruction as a candidate for a modified or short register. At the rename stage, an instruction that has been marked as a candidate for a modified or short register may be paired with a set of contiguous operable cells from a member of the set of available registers. In this regard, the operand manager communicates the pairing or assignment to the register file manager. At write back, if the operand manager receives a signal indicative of an overflow condition, the operand manager releases the assigned or paired set of contiguous operable cells and reassigns a larger set of contiguous cells. Alternatively, the operand manager may signal the register file manager to use a full-size register from the set of available registers. At the commit stage, the physical register entry is dis-associated or released.

[0038] In an example embodiment, the operand manager maintains a list of available sets of contiguous operable cells by the number of contiguous cells. When an instruction identifies a candidate operand and P bits are desired, where P is an integer less than M, the operand manager uses the list to identify an available register with a set of operable cells that can be used to support the operand. When no registers are available that can support P bits, the operand manager may assign a register with a set of operable cells in a number greater than P, including a register with a full set of M operable cells.

[0039] In accordance with the improved instruction processing system, an operand manager is provided. The operand manager is coupled to the set of available registers having one or more inoperable cells. When so arranged, the operand manager includes a cell analyzer that detects inoperable cells and generates a map or report that identifies registers with inoperable cells and the location of the inoperable cells, the inoperable cells, or both operable and inoperable cells. Alter-

natively, the operand manager receives the map or identifying information from an external cell analyzer.

[0040] In an example embodiment, the operand manager responds to a decrease in the supply voltage that exceeds a threshold amount by directing the cell analyzer to recheck initial operation of the cells. The instruction processing system is suspended until the updated map or list of inoperable cells is available to the operand manager. Thus, the operand manager is responsive or sensitive to changes in the supply voltage provided to the physical registers in the instruction processing system.

[0041] The operand manager is also coupled to a decoder, a register file manager, and to an arithmetic logic unit. The operand manager uses instructions and other information (but these elements when determining how to appropriately pair a register with a set of operable cells and a set of inoperable cells. For example, the operand manager may use the present instruction in the current instruction register to analyze the immediate operands. An immediate operand that includes a value (e.g., an integer) that can be supported in less than M bits is a candidate to be supported by a physical register with less than a complete set of operable cells.

[0042] In addition, the operand manager may use the current value of a register to determine that the register is a candidate to be supported by a register with less than a complete set of operable cells. The operand manager can use information from the register file manager to make such a determination. For example, the current instruction register of the instruction processing system may include an instruction that directs the register file manager to move the content in a first register to a second register. When the content in the first register includes a value that can be supported in less than M bits, the second register is a candidate to be supported by a physical register with less than a complete set of operable cells.

[0043] Furthermore, the operand manager receives an indication of an overflow condition at an output register designated to support a write back execution stage. An overflow condition occurs when a calculation produces a result that is greater in magnitude than that which an identified register or storage location can store or represent. The operand manager responds to the overflow by signaling the register file manager that a remapping operation is in order. In some arrangements, the operand manager forwards the identity of a candidate physical register with the remapping signal. Alternatively, the register file manager upon receipt of the remapping signal may assign the next available register having a full set of operable cells to support the output from the arithmetic logic unit.

[0044] The improved instruction processing system can be used with architectures that support debugging features. For example, for architectures that provide an system programming support that enable a user to controllably map or assign an input value that was previously supported by a set of operable cells in a register with one or more inoperable cells, the operand manager will identify when such a mapping or assignment includes an input value that is too large to be supported by the set of operable cells. When this is the case, the operand manager will direct the register file manager to remap the input value to an appropriate register or will identify an appropriate set of operable cells from a different register to support the input value.

[0045] A modified compiler arranged to support an improved instruction processing system as described above

generates encoded instructions that indicate which the operand manager should apply or assign a physical register with less than a complete set of operable cells to support an operand resulting from the encoded instruction. Such encoded instructions may include an indication of a desired and/or a minimum number of cells to support the operand. The operand manager assigns a physical register to the operand by logically coupling the set of operable cells in the physical register to the operand and communicating a signal or signals to the register file manager. Otherwise, when the operable cells in the set of available registers cannot support the operand, the operand manager may select the next available register from the set of available registers having a first set of operable cells. Alternatively, the operand manager may simply rely on the renaming logic provided in the register file manager to assign the next available register.

[0046] As briefly described, many of the disclosed functions can be enabled in conjunction with register renaming logic (e.g., circuitry) to extend the flexibility and functionality of an instruction processor. While conventional register renaming techniques enable out-of-order execution by shifting use of directly named registers (i.e., registers defined in the assembly code) to additional physical registers to provide sequence control or parallelism, the present techniques when appropriately applied re-purpose operational bus in an otherwise incompatible register to enable the instruction processor to continue to provide accurate results under low input voltage conditions. Note that the present circuits and techniques can be applied to improved register renaming logic to support the identification, management and use of reduced size physical registers (registers other than logical architectural registers) in an instruction processing system.

[0047] Attention is now directed to the illustrated embodiment of the improved instruction processing system and its components as shown in FIGS. 1-6 [1c]. In a schematic diagram illustrating an exemplary embodiment of an instruction processing system 100, as illustrated, the various elements of the instruction processing system 100 are connected to one another via a bus 102 that conveys commands, data and power to several of the functional circuits or elements. Although several elements including the decoder 112, register file manager 114, operand manager 300, memory controller 130 and register cell map 200 are not shown with a direct connection to the voltage regulator 103 via the bus 102, distributing a regulated input voltage as provided by the voltage regulator 103 or a similar circuit element(s) is, in light of this disclosure, within the skill level of a person having ordinary skill in the art of microprocessor circuit design.

[0048] An instruction register 110 is arranged to identify instructions when they are present on the bus 102. The instruction register 110 is coupled to a decoder 112, which identifies the designated operation and is arranged to communicate the same to the arithmetic logic unit 108, the register file manager 114, the register file 120, and the operand manager 300. The arithmetic logic unit (ALU) 108 is a circuit or collection of circuits arranged to perform operations on the information provided in the accumulator and/or the temporary register 106 in accordance with information received from the decoder 112. The ALU 108 may include a control unit which generates the necessary signals to carry out or perform the instruction. The result of the designated operation is stored in the accumulator 104 and is available for later transfer to one of the designated registers in the register array

122 of the register file 120. The register file manager 114 includes circuits for directing and coordinating information stored in the register file 120. In this regard, the register file manager 114 includes logic for updating the stack pointer 124, the program counter 126, and a memory address increment/decrement register 128.

[0049] A memory controller 130 allows data transfers to and from storage coupled to the memory bus 110, which may be coupled to one or more random-access memory circuits (volatile memory), and one or more flash memory circuits or elements (non-volatile) both not shown in FIG. 1.

[0050] The cell controller 130 is an optional circuit or circuits that controllably de-energize or otherwise render a desired number of configurable cells of select registers within the register array 122 inoperable. The cell controller 130 may function in conjunction with one or more sensors from a device controller or other device or mechanism external to the instruction processing system. In this regard, the cell controller 130 provides a mechanism to controllably reduce the power consumed in processing instructions with the various registers provided in the register array 122.

[0051] A register cell analyzer 140 includes circuits arranged to identify whether individual cells or bit locations within the registers of the register array 122 are operable and/or inoperable under present conditions. As described, the register cell analyzer 140 may respond a signal or other indicia responsive to the supply voltage crossing a threshold value. When this is the case, the signal or other indicia may be forwarded to the decoder 112, the ALU 108, and the register file manager 114 to controllably suspend the processing of instructions. When enabled, the register cell analyzer 140 identifies the operable and/or inoperable storage cells for each of the registers in the register array and provides or stores the information in the register cell map 200, which is described in example embodiments in conjunction with FIGS. 2A-2C.

[0052] Although the example embodiments do not illustrate or expressly describe the use of the cell controller 130 in conjunction with the register cell analyzer 140, register cell map 200 and operand manager 300, such combinations are, at light of this disclosure, within the skill level of a person having ordinary skill in the art of microprocessor circuit design.

[0053] As further illustrated in FIG. 1, the operand manager 300 receives the information in the register cell map 200 and uses the same in conjunction with the register file manager 114 and the register file 120 to expose and use reduced cell registers that use one or more of the separate operable cells identified from within an inoperable register.

[0054] FIGS. 2A-2C each includes respective schematic diagrams illustrating the operability of individual cells in the set of available registers of the instruction processing system 100 of FIG. 1. For example, FIG. 2A includes a representation of map 200a of N_xN_y registers of M bits, where both N and M are positive integers. The map 200a is a record of the bits or storage locations that are both operable (unmarked) and inoperable (marked) at the present input voltage provided by the voltage regulator 103 [FIG. 1]. A first register represented by the top or uppermost row of N registers (i.e., row 1, as labeled on the left-hand side) includes an inoperable cell at the second cell location from the left, indicated by the enunciative label "2"; while each of the remaining M-1 cell locations of the register are operable. Thus, the first register is capable of supporting a flag or bit in the first cell location from the left or

a reduced cell register with M-2 or less bits in the operable cells to the right of the inoperable cell location [0055]. In one embodiment, a single reassignment of the first register from an M-bit register to other than an M-bit register is enabled. In this embodiment, if the bit locations 3 through 20 are reassigned to support the storage of an operand that requires a byte to represent the operand, the remaining operable storage locations in locations 1 and 11 through M are not reallocated.

[0056] In an alternative embodiment, the M-2 bits of operable cells to the right of the inoperable cell location may be assigned or allocated as a reduced cell register ranging from a single bit up to M-2 bits. When allocated as a single bit, M-2 single bits or flags can be supported, as needed or desired. When allocated as multiple bit reduced cell registers, a combination of reduced cell registers and bit lengths are possible. For example, when M is the integer 32 and an inoperable bit is located in the second bit position, a total of 30 bits may be allocated to a single register or subdivided as may be desired. In some arrangements, the subdivisions may be determined based on factors of the integer 2. In other arrangements, the subdivisions may be determined based on preexisting requirements for bit lengths or by use of other mechanisms.

[0057] As further shown in FIG. 2A, a third register represented by the third row from the top (i.e., row 3, as labeled on the left-hand side) includes an inoperable cell in the fourth cell location from the left, while each of the remaining M-1 cell locations are operable. Thus, the third register is capable of supporting three single bit storage locations, a single three-bit storage register, or a combination of a single bit storage location and a two-bit storage register in the cells to the left of the inoperable cell. In addition, the third row is further capable of supporting an M-4 reduced cell register in the inoperable bits to the right of the inoperable cell. As indicated above, the M-4 inoperable cell locations can be subdivided and allocated as desired.

[0058] A fifth row from the top (i.e., row 5, as labeled on the left-hand side) includes inoperable cells in the 6th, 8th, and M-3 bit positions from the left. Accordingly, the cells or bits defined by column labels 1-5, 7-9 through M-4, and the remaining storage locations are operable and available for assignment as desired.

[0059] A second to last row (i.e., row M-1, as labeled on the left-hand side) includes inoperable cells in the 5th and 7th bit positions from the left. Thus, the cells or bits identified by column labels 1-4, 6, and 8 through M are available for assignment as desired.

[0060] As shown in FIG. 2B, several additional bits or storage locations are inoperable when compared to the map 200 of FIG. 2A. The map 200b is a record of the same sets of registers shown in FIG. 2A operating under a different condition or condition. For example, the map 200b represents bits or storage locations that are both operative (unmarked) and inoperative (marked) at a lower input voltage than the input voltage that resulted in the map 200a in FIG. 2A. As described in association with the map 200a in FIG. 2A, the operative cells in the map 200b are available for assignment to support one or more flags or single bit storage elements to locate or more reduced cell registers having any number of contiguous operative cells as may be desired.

[0061] FIG. 2C includes a map 201, that represents bits or storage locations that are both operative (unmarked) and inoperative (marked) as a result of a signal or signals communicated from the cell controller 130. As described, the cell

controller 130 actively de-energizes a subset of the cells of one or more of the N registers to provide one or more reduced cell registers to one or more of the decoder 112, the register file manager 114 and/or the operand manager 300. In the illustrated embodiment, the cell controller 130 has de-energized a byte corresponding to the left-most storage locations of the N registers. It should be understood that the cell controller 130 can be arranged to mark more or less bits than a byte for any one of the N registers, many desired combinations [0062]. FIG. 3 is a schematic diagram illustrating an embodiment of the operand manager 300 of FIG. 1. As illustrated in FIG. 3, the operand manager 300 includes various circuits, sub-assemblies or modules that are arranged to perform one or more tasks in the improved instruction processing system. In the illustrated embodiment, the operand manager 300 includes an inoperable register locator 310, an operable cell identifier 312, a candidate module 314, an assignment generator 316, a notified 318, a supply voltage comparator 320, a cell control identifier 322, an overflow module 324, a requirement module 326 and a bypass module 330. The inoperable register locator 310 is a circuit or assembly of circuits arranged to locate or otherwise identify at least one member from a set of available registers having at least one inoperable cell among the cells forming the at least one member register. As indicated, an inoperable cell is a cell that cannot be directed to a desired condition (e.g., to store or hold a voltage level corresponding to a defined logical value) or is unable to hold or store the desired condition under present conditions [0063]. The inoperable register locator 310 may be controllably activated in response to an indication that the regulated (or unregulated) supply voltage has fallen below a desired input voltage. Alternatively, the inoperable register locator 310 may be controllably activated in response to a signal from the supply voltage comparator 320 indicating that the regulated supply voltage supplying the registers and in other circuitry in the instruction processing system 100 has changed by more than a threshold amount. In another alternative, the inoperable register locator 310 may be controllably activated in response to a controller that de-energized at least one cell in the set of available registers. When such a controller de-energizes a block or blocks of cells, the inoperable register locator 310 may be directed to only check the operability of the cells that were not controllably de-energized. However, as applied, results provided by the inoperable register locator 310 are used to populate the register cell map 200. The operable cell identifier 312 is a circuit or assembly of circuits arranged to confirm or otherwise identify a subset of cells in a M-bit register that can be directed to a desired condition.

[0064] The candidate module 314 includes circuits arranged to identify valid candidates. The operand manager uses instructions and other information from these elements when determining how to appropriately pair a register with a set of operable cells and a set of inoperable cells. For example, the operand manager may use the present instruction in the current instruction register to analyze the immediate operands. An immediate operand that includes a value (e.g., an integer that can be supported in less than M bits) is a candidate to be supported by a physical register with less than a complete set of operable cells.

[0065] In addition, the operand manager may use the current value of a register to determine that the register is a candidate to be supported by a register with less than a complete set of operable cells. The operand manager can use information from the register file manager to make such a

determination. For example, the current instruction register of the instruction processing system may include an instruction that directs the register file manager to move the content in a first register to a second register. When the content in the first register includes a value that can be supported in less than M bits, the second register is a candidate to be supported by a physical register with less than a complete set of operable cells.

[0066] The assignment generator 316 is a circuit or collection of circuits under the control of the operand manager 300 that is to locate a select storage cell or set of contiguous operable storage cells to use as an assigned storage location for an operand. The assignment remains active until the operand stored therein is cleared or power is removed from the instruction processing system 100.

[0067] The notifier 318 is a circuit or collection of circuits under the control of the operand manager 300 that signal the register file manager 114 and/or the register file 120 of the assignment generated by the assignment generator 316.

[0068] The cell control identifier 322 identifies when a controller such as the optimal cell controller 130 has signaled to whom it is desired that some number of otherwise operable cells should be disabled, or de-energized from the otherwise available number of cells in a desired number of registers to achieve a desired power savings. In this arrangement, the cell control identifier 322 may provide configuration information or signals indicative of an array of cells that were de-energized by the operand manager. Such an array can define a set of operable/inoperable cells in a physical register or a set of similarly positioned operable/inoperable cells in two or more of the available physical registers.

[0069] The overflow module 324 is a circuit or collection of circuits that in response to an overflow condition received from the ALU 108 or other sources in the instruction processing system 100 generate appropriate signals to avoid or correct the overflow condition by reassigning the storage resources that are available to support the ALU 108. As a part of the reassigning, the operand manager 300 may notify or otherwise communicate that a remapping of the register array 122 is required to the register file manager 114.

[0070] The requirement module 326 is a circuit or collection of circuits arranged to identify the number of cells or bits will be required to adequately support a likely result from the present instruction being processed by the instruction processing system 100.

[0071] The bypass module 330, in response to an input signal, de-energizes the operand manager 300 and the various sub-assemblies and modules thereof. When the operand manager 300 is bypassed or circumvented, the instruction processing system 100 assigns registers in a conventional manner.

[0072] The operand manager 300 may use several distinct mechanisms to identify potential candidate operands that may be supported by a reduced cell register. For example, the operand manager 300 may respond via a stored lookback of the accumulator 104 and the temporary register 106 to make such a determination. Alternatively, and/or additionally, the operand manager 300 may use an output of the ALU 108 to determine if the operand is a suitable candidate for a reduced cell register.

[0073] In some embodiments, the instruction register 110 and the decoder 112 may be arranged to support a compiler by recognizing and forward an indication that an identified instruction can be supported with a reduced cell register.

These circuit elements and the compiler they support may further indicate a desired number of cells required for the modified operand. When this is the case, the decoder 112 may forward an indication via a signal that the present instruction is a candidate for support with a register having less than M operable cells. In some arrangements, the decoder 112 may be arranged to communicate the desired minimum number of cells to support an operand for the present instruction.

[0074] FIG. 4 is a flow chart illustrating an embodiment of a method 400 for allocating operable locations of operable cells in the available registers of FIGS. 2A-2C that may be implemented by the reduced cell register coordinator 160 illustrated by way of broken line in the instruction processing system 100 of FIG. 1. The method 400 begins with input block 402 where an instruction is received at a decoder (e.g., the decoder 112 of FIG. 1). A series of respective queries are performed as indicated in decision block 404, decision block 406 and decision block 408. In decision block 404, the decoder 112 determines if the present instruction identifies or includes a small immediate operand. A small immediate operand is an operand that requires less than M bit cell locations to represent the present value associated with the operand. When the response to the query in decision block 404 is negative, the decoder 112 continues by determining whether the instruction identifies a register source that is using (i.e., a present value) less than M bit cell locations. When the response to the query in decision block 406 is negative, the decoder 112 continues by determining whether a small operand is predicted (e.g., a value predictor may receive one or more signals or other indicators to predict that a result of the instruction will likely fit within a set of bit locations that is less than M bit locations). When the response to the query in decision block 408 is negative, the decoder 112 continues by directing the register file manager or register renaming logic enabled in the decoder 112 or one or more of the operand manager 300 or the register file manager 114 to allocate a set of available fully operational register to support the operand, as indicated in block 410. Upon completion of the allocation in block 410, the decoder 112 may repeat the functions illustrated in the blocks 402 through 408.

[0075] As indicated in FIG. 4, an affirmative response to any of the separate queries in decision block 404, decision block 406, or decision block 408 is followed by an additional query, as indicated in decision block 412, to determine if a reduced cell register is available. When a reduced cell register is available, one or more of the operand manager 300 or the register file manager 114 may be configured to allocate an available and suitably sized reduced cell register for supporting the operand, as shown in block 414. Thereafter, a determination is made whether to continue or to terminate the method 400. When it is desired to continue, the method 400 repeats the functions described in association with blocks 402 through 414, as described. Otherwise, the method 400 terminates and the decoder 112 no longer identifies instructions with a short operand that can be supported by a register with less than a full set of operable bit cell locations.

[0076] FIG. 5 is a schematic diagram illustrating a collection of sets of operable cells available to support modified or short operands with the operand manager of FIG. 3. As illustrated in FIG. 5, the representation or map 200a of N registers of M bits, as introduced in FIG. 2A with inoperable cells scattered throughout the N registers provides opportunities in registers identified by labels 1, 3, 5 and N-1 for assignments of one or more reduced cell registers or storage locations. In

the illustrated embodiment, operative cells in rows of M bits including at least one inoperative bit are assigned to reduced cell registers.

[0077] For example, in the M bits in the first row from the top, in the revised map 500, a one-bit flag 501 is assigned to left-most storage cell of the first row, an eight bit register 502 or byte is assigned to bits 3 through 10, and bit locations 11 through M are assigned to reduced cell register 503. By way of further example, as shown in row 3, operative bits in cells identified by column labels 5 through M are assigned to reduced cell register 504. Although cells in the first three cell locations are operative they are not assigned in the illustrated embodiment.

[0078] In row 5, operative cells in the first five cell locations are assigned to reduced cell register 505 and each of the last three individual cell locations are assigned to flag 506, flag 507 and flag 508. Although the cell in the seventh position from the left in row 5 is operative the cell is not assigned in the illustrated embodiment.

[0079] By way of additional example, in the row identified by the label N-1, the operative cell in bit location S is assigned to a flag 509. Although the cells in the first 4 bit locations, the sixth position, and in bit positions along through M in row N-1 are operative the cells in these locations are not assigned either to flags or to reduced cell registers in the illustrated embodiment.

[0080] FIG. 6 is a flow chart illustrating an embodiment of a method 600 for adaptively managing registers in the instruction processing system 100 of FIG. 1. The method 600 may be implemented by the register file manager 300 in conjunction with the register file manager 114 and the register file 120. Alternatively, the method can be extended to apply the above-described changes to retaining logic to identify and manage reduced cell registers in other storage elements coupled to the instruction processing system 100.

[0081] The method 600 begins with block 612 where registers with inoperable cells are located or otherwise identified. As described, this can be accomplished by various elements arranged to controllably write logical values to and read logical values from the separate locations of an N-bit register. In block 604, for registers with inoperative bit locations, contiguous operable bit locations are identified and recorded. In decision block 606, it is determined when indicia is received that an operand with less than the nominal number of bit locations in a register will be needed. As indicated by the flow control arrow labeled "No," exiting decision block 606, when no indicia of a short operand is present, the improved instruction processor 100 selects the next available register with a complete set of operable cells, as indicated in 612. Thereafter, the instruction processor 100 associates or assigns the operand to the identified register and notifies the register file manager of the assignment, as indicated in block 614.

[0082] Otherwise, as indicated by the flow control arrow labeled "Yes," exiting decision block 606, when indicia of a short operand is present, it is determined in decision block 608 if operable cells are available to support the short operand. For example, if the indicia indicate that a short operand will require at least P bits to support the short operand, where P is a positive integer less than M, the query in decision block 608 will determine if an otherwise inoperable register includes a contiguous set of at least P bit locations. When an inoperable register includes a contiguous set of at least P bit locations, as indicated in block 610, the instruction processor 100 generates an assignment that pairs the operand to the

operable cells or bit locations before notifying the register file manager of the assignment or pairing, as indicated in block 614.

[0083] Otherwise, when no sets of operable cells of an identified inoperable complete or full register will support the short operand, the method 600 continues with block 612 where the next available register with a complete set of operable cells is assigned or paired with the operand before notifying the register file manager of the assignment, as shown in block 614. Whether a modified or short operand was supported by a reduced capacity register or a fully operational register, as indicated in block 616, a determination is made whether to continue or terminate the method 600. When it is desired to continue, the method 600 repeats the functions described in association with blocks 606 through 616, as described. Optionally, as indicated in decision block 618, when a change to the supply voltage has occurred, the method 600 will repeat the functions illustrated in block 602 and block 614 before looking for indicia of a short operand. Otherwise, when the supply voltage has not changed and/or exceeded a threshold change, processing continues with the functions described in conjunction with blocks 606 through 616.

[0084] Certain steps in the processes or process flows described in this specification naturally precede others for the improved instruction processing system to function as described. However, the instruction processing system is not limited to the order of the steps described if such order or sequence does not alter the described functionality. That is, it should be recognized that some steps may be performed before, after, or parallel (substantially simultaneous) with other steps without departing from the disclosed methods. In some instances, certain steps may be omitted or not performed without departing from the method as understood by one of ordinary skill in the art. Further, words such as "thereafter," "then," "next," etc., are not intended to limit the order of the steps. These words are simply used to guide the reader through the description of the exemplary method.

[0085] In view of the disclosure above, one of ordinary skill in programming is able to write computer code or identify appropriate hardware and/or circuits to implement the disclosed systems and methods without difficulty based on the schematic drawings, flow charts and associated description in this specification. Therefore, disclosure of a particular set of program code instructions or detailed hardware devices is not considered necessary for an adequate understanding of how to implement and use the improved instruction processing system.

[0086] As described, one or more exemplary aspects, the improved instruction system may be implemented in hardware, software, firmware, or any combination thereof. If implemented in software, the functions may be stored or transmitted in one or more instructions or code on a computer-readable medium. Computer-readable media include both computer storage media and communication media, including any medium that facilitates transfer of a computer program from one place to another. A non-transitory storage media may be any available media that may be accessed by the instruction processing system. By way of example, and not limitation, such computer-readable media may comprise RAM, ROM, EPROM, EEPROM or other optical disk storage, magnetic disk storage or other magnetic storage devices, or any other medium that may be used to carry or store desired

program code in the form of instructions or data structures and that may be accessed by a computer.

[0087] Disk and disc, as used herein, includes compact disc ("CD"), laser disc, optical disc, digital versatile disc ("DVD"), floppy disk and Blu-ray disc where disks usually reproduce data magnetically, while discs reproduce data optically with lasers. Combinations of the above should also be included within the scope of computer-readable media [0088]. Therefore, although selected aspects have been illustrated and described in detail, it will be understood that various substitutions and alterations may be made therein without departing from the scope of the present systems and methods, as defined by the following claims.

What is claimed is:

1. A method for adaptively managing registers in a processor, the method comprising:
locating at least one member register from a set of available registers having at least one inoperable cell using the cells forming the at least one member register;
identifying a set of operable cells in the at least one member register that has at least one inoperable cell;
in response to an indication that an operand can be supported by less than a nominal number of cells, determining if the set of inoperable cells in the at least one member register that has at least one inoperable cell can support the operand; and
in response to determining that the set of operable cells in the at least one member register that has at least one inoperable cell can support the operand generating an assignment that logically couples the set of operable cells to the operand; and
notifying a register file manager of the assignment.
2. The method of claim 1, wherein the locating at least one member register from the set of available registers having at least one inoperable cell is responsive to a first input voltage.
3. The method of claim 2, wherein the locating at least one member register from the set of available registers having at least one inoperable cell is responsive to a second input voltage lower than the first input voltage.
4. The method of claim 1, wherein the locating at least one member register from the set of available registers having at least one inoperable cell is responsive to a controller that de-energized at least one cell in the set of available registers.
5. The method of claim 1, wherein identifying the set of operable cells in the at least one member register is responsive to a controller that de-energized at least one cell in the set of available registers.
6. The method of claim 1, wherein the determining if the set of operable cells can support the operand is responsive to a decoder arranged to analyze a present instruction.
7. The method of claim 1, wherein the determining if the set of operable cells can support the operand is responsive to a stored value.
8. The method of claim 1, wherein the determining if the set of inoperable cells can support the operand is further responsive to an output of an arithmetic logic unit.
9. The method of claim 8, wherein when the output of the arithmetic logic unit is an indication of an overflow condition, an operand manager responds by notifying a register file manager that a remapping is required.
10. The method of claim 1, wherein the determining if the set of operable cells can support the operand is responsive to a compiler arranged to forward an indication that an identified instruction can be supported with a modified operand.
11. The method of claim 10, wherein in the compiler indicates a required number of cells to support the modified operand.
12. An instruction processing system for adaptively managing a set of available registers, the system comprising:
a set of available registers including N members of M cells, where N and M are positive integers, and wherein at least one of the M cells of an identified member register of the set of N registers is inoperable;
an operand manager coupled to the set of available registers and arranged to:
identify a set of operable cells in the identified member register;
determine if the set of operable cells in the identified member register can support an operand; and
in response to determining that the set of operable cells in the identified member register can support the operand,
generate an assignment that logically couples the set of operable cells to the operand and notify a register file manager of the assignment.
13. The system of claim 12, wherein the set of available registers is provided a variable supply voltage.
14. The system of claim 13, wherein the operand manager, in response to a change in the supply voltage, identifies changes in the set of operable cells.
15. The system of claim 12, wherein the operand manager is responsive to an array of controllably de-energized cells.
16. The system of claim 12, wherein the operand manager is responsive to a decoder arranged to analyze a present instruction and forward an indication that the present instruction is a candidate to support with a register having less than M operable cells.
17. The system of claim 17, wherein the decoder further communicates a desired number of cells to support an operand and identified in the present instruction.
18. The system of claim 12, wherein the operand manager is responsive to an indication of an overflow condition and wherein the response includes signaling a register file manager that a remapping is required.
19. The system of claim 12, wherein the operand manager is responsive to a compiler modified instruction indicating that the instruction can be supported with a modified operand.
20. The system of claim 19, wherein the compiler indicates a desired number of operable cells.

* * *



US 20160103612A1

(19) United States

(21) Patent Application Publication

Christodorescu et al.

(10) Pub. No.: US 2016/0103612 A1

(43) Pub. Date:

Apr. 14, 2016

(54) APPROXIMATION OF EXECUTION EVENTS
USING MEMORY HIERARCHY
MONITORING

(71) Applicant: QUALCOMM Incorporated, San
Diego, CA (US)

(72) Inventors: Mihai Christodorescu, San Jose, CA
(US); Mostovitch Salajegheh, San Jose,
CA (US); Dario Suarez Gracia, Santa
Clara, CA (US)

(21) Appl. No.: 14/8512,434

(22) Filed: Oct. 12, 2014

Publication Classification

(51) Int. Cl.
G06F 2096
G06F 21/56

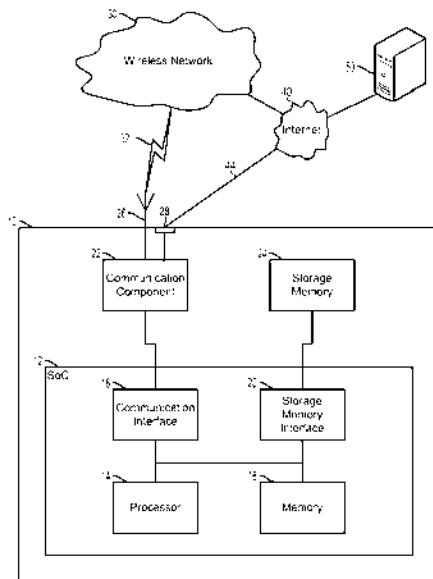
20060111

20060111

(52) U.S. Cl.
CPC G06F 3/06601 (2013.01), G06F 21/56
(2013.01); G06F 3/0653 (2013.01), G06F
20673 (2013.01); G06F 2221/034 (2013.01)

(57) ABSTRACT

Aspects include computing devices, systems, and methods for implementing monitoring communications between components and a memory hierarchy of a computing device. The computing device may determine at least one identifying factor for identifying execution of the processor-executable code. A communication between the components and the memory hierarchy of the computing device may be reinforced for at least one communication factor of a same type as the at least one identifying factor. A determination whether a value of the at least one identifying factor matches a value of the at least one communication factor may be made. The computing device may determine that the processor-executable code is executed in response to determining that the value of the at least one identifying factor matches the value of the at least one communication factor.



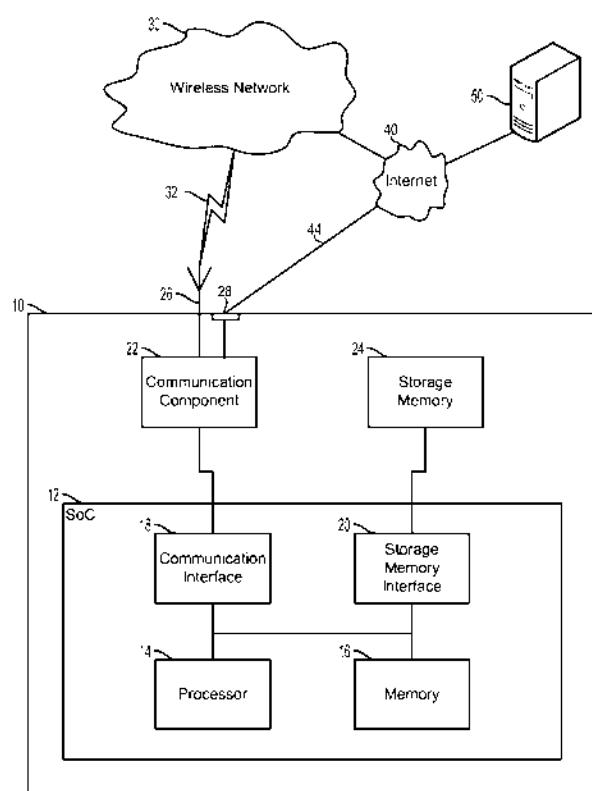


FIG. 1

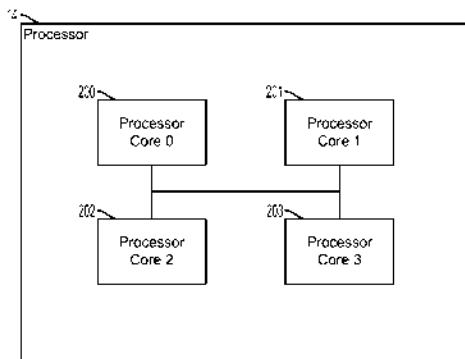


FIG. 2

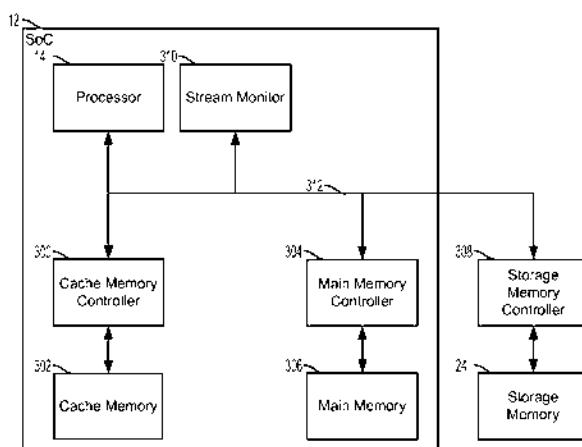


FIG. 3

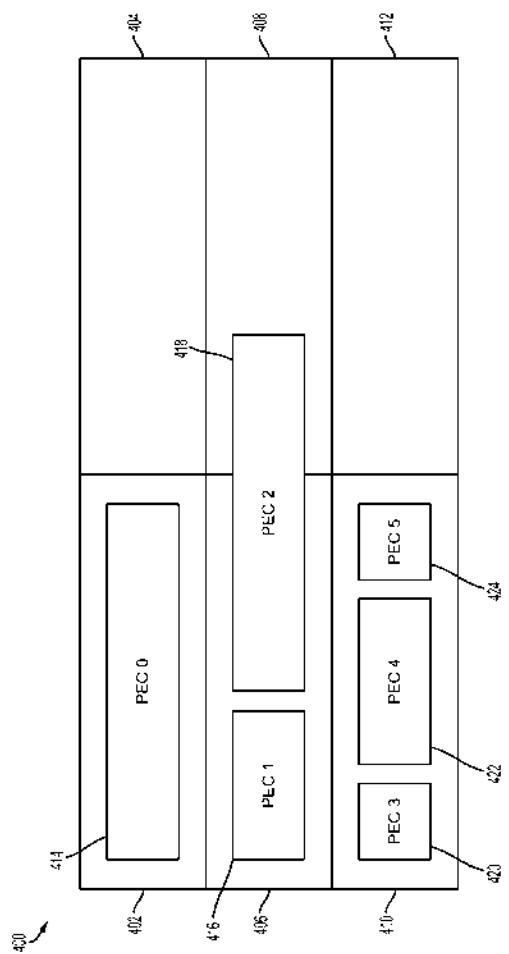


FIG. 4

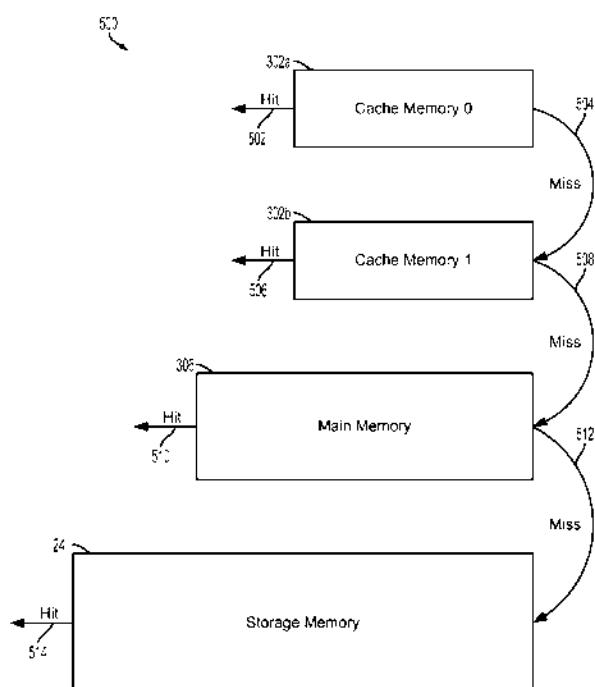


FIG. 5

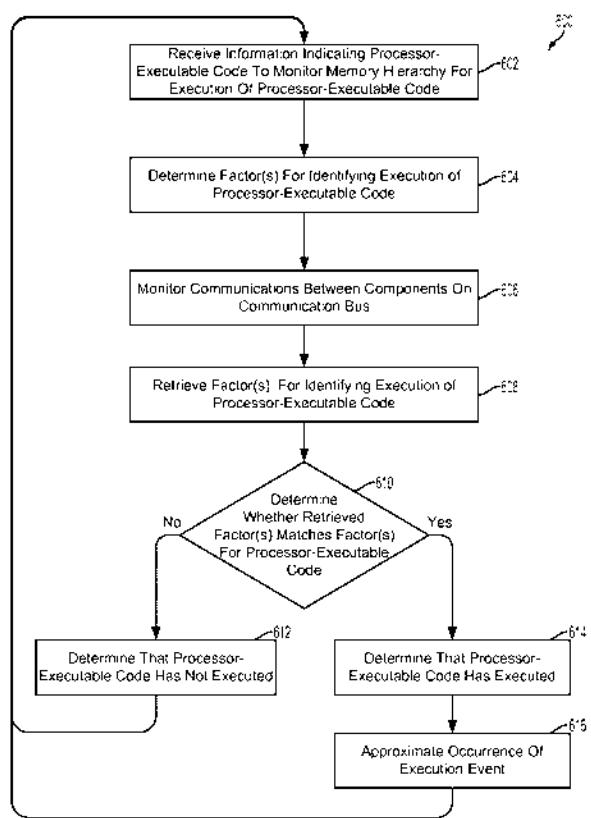


FIG. 6

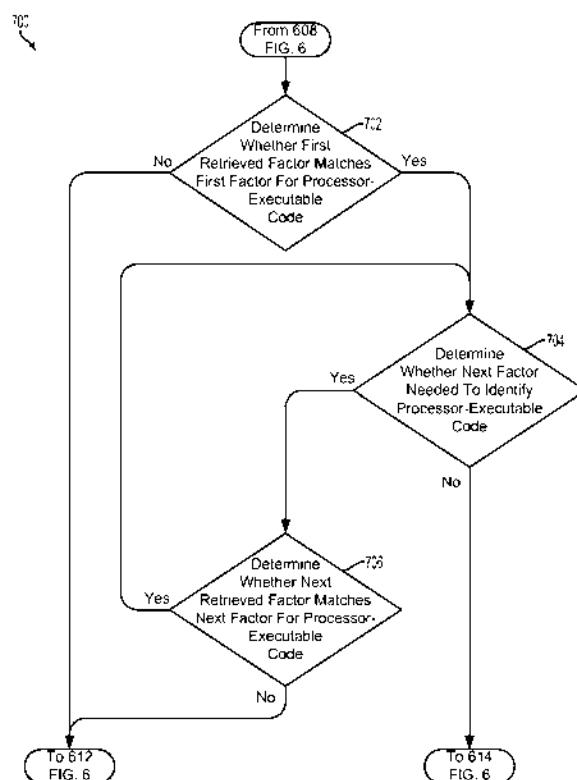
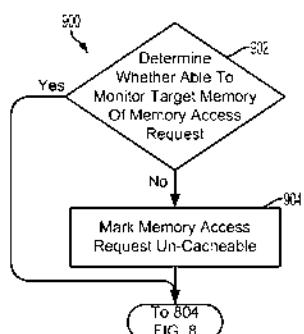
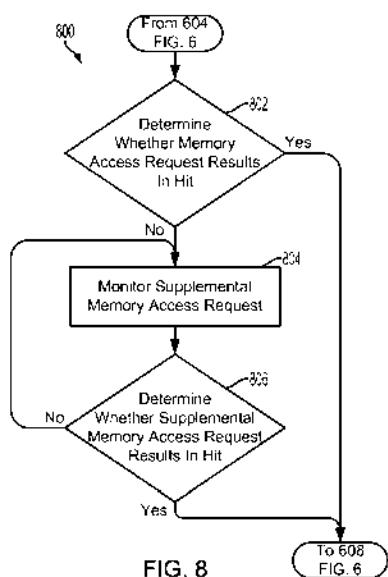


FIG. 7



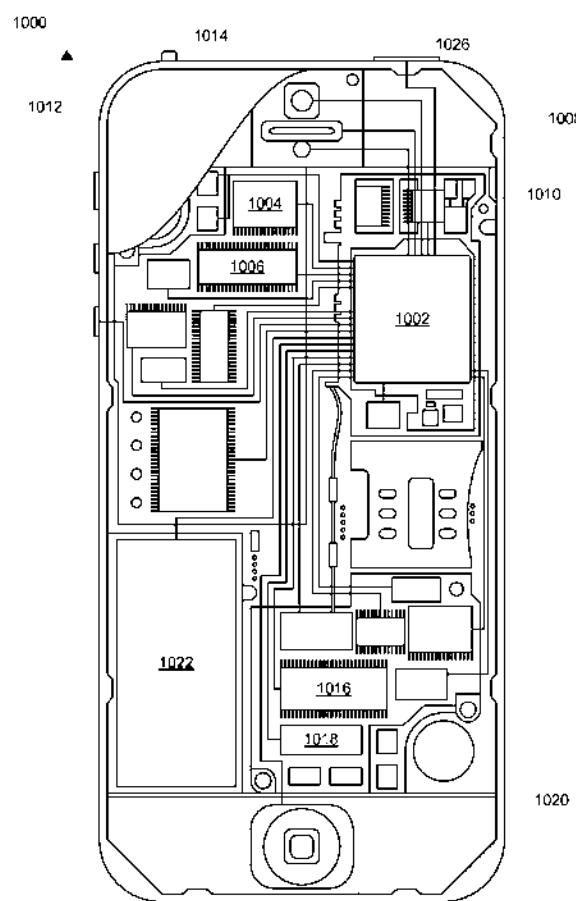


FIG. 10

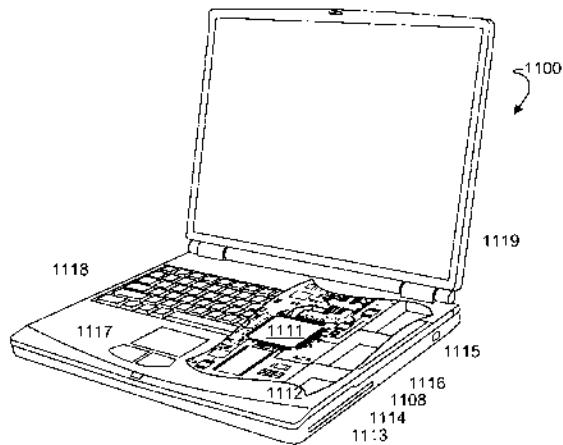


FIG. 11

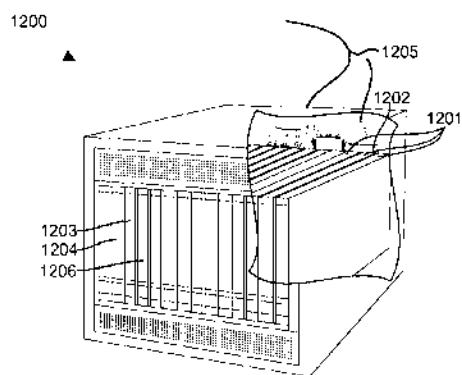


FIG. 12

APPROXIMATION OF EXECUTION EVENTS USING MEMORY HIERARCHY MONITORING

BRIEF DESCRIPTION

[0001] Monitoring execution events at the hardware layer and in real-time allows for monitoring of application programming interface (API) calls. Monitoring API calls is useful for malware detection, malfunction detection, protecting software with hardware, and tying monitoring to hardware. The API calls may be monitored in multiple instances and patterns that may indicate that a computing device is not operating as intended. One way to monitor execution events is by monitoring central processor unit (CPU) instruction streams. The instructions executed by the CPU may occur at instances and patterns that are identified as problematic for the computing device. However, monitoring all CPU instructions to find an execution of a specific address is both complicated and inefficient. Moreover, not all computing device systems support CPU monitoring. To monitor the CPU instructions at the high frequency at which CPUs execute instructions requires additional high-speed hardware added to the CPU and capable of monitoring the execution in the CPU at the same frequency.

SUMMARY

[0002] The methods and apparatuses of various aspects provide circuits and methods for monitoring communications between components and a memory hierarchy of a computing device that may include determining an identifying factor for identifying execution of a processor-executable code, monitoring a communication factor at a communication between the components and the memory hierarchy of the computing device or the identifying factor, determining whether a value of the identifying factor matches a value of the communication factor, and determining that the processor-executable code is executed in response to determining that the value of the identifying factor matches the value of the communication factor. In an aspect, determining whether a value of the identifying factor matches a value of the communication factor may include determining whether a value of a first identifying factor matches a value of a second communication factor in response to determining that the second identifying factor is needed to identify execution of the processor-executable code. In an aspect, a type of the identifying factor and the communication factor may include one of: an entry point address of a target memory; an exit point address of a target memory; a callee function, a caller function, a parameter, a unique instruction, a unique pattern, a cache footprint, a local variable, and a return value.

[0003] An aspect method may further include determining whether the communication matches another identifying factor is used to identify execution of the processor-executable code in response to determining that the value of the second identifying factor matches the value of the second communication factor. In an aspect, a type of the first identifying factor and the first communication factor is different from a type of the second identifying factor and the second communication factor. In an aspect, determining whether a second identifying

factor is used to identify execution of the processor-executable code may include determining whether the second identifying factor is used to identify execution of the processor-executable code in response to determining that the value of the first identifying factor matches the value of the first communication factor, the value of the first communication factor matches the value of the first communication factor or an overhead for monitoring the first communication factor exceeds a threshold.

[0004] An aspect method may further include determining that the processor-executable code is not executed in response to determining that the value of the identifying factor does not match the value of the communication factor.

[0005] In an aspect, monitoring for a communication factor in a communication between the components and the memory hierarchy of the computing device of a same type as the identifying factor may include determining whether a memory access request to a first target memory of the memory hierarchy results in a miss, and monitoring a supplemental memory access request to a second target memory of a lower level of the memory hierarchy in response to determining that the memory access request results in a miss.

[0006] In an aspect, the communication may be associated with a target memory of the memory hierarchy, and the method further include determining whether the communication can be monitored and marking the communication un-monitored in response to determining that the communication cannot be monitored.

BRIEF DESCRIPTION OF THE DRAWINGS

[0007] The accompanying drawings, which are incorporated herein and constitute part of this specification, illustrate example aspects of the invention, and together with the general description given above and the detailed description given below, serve to explain the features of the invention.

[0008] FIG. 1 is a component block diagram illustrating a computing device suitable for implementing an aspect.

[0009] FIG. 2 is a component block diagram illustrating an example multi-core processor suitable for implementing an aspect.

[0010] FIG. 3 is a component block diagram illustrating an example system on chip (SoC) suitable for implementing an aspect.

[0011] FIG. 4 is an illustration of memory contents stored in various configurations relative to respective memory regions in a memory in accordance with an aspect.

[0012] FIG. 5 is an illustration of an interaction of memories in a memory hierarchy monitored by a stream monitor in accordance with an aspect.

[0013] FIG. 6 is process flow diagram illustrating an aspect method for implementing an approximation of execution events using memory hierarchy monitoring.

[0014] FIG. 7 is process flow diagram illustrating an aspect method for identifying memory contents of a monitored memory access request.

[0015] FIG. 8 is process flow diagram illustrating an aspect method for monitoring a memory access request resulting in a hit or a miss.

[0016] FIG. 9 is process flow diagram illustrating an aspect method for monitoring a memory access request targeting a memory that is not monitored.

[0017] FIG. 10 is component block diagram illustrating an example mobile computing device suitable for use with the various aspects.

[0018] FIG. 11 is component block diagram illustrating an example mobile computing device suitable for use with the various aspects.

[0019] FIG. 12 is component block diagram illustrating an example server suitable for use with the various aspects.

DETAILED DESCRIPTION

[0020] The various aspects will be described in detail with reference to the accompanying drawings. Wherever possible, the same reference numbers will be used throughout the drawings to refer to the same or like parts. References made to particular examples and implementations are for illustrative purposes, and are not intended to limit the scope of the invention or the claims.

[0021] The terms "computing device" and "mobile computing device" are used interchangeably herein to refer to any one of all of cellular telephones, smartphones, personal or mobile multi-media players, personal data assistants (PDAs), laptop computers, tablet computers, smartbooks, ultrabooks, palm-top computers, wireless electronic mail receivers, multimedia Internet enabled cellular telephones, wireless gaming controllers, and similar personal electronic devices that include a memory, and a multi-core programmable processor. While the various aspects are particularly useful for mobile computing devices, such as smartphones, which have limited memory and battery resources, the aspects are generally useful in any electronic device that implements a plurality of memory devices and a limited power budget in which reducing the power consumption of the processors can extend the battery-operating time of the mobile computing device.

[0022] The term "system-on-chip" (SoC) is used herein to refer to a set of interconnected electronic circuits typically, but not exclusively, including a hardware core, a memory, and a communication interface. A hardware core may include a variety of different types of processors, such as a general purpose processor (GPP), a central processing unit (CPU), a digital signal processor (DSP), a graphics processing unit (GPU), an accelerated processing unit (APU), an auxiliary processor, a single-core processor, and a multi-core processor. A hardware core may further embody other hardware and hardware combinations, such as a field programmable gate array (FPGA), an application-specific integrated circuit (ASIC), other programmable logic device, discrete gate logic, transistor logic, performance monitoring hardware, watchdog hardware, and timer references. Integrated circuits may be configured such that the components of the integrated device reside on a single piece of semiconductor material, such as silicon.

[0023] Aspects include methods and computing devices implementing such methods for executing event monitoring by monitoring instruction request lines to detect or recognize certain execution events. An aspect may use memory addresses as unique function identifiers in order to increase the probability of detecting execution events.

[0024] Code may be copied from a storage device or a processor to a main memory when an instruction execution function is called, and a code may jump to the entry point of the function. The code may be copied to an instruction cache from the storage device or the processor either instead of the main memory or in addition to the main memory. The code may also be copied from the main memory to the instruction cache. No matter the manner in which the code is copied to the instruction cache, an association is created between execution events, such as calling the instruction execution function and

cache entries. This association may be recognized at a bus level by observing instruction request lines, such as a miss instruction stream from the cache and non-cacheable accesses to the main memory. Misses, monitoring instruction request lines can provide information for monitoring of API calls triggered by specific execution events.

[0025] In an aspect, a stream monitor executing in hardware, software, or a combination of hardware and software may determine a memory address to monitor for an identified function. Based on the memory address, the stream monitor may monitor a memory region of the instruction cache and/or main memory. The memory region may be any portion of the instruction cache and/or main memory, for example a block of memory or a page of memory. The stream monitor may monitor all access requests to the memory region to identify access request containing the identified address as an entry point.

[0026] The memory address may point to a line in the instruction cache and/or main memory containing multiple functions. Monitoring access requests for the memory address may result in false identifications of an execution event if the function accessed at the memory address is a function other than the identified function. The memory address may be used in conjunction with other identifiers for the identified function to increase the probability of successfully detecting execution events. Examples of such other identifiers may include entry point, exit point, callee functions, caller functions, parameters (e.g., non-integers and buffers), unique instructions and patterns (e.g., loops), cache footprint, local variables, and return values.

[0027] With multiple cache levels, it may be difficult to monitor streams from each of the cache levels. Instructions stored at one of the difficult-to-monitor cache levels may not be monitored until the instructions are evicted from the cache. Thus, exit events may be lost for the excess requests to these difficult-to-monitor cache levels. The stream monitor may mark access request as non-cacheable to force a cache miss and to direct the access request, and subsequent access request for the same memory address, to the main memory so that the access request may be monitored.

[0028] Being able to monitor access requests to the cache and/or main memory for specified memory address reduces the amount of monitoring that would otherwise have to be done to monitor CPU instructions because not all of the memory access requests must be monitored. Further, the frequency with which access requests to the specified memory address are made is likely slower than the processing frequency of the CPU. The memory addresses may be used in conjunction with other identifiers to identify access requests for certain functions where monitoring only the memory address may lead to false positives. Difficult-to-monitor access requests in certain levels of the cache may be directed to force the excess requests to the main memory in order to make the access request more visible to the stream monitor.

[0029] FIG. 1 illustrates a system including a computing device 10 in communication with a remote computing device 50 suitable for use with the various aspects. The computing device 10 may include an SoC 12 with a processor 14, a memory 16, a communication interface 18, and a storage memory interface 20. The computing device may further include a communication component 22 such as a wired or wireless modem, a storage memory 24, an antenna 26 for establishing a wireless connection 32 to a wireless network 30, and/or the network interface 28 for connecting to a wired

connection 44 to the Internet 40. The processor 14 may include any of a variety of hardware cores, as well as a number of processor cores. The SoC 12 may include one or more processors 14. The computing device 10 may include more than one SoC 12, thereby increasing the number of processor 14 and processor cores. The computing device 10 may also include processor 14 that are not associated with an SoC 12. Individual processors 14 may be multi-core processors as described below with reference to FIG. 2. The processors 14 may each be configured for specific purposes that may be the same as or different from other processors 14 of the computing device 10. One or more of the processors 14 and processor cores of the same or different configurations may be grouped together.

[0030] The memory 16 of the SoC 12 may be a volatile or non-volatile memory configured for storing data and processor-executable code for access by the processor 14. The computing device 10 and/or SoC 12 may include one or more memories 16 configured for various purposes. In an aspect, one or more memories 16 may include volatile memories such as random access memory (RAM) or main memory, or cache memory. These memories 16 may be configured to temporarily hold a limited amount of data and/or processor-executable code instructions that is requested from non-volatile memory loaded to the memories 16 from non-volatile memory in anticipation of future access based on a variety of factors, and/or intermediary processing data and/or processor-executable code instructions produced by the processor 14 and temporarily stored for future quick access without being stored in non-volatile memory.

[0031] In an aspect, the memory 16 may be configured to store processor-executable code, at least temporarily, that is loaded to the memory 16 from another memory device, such as another memory 16 or storage memory 24, for access by one or more of the processors 14. In an aspect, the processor-executable code loaded to the memory 16 may be loaded in response to execution of a function by the processor 14. Loading the processor-executable code to the memory 16 in response to execution of a function may result from a memory access request to the memory 16 that is unsuccessful, or a miss, because the requested processor-executable code is not located in the memory 16. In response to a miss, a memory access request to another memory device may be made to load the requested processor-executable code from the other memory device to the memory device 16. In an aspect, loading the processor-executable code to the memory 16 in response to execution of a function may result from a memory access request to another memory device, and the processor-executable code may be loaded to the memory 16 for later access.

[0032] The communication interface 18, communication component 22, antenna 26, and/or network interface 28, may work in unison to enable the computing device 10 to communicate over a wireless network 30 via a wireless connection 32, and/or a wired network 44 with the remote computing device 50. The wireless network 30 may be implemented using a variety of wireless communication technologies, including, for example, radio frequency spectrum used for wireless communications, to provide the computing device 10 with a connection to the Internet 40 by which it may exchange data with the remote computing device 50.

[0033] The storage memory interface 20 and the storage memory 24 may work in unison to allow the computing device 10 to store data and processor-executable code on a

non-volatile storage medium. The storage memory 24 may be configured much like an aspect of the memory 16 in which the storage memory 24 may store the processor-executable code for access by one or more of the processors 14. The storage memory 24, being non-volatile, may retain the information even after the power of the computing device 10 has been shut off. When the power is turned back on and the computing device 10 restarts, the information stored on the storage memory 24 may be available to the computing device 10. The storage memory interface 20 may control access to the storage memory 24 and allow the processor 14 to read data from and write data to the storage memory 24.

[0034] Some or all of the components of the computing device 10 may be efficiently arranged and combined while still serving the necessary functions. Moreover, the computing device 10 may not be limited to use of each of the components, and multiple instances of each component may be included in various configurations of the computing device 10.

[0035] FIG. 2 illustrates a multi-core processor 14 suitable for implementing aspects. The multi-core processor 14 may have a plurality of homogeneous or heterogeneous processor cores 200, 201, 202, 203. The processor cores 200, 201, 202, 203 may be homogeneous in that, the processor cores 200, 201, 202, 203 of a single processor 14 may be configured for the same purpose and have the same or similar performance characteristics. For example, the processor 14 may be a general purpose processor, and the processor cores 200, 201, 202, 203 may be homogeneous general purpose processor cores. Alternatively, the processor 14 may be a graphics processing unit, a digital signal processor, and the processor cores 200, 201, 202, 203 may be homogeneous graphics processor cores or digital signal processor cores, respectively. For ease of reference, the terms "processor" and "processor core" may be used interchangeably herein.

[0036] The processor cores 200, 201, 202, 203 may be heterogeneous in that, the processor cores 200, 201, 202, 203 of a single processor 14 may be configured for different purposes and/or have different performance characteristics. Example of such heterogeneous processor cores may include what are known as "big.LITTLE" architectures in which slower, low-power processor cores may be coupled with more powerful and power-dissipating processor cores.

[0037] In the example illustrated in FIG. 2, the multi-core processor 14 includes four processor cores 200, 201, 202, 203 (i.e., processor core 0, processor core 1, processor core 2, and processor core 3). For ease of explanation, the examples herein may refer to the four processor cores 200, 201, 202, 203 illustrated in FIG. 2. However, the four processor cores 200, 201, 202, 203 illustrated in FIG. 2 and described herein are merely provided as an example and in no way are meant to limit the various aspects to a four-core processor system. The computing device 10, the SoC 12, or the multi-core processor 14 may individually or in combination include fewer or more than the four processor cores 200, 201, 202, 203 illustrated and described herein.

[0038] FIG. 3 illustrates an example SoC 12 including a cache memory controller 300, a cache memory 302, a main memory controller 304, a main memory 306, stream monitor 310, and other components such as the components of the SoC 12 described above. The SoC 12 may also include or be communicatively connected to a storage memory controller 308 and the storage memory 24. Each of the cache memory 302, the main memory 306, and the storage memory 24 may

be configured to store memory contents, such as data and/or processor-executable code. The memory contents may be stored at specific locations identified by physical addresses of the cache memory 302, the main memory 306, and the storage memory 24. In an aspect, memory access requests to the memories 24, 302, and 306 may be made using a virtual address that may be translated to the physical address of the respective memory 24, 302, and 306 in order to retrieve the requested memory contents of the memory access request. The storage locations of any of the data and/or processor-executable code may change with time. The physical addresses associated with the data and/or processor-executable code may be updated in a data structure mapping the locations of the data and/or processor-executable code for access by the processor 14.

[0039] The cache memory 302 may be configured to temporarily store data and/or processor-executable code for quicker access than is achievable accessing the main memory 306 or the storage memory 24. The cache memory 302 may be dedicated for use by a single processor 14 or shared between multiple processors 14, and/or subsystems (not shown) of the SoC 12. In an aspect, the cache memory 302 may be part of the processor 14, and may be dedicated for use by a single processor core or shared between multiple processor cores of the processor 14. The cache memory controller 300 may manage access to the cache memory 302 by various processors 14 and subsystems (not shown) of the SoC 12. The cache memory controller 300 may also manage memory access requests for access from the cache memory controller 300 to the main memory 306 and the storage memory 24 for retrieving memory contents that may be requested from the cache memory 302 by the processor 14, but not found in the cache memory 302 resulting in a cache miss.

[0040] The main memory 306 may be configured to temporarily store data and/or processor-executable code for quicker access than when accessing the storage memory 24. The main memory 306 may be available for access by the processors 14 of one or more SoC's 12, and/or subsystems (not shown) of the SoC 12. The main memory controller 304 may manage access to the main memory 306 by various processors 14 and subsystems (not shown) of the SoC 12 and computing device. The main memory controller 304 may also manage memory access requests for access by the main memory controller 304 to the storage memory 24 for retrieving memory contents that may be requested from the main memory 306 by the processor 14 or the cache memory controller 300, but not found in the main memory 305 resulting in a main memory miss.

[0041] The storage memory 24 may be configured to provide persistent storage of data and/or processor-executable code for retention when the computing device is not powered. The storage memory 24 may have the capacity to store more data and/or processor-executable code than the cache memory 302 and the main memory 306, and to store data and/or processor-executable code including those not being used or predicted for use in the near future by the processors 14 or subsystems (not shown) of the SoC 12. The storage memory 24 may be available for access by the processors 14 of one or more SoC's 12, and/or subsystems (not shown) of the SoC 12. The storage memory controller 308 may manage access to the storage memory 24 by various processors 14 and subsystems (not shown) of the SoC 12 and computing device. The storage memory controller 24 may also manage memory access requests for access from the cache memory controller

300 and the main memory controller 304 to the storage memory 24 for retrieving memory contents that may be requested from the cache memory 302 or the main memory 306 by the processor 14, but not found in the cache memory 302 or the main memory 305 resulting in a cache memory miss.

[0042] The stream monitor 310 may be configured to monitor communications between the processor 14, subsystems of the SoC 12 (not shown), the cache memory controller 300, the main memory controller 304, and the storage memory controller 308. The stream monitor 310 may monitor these communications by monitoring the communication activity on one or more communication buses 312 connecting the processor 14 and/or the sub-systems of the SoC 12 that are/were part of the controllers 300, 304, and 308.

[0043] Monitoring the communications between the components of the SoC 12 may include monitoring instruction request lines used to approximate execution events. The instruction request lines may be used to identify the requested processor-executable code of a memory access request to the memories 24, 302, and 306. Monitoring all instruction request lines may be overly taxing or inefficient in some implementation because not all the requested processor-executable code may be of interest for approximating or detecting execution events. So in an aspect, monitoring instruction request lines may be implemented selectively by determining processor-executable code of interest and an address in one or more of the memories 24, 302, and 306 associated with the processor-executable code.

[0044] The stream monitor 310 may monitor communications to the memories 24, 302, and 306 for accesses of memory regions containing the processor-executable code. The sizes and/or types of the memory regions may vary for different aspects, including a line, a block, a page, or any other memory unit size and/or type. In an aspect, the stream monitor 310 may monitor communications for memory access requests containing entry point addresses to the memories 24, 304, and 306. Identifying a memory access request including the entry point address may allow for identification of the processor-executable code requested for execution and identification of an execution event related to the processor-executable code. It should be understood that the entry point address is simply one example of many factors that may be used to identify the processor-executable code requested for execution. References to the entry point address in the descriptions of the various aspects are for example purposes only and are not meant to be limiting as to the factors that may be used to identify processor-executable code requested for execution.

[0045] In an aspect, monitoring the communications between the components of the SoC 12 may include monitoring instruction request lines, and using a combination of factors to approximate or recognize certain execution events. In various aspects, the entry point address to the memories 24, 302, and 306 may not suffice to identify the processor-executable code requested for execution. For example, the memories 24, 302, and 306 may be divided into storage units, such as the various memory regions described above. The size of a memory region may vary for the different memories 24, 302, and 306. In an aspect where a memory region contains a single processor-executable code, the entry point address indicating a certain memory region may be sufficient to use for identifying the processor-executable code. In an aspect in which a memory region contains at least part of multiple

processor-executable codes, the entry point address indicating a certain memory region may not be able to uniquely identify a single processor-executable code.

[0046] As demonstrated above, a factor for identifying the processor-executable code requested for execution may not always uniquely identify the processor-executable code. This may cause ambiguity identifying the processor-executable code requested for execution. In an aspect, the stream monitor 310 may employ at least two of the following factors to identify the processor-executable code of a memory access request:

- [0047] Entry point address;
- [0048] Exit point address;
- [0049] Caller function;
- [0050] Parameters (e.g., non-integers, buffers);
- [0051] Unique instructions and patterns (e.g., loops);
- [0052] Cache footprint (e.g., lines in the cache memory 302);
- [0053] Local variables; and
- [0054] Return value, whence or a return value is written,

there is a chance that a new function call may happen. [0056] The overhead cost of measuring the factors for identifying the processor-executable code requested for execution may cause degradation of performance of the computing device for various tasks and resources. Such tasks may include general or specific processing, including identifying the processor-executable code requested for execution. The performance degradation on resource may include power availability. Substituting a factor(s) with lower overhead cost for the factor(s) with greater overhead cost may help reduce the performance degradation.

[0057] In an aspect, monitoring all, or even a portion of the communications between the components of the SoC 12 may be difficult. The number and speed of the communications may be beyond the capacity of the stream monitor 310. This may be especially true for monitoring communications to multiple memories 24, 302, and 306 when any of them have a multilevel memory hierarchy. The stream monitor 310 may lose track of processor-executable code that is moved around within in a multilevel memory hierarchy. In an aspect, the stream monitor 310 may track a memory access request as un-storeable (or given memory 302 and 306 in order to force a memory miss). The stream monitor 310 may monitor the access request to the other memory 24 and 306 resulting from the memory miss it forced. The stream monitor 310 may use the information obtained from monitoring the memory miss to follow future memory access requests for a processor-executable code, because this information may inform the stream monitor about where processor-executable code is located in the memories 24, 302, and 306.

[0058] In an aspect, the stream monitor 310 may identify the processor-executable code of a memory access request, regardless of whether there is a memory miss during the memory access request. The identified processor-executable code may be used to identify an execution event, which may prompt an API call. In an aspect, the execution event may be identified as unwanted or malicious, and the API call may be used to prevent further execution of the execution event. With the execution event blocked, at least temporarily, the source of the execution event may be identified and handled to prevent future execution of that execution event.

[0059] In an aspect, the above described process may be applied to monitoring memory access request for data, rather

than for processor-executable code. Data producing component may be mapped to memory regions where the components read and write data. The stream monitor 310 may detect reads from the mapped memory region to verify the component or module that is reading the location, and also detect writes to the mapped memory region in case an attacker attempts to corrupt the data.

[0060] In an aspect, processor-executable code may reference to other processor-executable code and/or data stored in the memories 24, 302, and 306 using virtual addresses. For example, this is common when the processor-executable code is executed via a virtual machine run by the processor 14. However communications between some of the components of the SoC 12 via the communication buses 312 may identify locations in the memories 24, 302, and 306 using physical addresses. The stream monitor 310 may monitor memory access requests at various points, some using virtual addresses and some using physical addresses. The stream monitor 310, like other components of the SoC 12 may be configured to understand and use physical addresses to communicate among the components of the SoC 12.

[0061] In an aspect, the stream monitor 310 may also be configured to understand and use virtual addresses in its communications. An aspect of the stream monitor 310 handling virtual addresses may include use of a software component, which may be part of the operating system (OS) kernel, to perform translations from virtual addresses to physical addresses as needed by the stream monitor 310. In an aspect, a translation lookaside buffer (TLB) may be monitored during a memory access request to determine the physical address range, translated by the TLB, for monitoring. In response to the processor-executable code executing, the memory region for monitoring defined by the physical address range, may be stored on a content-addressable memory (CAM) array, and the address(s) may be compared during a refill. In an aspect, code may be injected into each virtual address space to access the region for monitoring by the physical address range.

[0062] The stream monitor 310 may be implemented as software executed by the processor 14, as dedicated hardware, such as on a programmable processor device, or a combination of software and hardware modules. Some or all of the components of the SoC 12 may be differently arranged and/or contributed while still serving the necessary functions. Moreover, the SoC 12 may not be limited to one of each of the components, and multiple instances of each component may be included in various configurations of the SoC 12. Aspect configurations of the SoC 12 may include components, such as the main memory controller 304, the main memory 306, and stream monitor 310 separate from, but connected to the SoC 12 via the communication buses 312.

[0063] FIG. 4 is an illustration of memory contents stored in various configurations relative to respective memory regions 402-412 in a memory 400 in accordance with an aspect. The memory 400 may have any of the above described memories. For example, the cache memory, the main memory, or the storage memory. The memory 400 may be divided into the memory regions 402-412. As discussed above, the memory regions 402-412 maybe of any memory unit size and in type, such as a line, a block, or a page. The memory regions 402-412 may be the memory unit size and in type that may be used for memory access request in a respective computing device.

[0064] Memory contents stored in the memory 400 may include data and/or processor-executable code. For ease of explanation, and without limiting the scope of the description, the following examples are expressed in terms of processor-executable code. The memory regions 402-412 may contain one or more processor-executable codes (P/C) 5; 414-424. For example, the memory region 402 may store a single processor-executable code (P/C) 0-414 within the boundaries of the memory region 402. In another example, the memory region 402 may store one or more processor-executable codes (P/C) 1-414 within the boundaries of memory region 402. In yet another example, the memory region 402 may store multiple processor-executable codes (P/C) 3-420, (P/C) 4-422, and (P/C) 5-424 within the boundaries of the memory region 402.

[0065] In the case of memory region 402 storing a single processor-executable code (P/C) 0-414, the stream monitor may employ the aspect of selectively monitoring instruction request lines by determining processor-executable code of interest and an address in the memory 400 associated with that processor-executable code. The stream monitor may monitor communications to the memory 400 for accesses of memory region 402 containing the processor-executable code (P/C) 0-414. In this aspect, the stream monitor may monitor communications for a memory access request containing an entry point address to the memory 400 in memory region 402. The entry point address of the memory access request related to the memory region 402 may uniquely identify the processor-executable code (P/C) 0-414, as the processor-executable code (P/C) 0-414 is the only processor-executable code to reside in the memory region 402. Therefore, the stream monitor may identify when the processor-executable code (P/C) 0-414 is called for execution by the processor by monitoring a memory access request for the memory region 402.

[0066] The above described aspect applied for monitoring memory region 402 may not be as accurate at identifying the processor-executable code that is being retrieved for execution by the processor when a memory access request involves memory regions 406, 410. Since each of memory regions 406, 410 may store multiple processor-executable codes 416-424, identifying the memory region related to the entry point address of the memory access request may lead to false positives.

[0067] One such false positive may include the identification of multiple processor-executable codes 416-424 of a respective memory region 406, 410 when less than all of the processor-executable codes 416-424 of the respective memory region 406, 410 are retrieved for execution. In this example, while multiple processor-executable codes 416-424 may be retrieved in response to the memory access request, not all of them may be executed. Another false positive may result from identifying processor-executable codes 416-424 known to be stored in one of memory regions 406, 410 accidentally, when the processor-executable code 416-424 being retrieved for execution is not known to be in the same memory region 406, 410. These examples of false positives are similar, except that in the first example a target processor-executable code 416-424 may be identified along with other processor-executable codes 416-424, and in the second example only either processor-executable codes 416-424 may be identified. Therefore, relying on the entry point address of the memory access request alone may produce overly inclusive or incomplete information.

[0068] Identifying the processor-executable code that is being retrieved from memory regions 406, 410 may employ the aspect of using a combination of factors, as illustrated in the examples provided above. Since the entry point address alone may produce overly inclusive or incomplete information, use of other factors may enable the stream monitor to identify a specific processor-executable code 416-424 from the group of other processor-executable codes 416-424 stored in the same memory region 406, 410. While unnecessary, this aspect may also be used to identify the single processor-executable codes (P/C) 0-414 stored in memory region 402.

[0069] In an example, using the entry point address and the exit point address of the memory access may be used to identify processor-executable code (P/C) 2. Since processor-executable code (P/C) 2-418 is partially stored in memory region 406 and memory region 408, the entry point address and exit point address may be associated with a respective memory region 406, 408. Among any of the processor-executable codes 416, 418 stored in memory regions 406, 408, the combination of an entry point address associated with memory region 406 and an exit point address associated with memory region 408 is unique to processor-executable code (P/C) 2-418.

[0070] The other factors may be applied to identify any of the processor-executable codes 416-424. For example, any of the factors may be predetermined to be associated with one or more processor-executable codes 416-424. The stream monitor may be configured to identify any combination of the factors. In response to a memory access request, the stream monitor may identify the factors and compare the factors to the processor-executable codes 416-424 with which they are related. For any two or more factors identified by the stream monitor, the processor-executable codes 416-424 associated with each of the identified factors may be the processor-executable code 416-424 targeted by the memory access request. The stream monitor may be configured such that the factors it identifies are selected for uniquely identifying one of the processor-executable codes 416-424.

[0071] FIG. 5 is an illustration of interaction of memories in a memory hierarchy 500 monitored by the stream monitor in accordance with this aspect. The memory hierarchy 500 may include multiple levels of memory, such as multiple levels of cache memory (cache memory 0-302a, (cache memory 1-302b, the main memory 306, and the storage device 24). If a memory access request monitored by the stream monitor may result in a hit or a miss for the memory 24, 302a, 302b, and 306 targeted by the memory access request. A hit may result from a successful memory access request, such that the memory location of the memory access request is populated and the memory contents are returned 502, 506, 510, 514. A miss may result from an unsuccessful memory access request, such that the memory location of the memory access request is not populated. For a miss, rather than returning the memory contents requested by the memory access request, a supplemental memory access request 504, 508, 512 may be made to a lower level of the memory hierarchy 500. The supplemental memory access request may be made by the memory 302a, 302b, and 306 (or its respective controller) at which the memory access request missed.

[0072] The stream monitor may monitor each memory access request, supplemental memory access request 504, 508, 512, and memory contents return 502, 506, 510, 514. A memory access request may target any of the memories 24, 302a, 302b, 306 in the memory hierarchy 500. In an example,

memory access request may target cache memory 3012a. In response to a hit, the request memory contents may be returned 502. In response to a miss, a supplemental memory access request 504, for the same memory contents, may be made to the next lower level in the memory hierarchy 500, cache memory 1302b. The stream monitor may monitor the output of the cache memory 3012a for the return 502 or the supplemental memory access request 504. In response to the return 502, the stream monitor may identify the information it may use to estimate an execution event. In response to the supplemental memory access request 504 to the cache memory 1302b, the stream monitor may monitor the output of the cache memory 1302b. The supplemental access requests 504, 508, 512 may occur at each level of memory in the memory hierarchy 500, as long as there is a next lower level, until one results in a hit. The stream monitor may monitor the output of the memories 24, 302a, 302b, 306 receiving a supplemental memory access request 504, 508, 512. A supplemental memory access request may be directed to any lower level of memory in the memory hierarchy 500, and does not have to be directed only to the next lower level.

[0073] In an aspect, once memory content is stored to one of the cache memories 3012a, 3012b, the stream monitor may lose track of the memory content until the memory content is evicted. The stream monitor may not be configured to monitor all of the memory levels of the memory hierarchy 500. Memory contents return 502, 506 may be missed by the stream monitor. A memory access request, which may include supplemental memory access request 504, may be sent to a cache memory 3012a, 3012b that the stream monitor does not monitor. The stream monitor may mark the memory access request as non-cacheable. This may force a miss at the targeted cache memory 3012a, 3012b so that the stream monitor may monitor the supplemental memory access request 504, 508, 512, and the potential memory contents return 506, 510, 514, from a memory 24, 302a, 302b, 306 that the stream monitor may be configured to monitor. Marking the memory access requests as non-cacheable may be repeated for each instance of the memory access request, or may be persistent, for example, by saving the marking to a controller of the targeted cache memory 3012a, 3012b. Marking the memory access request as non-cacheable may be implemented at any level of memory of the memory hierarchy 500. However, doing so at lower levels of the memory hierarchy 500, such as the main memory 306, is a lowest level cache memory, cache memory 1302b, in the examples herein, may cause performance degradation. To avoid such performance degrading, the stream monitor may avoid marking memory access requests to the lower memory levels as un-cacheable.

[0074] The memories 24, 302a, 302b, 306 referred to in these examples are not meant to be limiting in number or configuration. The memory hierarchy 500 may have a variety of configurations including more or fewer of any of cache units and storage, memories of varying types, sizes, and speeds. The memory hierarchy 500 may also be configured to have multiple memories 24, 3012a, 3012b, 306 share the same memory level.

[0075] FIG. 6 illustrates an aspect method 600 for implementing an approximation of execution events using memory hierarchy monitoring. The method 600 may be executed in a computing device using software, general purpose or dedicated hardware, such as the processor and/or the stream monitor, or a combination of software and hardware. In block 602, the computing device may receive information identifying

processes to look for by monitoring the memory hierarchy and the factors that the processor can use to identify when those processes are executing. This received information may identify the processes that are the subject of such monitoring as processor-executable code that may be executed by the computing device. In an aspect, the computing device may determine whether an execution event occurs by recognizing when the identified processor-executable code is the target of a memory access request. The information indicating the processor-executable code the execution of which is to be recognized via monitoring the memory hierarchy may be preprogrammed on the computing device or provided to the computing device by a software program running on the computing device. The processor-executable code that is the subject of such monitoring may be related to functions of the computing device that may correlate to execution events in the computing device that are not authorized by a user, or by software selected for execution by the user or a system of the computing device.

[0076] In block 604, the computing device may determine the factors to be used for identifying the processor-executable code that may be executed in response to the memory access request. As described above, one or more factors may be used to identify the processor-executable code that is the target of a memory access request. Such factors may include, for example, an entry point address, an exit point address, callee function, caller functions, parameters (e.g., non-integers, buffers), unique instructions and patterns (e.g., loops), cache footprint (e.g., lines in the cache memory), local variables, and return values. In various aspects, any one factor, such as the entry point address, or combination of factors may be used to uniquely identify the processor-executable code that is the target of a memory access request. As with the identification of the processor-executable code in block 602, the determination of the factors to be used for identifying or recognizing the processor-executable code may be preprogrammed on the computing device or provided to the computing device by a software program running on the computing device.

[0077] In block 606, the computing device may atomic communications between components connected to the communication buses. Examples of such communications include memory access requests, supplemental memory access requests between memories used when there is a miss at memory, and return values in response to the various types of memory access requests. The computing device may monitor the communications for the information relating to the factors(s) that it may use to identify whether a certain processor-executable code is accessed from memory for execution by the computing device. In block 608, the computing device may retrieve the information relating to the factors(s) from the monitored communications for identifying whether the certain processor-executable code is accessed from memory for execution by the computing device. In an aspect, the computing device may be configured to retrieve only the information relating to the factors(s) determined for identifying the certain processor-executable code. In another aspect, the computing device may be configured to retrieve all of the information of a communication on the communication buses, and to parse out the information relating to the factor(s) determined for identifying the certain processor-executable code.

[0078] In determining block 610, the computing device may determine whether the information relating to the factor(s) is retrieved from the monitored communication batches the

factors) determined for identifying the certain processor-executable code. The computing device may compare values of the factors(s) of the target of a memory access request with the information relating to the factor(s) of the monitored communication.

[0079] In response to determining that the retrieved information relating to the factor(s) of the monitored communication do not match the factor(s) determined to be indicative of the certain processor-executable code (i.e., determination block 610 “No”), the computing device may determine that the certain processor-executable code is not being executed by the computing device in block 612. In other words, the target memory contents of the monitored memory access request are not the processor-executable code of interest.

[0080] In response to determining that the retrieved information relating to the factor(s) of the monitored communication match the factor(s) determined to be indicative of the certain processor-executable code (i.e., determination block 610 “Yes”), the computing device may determine that the certain processor-executable code is being executed by the computing device in block 614. In other words, the target memory contents of the monitored memory access request are the processor-executable code of interest. In block 616, the computing device may approximate the occurrence of an execution event based on the determination that the certain processor-executable code is being executed and the certain processor-executable code's relation to the execution event.

[0081] FIG. 7 illustrates an aspect method 700 for identifying memory contents of a monitored memory access request. The method 700 may be executed in a computing device using software, general purpose or dedicated hardware, such as the processor and/or the stream manager, or a combination of software and hardware. The method 700 includes an embodiment of operations that may be implemented in determination block 610 of method 600 described above.

[0082] In determination block 702, the computing device may determine whether a first retrieved information relating to a first factor of the monitored communication matches a first factor determined for identifying the certain processor-executable code. The first factor may be any factor that may be used for identifying the certain processor-executable code as the target memory contents of the monitored memory access request. For example, the first factor may be the entry point address of the memory access request as the entry point address may be used by itself to uniquely identify the certain processor-executable code.

[0083] In response to determining that the first retrieved information relating to the first factor of the monitored communication does not match the first factor determined for identifying the certain processor-executable code (i.e., determination block 702 “No”), the computing device may determine that the certain processor-executable code is not executed by the computing device in block 612.

[0084] In response to determining that the first retrieved information relating to the first factor of the monitored communication does match the first factor determined for identifying the certain processor-executable code (i.e., determination block 702 “Yes”), the computing device may determine whether a next factor is needed to identify the certain processor-executable code in determination block 704. As described above, identifying a processor-executable code as the target of the monitored memory access request may require a combination of factors when a single factor may result in ambiguities or false positives for other processor-executable codes. In other words, the factor may not uniquely identify the certain processor-executable code. The next factor may be any of the factors that have not already been used to identify the certain processor-executable code. In at aspect, the determination of whether a next factor is needed may be based on the overhead of measuring the factors. For example, in response to a factor being too costly to monitor, a next factor that is less costly to monitor while providing suitable recognition of the certain code may be monitored instead. Such a substitute factor may be monitored alone or in conjunction with another factor(s) to identify the certain processor-executable code. A determination that the overhead of a factor is too costly to monitor may be based on whether the overhead for monitoring the factor exceeds a threshold.

[0085] In response to determining that the next factor is not needed to identify the certain processor-executable code (i.e., determination block 704 “No”), the computing device may determine that the certain processor-executable code is executed by the computing device in block 614.

[0086] In response to determining that the next factor is needed to identify the certain processor-executable code (i.e., determination block 704 “Yes”), the computing device may determine whether the next retrieved information relating to the next factor of the monitored communication matches the next factor determined for identifying the certain processor-executable code in determination block 706. In response to determining that the next retrieved information relating to the next factor of the monitored communication does not match the next factor determined for identifying the certain processor-executable code (i.e., determination block 706 “No”), the computing device may determine that the certain processor-executable code is not executed by the computing device in block 612. In response to determining that the next retrieved information relating to the next factor of the monitored communication does match the next factor determined for identifying the certain processor-executable code (i.e., determination block 706 “Yes”), the computing device may determine that a next factor is needed to identify the certain processor-executable code in determination block 704 as described above.

[0087] FIG. 8 illustrates an aspect method 800 for monitoring a memory access request resulting in a hit or a miss. The method 800 may be executed in a computing device using software, general purpose or dedicated hardware, such as the processor and/or the stream manager, or a combination of software and hardware. The method 800 includes an embodiment of operations that may be implemented in block 606 of method 600 described above.

[0088] In determination block 802, the computing device may determine whether a monitored memory access request results in a hit. In other words, the computing device may determine whether the target memory content of the monitored memory access is located at the location of the memory specified by the monitored memory access request. The monitored memory access request may alternatively result in a miss, such that the target memory content of the monitored memory access is not located at the location of the memory specified by the monitored memory access request. In response to determining that the monitored memory access request results in a hit (i.e., determination block 802 “Yes”), in block 608 the computing device may retrieve the information relating to the factor(s) from the monitored communica-

tions for identifying whether the certain processor-executable code is accessed from memory for execution by the computing device.

[0089] In response to determining that the monitored memory access request results in a miss (i.e., determination block 802 "Yes"), the computing device may monitor a supplemental memory access request for the target memory contents in another memory in block 804. A miss for the monitored memory access request may prompt the computing device to generate a supplemental memory access request to another memory that may be at a lower level in the memory hierarchy of the computing device. The computing device may monitor the supplemental memory access request in much the same way that it may monitor the memory access request.

[0090] In determination block 806, the computing device may determine whether the supplemental memory access request results in a hit. In response to determining that the supplemental memory access request results in a hit (i.e., determination block 806 "Yes"), in block 618 the computing device may retrieve the information relating to the factor(s) from the monitored communications for identifying whether the certain processor-executable code is accessed from memory for execution by the computing device. In response to determining that the supplemental memory access request results in a miss (i.e., determination block 806 "No"), the computing device may monitor a supplemental memory access request for the target memory contents in another memory in block 804. A miss for the supplemental memory access request may prompt the computing device to generate another supplemental memory access request to another memory that may be at a lower level in the memory hierarchy of the computing device. Supplemental memory access requests may continue to be generated by the computing device as long as there is a lower level in the memory hierarchy of the computing device to target with the supplemental memory access request.

[0091] FIG. 9 illustrates an aspect method 900 for monitoring a memory access request targeting a memory that is not monitored. The method 900 may be executed in a computing device using software, general purpose or dedicated hardware, such as the processor and/or the system memory, or a combination of software and hardware. In determination block 902, the computing device may determine whether it is able to monitor a target memory of a memory access request. As described above, in computing devices with multi-leveled memory hierarchies, the computing device may not always be configured to monitor the inputs and outputs of each level of the memory hierarchy. As such, some of the information relating to the factors for identifying a processor-executable code of a memory access request may not be retrieved by the computing device. Without the information, the computing device may not be able to accurately identify the processor-executable code of the memory access request.

[0092] In response to determining that the computing device can monitor the target memory of the memory access request (i.e., determination block 902 "Yes"), the computing device may monitor communications between components connected to the communication buses in block 616 as described above.

[0093] In response to determining that the computing device cannot monitor the target memory of the memory access request (i.e., determination block 902 "No"), the computing device may mark a memory access request targeting

the target memory that cannot be monitored as un-cacheable in block 914. Marking the memory access request un-cacheable may force a miss of the target memory, and the computing device may monitor a supplemental memory access request for the target memory contents in another memory in block 804 as described above.

[0094] The various aspects (including, but not limited to, aspects discussed above with reference to FIGS. 1-9) may be implemented in a wide variety of computing systems, which may include an example mobile computing device suitable for use with the various aspects illustrated in FIG. 10. The mobile computing device 1000 may include a processor 1002 coupled to a touchscreen controller 1004 and an internal memory 1006. The processor 1002 may be one or more multi-core integrated circuits designated for general or specific processing tasks. The internal memory 1006 may be volatile or non-volatile memory, and may also be secure and/or encrypted memory, or insecure and/or unencrypted memory, or any combination thereof. Examples of memory types that can be leveraged include but are not limited to DDR, LPDDR, GDDR, WDDR, R-AM, SRAM, DRAM, PRAM, R-RAM, M-RAM, STT-RAM, and embedded DRAM. The touchscreen controller 1004 and the processor 1002 may also be coupled to a touchscreen panel 1012, such as a resistive-sensing touchscreen, capacitive-sensing touchscreen, infrared-sensing touchscreen, etc. Additionally, the display of the computing device 1000 need not have touch screen capability.

[0095] The mobile computing device 1000 may have one or more radio signal transceivers 1008 (e.g., Peanut, Bluetooth, Zigbee, WiFi, RF, radio and antenna 1010, for sending and receiving communications, coupled to each other and/or to the processor 1002). The transceivers 1008 and antenna 1010 may be used with the above-mentioned circuitry to implement the various wireless transmission protocol stacks and interfaces. The mobile computing device 1000 may include a cellular network, wireless modem chip 1016 that enables communication via a cellular network and is coupled to the processor.

[0096] The mobile computing device 1000 may include a peripheral device connection interface 1018 coupled to the processor 1002. The peripheral device connection interface 1018 may be singularly configured to accept one type of connection, or may be configured to accept various types of physical and communication connections, common or proprietary, such as USB, FireWire, Thunderbolt, or PCIe. The peripheral device connection interface 1018 may also be coupled to a similarly configured peripheral device connection port (not shown).

[0097] The mobile computing device 1000 may also include speakers 1014 for providing audio outputs. The mobile computing device 1000 may also include a housing 1020, constructed of a plastic, metal, or a combination of materials, for containing all or some of the components discussed herein. The mobile computing device 1000 may include a power source 1022 coupled to the processor 1002, such as a disposable or rechargeable battery. The rechargeable battery may also be coupled to the peripheral device connection port to receive a charging current from a source external to the mobile computing device 1000. The mobile computing device 1000 may also include a physical button 1024 for receiving user inputs. The mobile computing device 1000 may also include a power button 1026 for turning the mobile computing device 1000 on and off.

[0098] The various aspects (including, but not limited to, aspects discussed above with reference to FIGS. 1-9) may be implemented in a wide variety of computing systems, which may include a variety of mobile computing devices, such as a laptop computer 1100 illustrated in FIG. 11. Many laptop computers include a touchpad touch surface 1117 that serves as the computer's pointing device, and thus may receive drag, scroll, and flick gestures similar to those implemented on computing devices equipped with a touch screen display and described above. A laptop computer 1100 will typically include a processor 1111 coupled to volatile memory 1112 and a large capacity non-volatile memory, such as a disk drive 1113 or flash memory. Additionally, the computer 1100 may have one or more antenna 1106 for sending and/or receiving electromagnetic radiation that may be connected to a wireless data link and/or cellular telephone transceiver 1116 coupled to the processor 1111. The computer 1100 may also include a floppy disc drive 1114 and a compact disc (CD) drive 1115 coupled to the processor 1111. In a notebook configuration, the computer housing includes the touchpad 1117, the keyboard 1118, and the display 1119 all coupled to the processor 1111. Other configurations of the computing device may include a computer mouse or trackball coupled to the processor 1111, via a USB input 1105 as are well known, which may also be used in conjunction with the various aspects.

[0099] The various aspects (including, but not limited to, aspects discussed above with reference to FIGS. 1-9) may be implemented in a wide variety of computing systems, which may include any of a variety of commercially available servers for compressing data in server cache memory. An example server 1200 is illustrated in FIG. 12. Such a server 1200 typically includes one or more multi-core processor assemblies 1201 coupled to volatile memory 1202 and a large capacity non-volatile memory, such as a disk drive 1204. As illustrated in FIG. 12, multi-core processor assemblies 1201 may be added to the server 1200 by inserting them into the racks of the assembly. The server 1200 may also include a floppy disc drive, compact disc (CD) or DVD disc drive 1206 coupled to the processor 1201. The server 1200 may also include network access ports 1203 coupled to the multi-core processor assemblies 1201 for establishing network interface connections with a network 1205, such as a local area network coupled to other fixed system computers and servers, the Internet, the public switched telephone network, and/or a cellular data network (e.g., CDMA, TDMA, GSM, PCS, 3G, 4G, 4G+, or any other type of cellular network).

[0100] Computer program code or "program code" for execution on a programmable processor for carrying out operations of the various aspects may be written in a high level programming language such as C, C++, C#, Smalltalk, Java, JavaScript, Visual Basic, a Structured Query Language (e.g., Transact-SQL), Perl, or in various other programming languages. Program code or programs stored on a computer readable storage medium as used in this application may refer to machine language code (such as object code) whose format is understandable by a processor.

[0101] Many computing devices operating system kernels are organized into a user space (where non-privileged code runs) and a kernel space (where privileged code runs). The separation is of particular importance in Android and other general public license (GPL) environments, in which code that is part of the kernel space must be GPL licensed, while code running in the user space may not be GPL licensed. It should be understood that the various software components/modules

discussed here may be implemented in either the kernel space or the user space, unless expressly stated otherwise.

[0102] The foregoing method descriptions and the process flow diagrams are provided merely as illustrative examples and are not intended to require or imply that the operations of the various aspects must be performed in the order presented. As will be appreciated by one of skill in the art, the order of operations in the foregoing aspects may be performed in any order. Words such as "thereafter," "then," "next," etc. are not intended to limit the order of the operations; these words are simply used to guide the reader through the description of the methods. Further, any reference to claim elements in the singular, for example, using the articles "a," "an" or "the" is not to be construed as limiting the element to the singular.

[0103] The various illustrative logical blocks, modules, circuits, and algorithmic operations described in connection with the various aspects may be implemented as electronic hardware, computer software, or combinations of both. To clearly illustrate this interchangeability of hardware and software, various illustrative components, blocks, modules, circuits, and operations have been described above generally in terms of their functionality. Whether such functionality is implemented in hardware or software depends upon the particular application and design constraints imposed on the overall system. Skilled artisans may implement the described functionality in varying ways for each particular application, but such implementation decisions should not be interpreted as causing a departure from the scope of the present invention.

[0104] The hardware used to implement the various illustrative logics, logical blocks, modules, and circuits described in connection with the aspects disclosed herein may be implemented or performed with a general purpose processor, a digital signal processor (DSP), an application specific integrated circuit (ASIC), a field programmable gate array (FPGA), or other programmable logic device, discrete gate or transistor logic, discrete hardware components, or any combination thereof designed to perform the functions described herein. A general-purpose processor may be a microprocessor, but, in the alternative, the processor may be any conventional processor, controller, microcontroller, or state machine. A processor may also be implemented as a combination of computing devices, e.g., a combination of a DSP and a microprocessor, a plurality of microprocessors, one or more microprocessors in conjunction with a DSP core, or any other such configuration. Alternatively, some operations or methods may be performed by circuitry that is specific to a given function.

[0105] In one or more aspects, the functions described may be implemented in hardware, software, firmware, or any combination thereof. If implemented in software, the functions may be stored as one or more instructions or code on a non-transitory computer-readable medium or a non-transitory processor-readable medium. The operations of a method or algorithm disclosed herein may be embodied in a processor-executable software module that may reside on a non-transitory computer-readable or processor-readable storage medium. Non-transitory computer-readable or processor-readable storage media may be any storage media that may be accessed by a computer or a processor. By way of example but not limitation, such non-transitory computer-readable or processor-readable media may include RAM, ROM, EEPROM, FLASH memory, CD-ROM or other optical disk storage, magnetic disk storage or other magnetic storage devices, or any other medium that may be used to store desired program

code in the form of instructions or data structures and that may be accessed by a computer disk and disc, as used herein, includes compact disc (CD), laser disc, optical disc, digital versatile disc (DVD), floppy disk, and blu-ray disc where disks usually reproduce data magnetically, while discs reproduce data optically with lasers. Combinations of the above are also included within the scope of non-transitory executable readable and processor-readable media. Additionally, the operations of a method or algorithm may reside as one or more combinations or set of codes and/or instructions on a non-transitory processor-readable medium and/or computer-readable medium, which may be incorporated into a computer program product.

[0116] The preceding description of the disclosed aspects is provided to enable any person skilled in the art to make or use the present invention. Various modifications to these aspects will be readily apparent to those skilled in the art, and the generic principles defined herein may be applied to other aspects without departing from the spirit or scope of the invention. Thus, the present invention is not intended to be limited to the aspects shown herein but is to be accorded the widest scope consistent with the following claims and the principles and novel features disclosed herein.

What is claimed is:

1. A method for monitoring communications between components and a memory hierarchy of a computing device, comprising:
 - determining an identifying factor for identifying execution of a processor-executable code;
 - monitoring a communication factor in a communication between the components and the memory hierarchy of the computing device of a same type as the identifying factor;
 - determining whether a value of the identifying factor matches a value of the communication factor; and
 - determining that the processor-executable code is executed in response to determining that the value of the identifying factor matches the value of the communication factor.
2. The method of claim 1, wherein determining whether a value of the identifying factor matches a value of the communication factor comprises:
 - determining whether a value of a first identifying factor matches a value of a first communication factor;
 - determining whether a second identifying factor is needed to identify execution of the processor-executable code;
 - determining whether a value of the second identifying factor matches a value of a second communication factor in response to determining that the second identifying factor is needed to identify execution of the processor-executable code;
3. The method of claim 2, further comprising:
 - determining whether another identifying factor is needed to identify execution of the processor-executable code in response to determining that the value of the second identifying factor matches the value of the second communication factor.
4. The method of claim 2, wherein a type of the first identifying factor and the first communication factor is different from a type of the second identifying factor and the second communication factor; and determining whether a second identifying factor is needed to identify execution of the processor-executable code comprises determining whether the second identifying factor is needed to identify execution of the processor-

executable code in response to in response to determining that the value of the first identifying factor matches the value of the first communication factor, the value of the first communication factor not uniquely identifying the processor-executable code, or an overhead for monitoring the first communication factor exceeds a threshold.

5. The method of claim 1, further comprising determining that the processor-executable code is not executed in response to determining that the value of the identifying factor does not match the value of the communication factor.

6. The method of claim 1, wherein monitoring for a communication factor in a communication between the components and the memory hierarchy of the computing device of a same type as the identifying factor comprises:

- determining whether a memory access request to a first target memory of the memory hierarchy results in a miss, and
- monitoring a supplemental memory access request to a second target memory of a lower level of the memory hierarchy in response to determining that the memory access request results in a miss.

7. The method of claim 1, wherein the communication is associated with a target memory of the memory hierarchy, the method further comprising:

- determining whether the communication can be monitored; and
- marking the communication as executable in response to determining that the communication cannot be monitored.

8. The method of claim 1, wherein a type of the identifying factor and the communication factor comprises one of an entry point address of a target memory, an exit point address of a target memory, a callee function, a caller function, a parameter, a unique instruction, a unique pattern, a cache footprint, a local variable, and a return value.

9. A computing device, comprising a stream monitor configured with stream monitor-executable instructions to perform operations comprising:

- determining an identifying factor for identifying execution of a processor-executable code;
- monitoring a communication factor in a communication between components of the computing device and a memory hierarchy of the computing device of a same type as the identifying factor;
- determining whether a value of the identifying factor matches a value of the communication factor; and
- determining that the processor-executable code is executed in response to determining that the value of the identifying factor matches the value of the communication factor.

10. The computing device of claim 9, wherein the stream monitor is configured with stream monitor-executable instructions to perform operations such that determining whether a value of the identifying factor matches a value of the communication factor comprises:

- determining whether a value of a first identifying factor matches a value of a first communication factor;
- determining whether a second identifying factor is needed to identify execution of the processor-executable code; and

- determining whether a value of the second identifying factor matches a value of a second communication factor.

- in response to determining that the second identifying factor is needed to identify execution of the processor-executable code.
11. The computing device of claim 10, wherein the stream monitor is configured with stream monitor-executable instructions to perform operations further comprising:
- determining whether another identifying factor is used to identify execution of the processor-executable code in response to determining that the value of the second identifying factor matches the value of the second communication factor;
 - 12. The computing device of claim 10, wherein a type of the first identifying factor and the first communication factor is different from a type of the second identifying factor and the second communication factor; and the stream monitor is configured with stream monitor-executable instructions to perform operations such that determining whether a second identifying factor is used to identify execution of the processor-executable code comprises determining whether the second identifying factor is used to identify execution of the processor-executable code in response to an response to determining that the value of the first identifying factor matches the value of the first communication factor, the value of the first communication factor not uniquely identifying the processor-executable code, or an overhead for monitoring the first communication factor exceeds a threshold;
 - 13. The computing device of claim 9, wherein the stream monitor is configured with stream monitor-executable instructions to perform operations further comprising determining that the processor-executable code is not executed in response to determining that the value of the identifying factor does not match the value of the communication factor;
 - 14. The computing device of claim 9, wherein the stream monitor is configured with stream monitor-executable instructions to perform operations such that monitoring for a communication factor in a communication between the components and the memory hierarchy of the computing device of a same type as the identifying factor comprises:
 - determining whether a memory access request to a first target memory of the memory hierarchy results in a miss; and
 - monitoring a supplemental memory access request to a second target memory of a lower level of the memory hierarchy in response to determining that the memory access request results in a miss; - 15. The computing device of claim 9, wherein the stream monitor is configured with stream monitor-executable instructions to perform operations such that the communication is associated with a target memory of the memory hierarchy, and wherein the processor is configured with processor-executable instructions to perform operations further comprising:
 - determining whether the communication can be monitored; and
 - marking the communication un-cacheable in response to determining that the communication cannot be monitored; - 16. The computing device of claim 9, wherein the stream monitor is configured with stream monitor-executable instructions to perform operations such that a type of the identifying factor and the communication factor comprises one of an entry point address of a target memory; an exit point address of a target memory; a callee function; a caller function; a parameters; a unique instruction; a unique pattern; a cache footprint; a local variable; and a return value;
 - 17. A computing device, comprising:
 - means for determining an identifying factor for identifying execution of a processor-executable code;
 - means for monitoring a communication factor in a communication between one or more components of the computing device and a memory hierarchy of the computing device of a same type as the identifying factor;
 - means for determining whether a value of the identifying factor matches a value of the communication factor; and
 - means for determining that the processor-executable code is executed in response to determining that the value of the identifying factor matches the value of the communication factor; - 18. The computing device of claim 17, wherein means for determining whether a value of the identifying factor matches a value of the communication factor comprises:
 - means for determining whether a value of a first identifying factor matches a value of a first communication factor;
 - means for determining whether a second identifying factor is used to identify execution of the processor-executable code; and
 - means for determining whether a value of the second identifying factor matches a value of a second communication factor in response to determining that the second identifying factor is used to identify execution of the processor-executable code; - 19. The computing device of claim 18, further comprising:
 - means for determining whether another identifying factor is used to identify execution of the processor-executable code in response to determining that the value of the second identifying factor matches the value of the second communication factor; - 20. The computing device of claim 18, wherein:
 - a type of the first identifying factor and the first communication factor is different from a type of the second identifying factor and the second communication factor; and
 - means for determining whether a second identifying factor is used to identify execution of the processor-executable code comprises means for determining whether the second identifying factor is used to identify execution of the processor-executable code in response to an response to determining that the value of the first identifying factor matches the value of the first communication factor, the value of the first communication factor not uniquely identifying the processor-executable code, or an overhead for monitoring the first communication factor exceeds a threshold; - 21. The computing device of claim 17, wherein means for monitoring for a communication factor in a communication between the components and the memory hierarchy of the computing device of a same type as the identifying factor comprises:
 - means for determining whether a memory access request to a first target memory of the memory hierarchy results in a miss; and
 - means for monitoring a supplemental memory access request to a second target memory of a lower level of the memory hierarchy in response to determining that the memory access request results in a miss;

22. The computing device of claim 17, wherein the communication is associated with a target memory of the memory hierarchy, the computing device further comprising:
means for determining whether the communication can be monitored; and
means for marking the communication un-cacheable in response to determining that the communication cannot be monitored.
23. The computing device of claim 17, wherein a type of the identifying factor and the communication factor comprises one of an entry point address of a target memory, an exit point address of a target memory, a callee function, a caller function, a parameters, a unique instruction, a unique pattern, a cache footprint, a local variable, and a return value.
24. A non-transitory processor-readable storage medium having stored thereon processor-executable instructions configured to cause a processor of a computing device to perform operations comprising:
determining an identifying factor for identifying execution of a processor-executable code;
monitoring a communication factor in a communication between components and a memory hierarchy of the computing device if a same type as the identifying factor;
determining whether a value of the identifying factor matches a value of the communication factor; and
determining that the processor-executable code is executed in response to determining that the value of the identifying factor matches the value of the communication factor.
25. The non-transitory processor-readable storage medium of claim 24, wherein the stored processor-executable instructions are configured to cause a processor of a computing device to perform operations such that determining whether a value of the identifying factor matches a value of the communication factor comprises:
determining whether a value of a first identifying factor matches a value of a first communication factor;
determining whether a second identifying factor is needed to identify execution of the processor-executable code; and
determining whether a value of the second identifying factor matches a value of a second communication factor in response to determining that the second identifying factor is needed to identify execution of the processor-executable code.
26. The non-transitory processor-readable storage medium of claim 25, wherein the stored processor-executable instructions are configured to cause a processor of a computing device to perform operations further comprising:
determining whether another identifying factor is needed to identify execution of the processor-executable code in response to determining that the value of the second identifying factor matches the value of the second communication factor.
27. The non-transitory processor-readable storage medium of claim 25, wherein the stored processor-executable instructions are of a type of the first identifying factor and the first communication factor is different from a type of the second identifying factor and the second communication factor; and
the stored processor-executable instructions are configured to cause a processor of a computing device to perform operations such that determining whether the processor-executable code comprises determining whether the second identifying factor is needed to identify execution of the processor-executable code in response to determining that the value of the first identifying factor matches the value of the first communication factor, the value of the first communication factor not uniquely identifying the processor-executable code, or an overhead in monitoring the first communication factor exceeds a threshold.
28. The non-transitory processor-readable storage medium of claim 24, wherein the stored processor-executable instructions are configured to cause a processor of a computing device to perform operations such that monitoring for a communication factor in a communication between the components and the memory hierarchy of the computing device of a same type as the identifying factor comprises:
determining whether a memory access request to a first target memory of the memory hierarchy results in a miss; and
monitoring a supplemental memory access request to a second target memory of a lower level of the memory hierarchy in response to determining that the memory access request results in a miss.
29. The non-transitory processor-readable storage medium of claim 24, wherein:
the stored processor-executable instructions are configured to cause a processor of a computing device to perform operations such that the communication is associated with a target memory of the memory hierarchy; and
the stored processor-executable instructions are configured to cause a processor of a computing device to perform operations further comprising:
determining whether the communication can be monitored; and
marking the communication un-cacheable in response to determining that the communication cannot be monitored.
30. The non-transitory processor-readable storage medium of claim 24, wherein the stored processor-executable instructions are configured to cause a processor of a computing device to perform operations such that a type of the identifying factor and the communication factor comprises one of an entry point address of a target memory, an exit point address of a target memory, a callee function, a caller function, a parameters, a unique instruction, a unique pattern, a cache footprint, a local variable, and a return value.



US 20150358810A1

(191) United States

(121) Patent Application Publication

Chao et al.

(101) Pub. No.: US 2015/0358810 A1

(143) Pub. Date: Dec. 10, 2015

(154) SOFTWARE CONFIGURATIONS FOR
MOBILE DEVICES IN A COLLABORATIVE
ENVIRONMENT(171) Applicant: QUALCOMM Incorporated, San
Diego, CA (US)(172) Inventors: Hui Chuo, San Jose, CA (US); Dariin
Suarez Gracia, Santa Clara, CA (US);
Gheorghe Calin Cascaval, Palo Alto,
CA (US)

(211) Appl. No.: 14/300,407

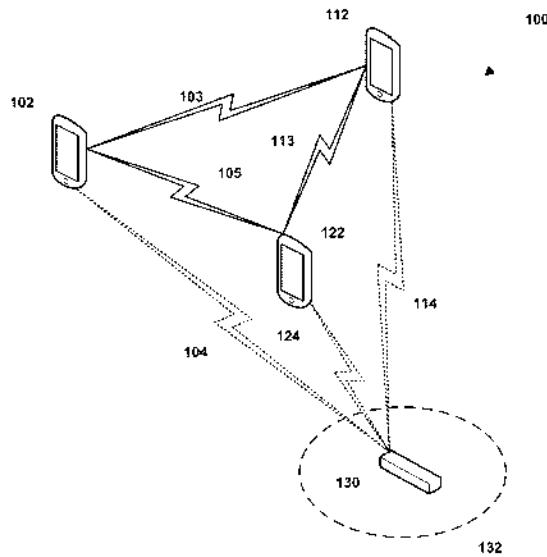
(221) Filed: Jun. 10, 2014

Publication Classification

(151) Int. Cl.
H04W 8/22 (2006.01)
H04W 52/02 (2006.01)
G06F 9/445 (2006.01)
H04W 4/00 (2006.01)(52) U.S. CL.
C10C *H04W 8/22* (2013.01); *H04W 4/00*
(2013.01); *H04W 52/029* (2013.01); *G06F
9/44505* (2013.01)

(57) ABSTRACT

Methods, non-transitory processor-readable storage media, devices, and systems for improving user experience, energy consumption, and performance of a mobile device by automatically configuring applications. An embodiment method includes operations for obtaining, by a processor, operating conditions of the mobile device using an application programming interface; identifying a first of a plurality of software configurations based on the obtained operating conditions of the mobile device, wherein each in the plurality of software configurations define a set of operating parameters for the application; activating the first software configuration with respect to the application; obtaining a first portion of a task shared between a plurality of nearby collaborating devices based on the activated first software configuration, wherein the task may be processing data collectively stored across the plurality of nearby collaborating devices; and performing, by the processor, the first portion of the task using the application configured with the activated first software configuration.



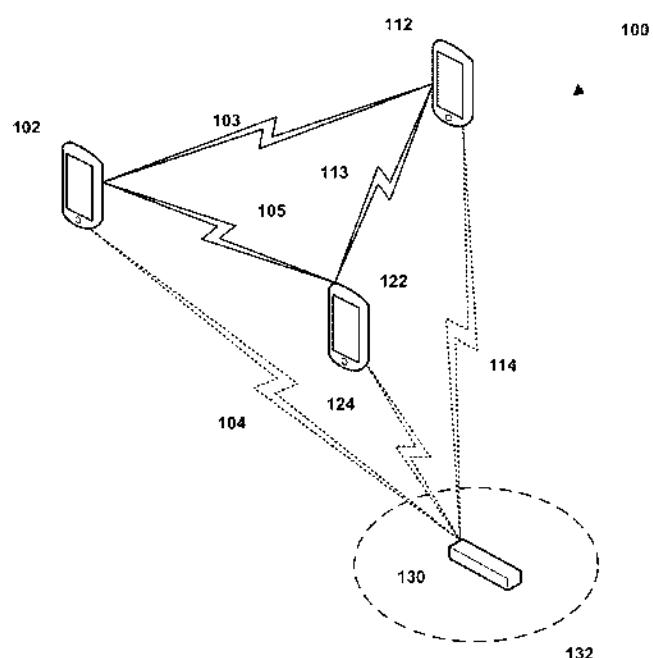


FIG. 1

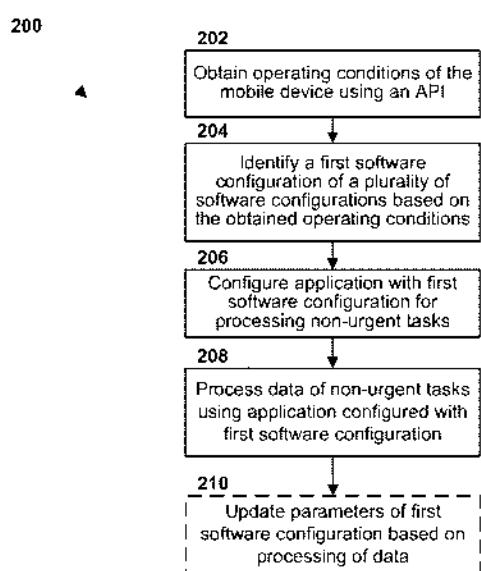


FIG. 2

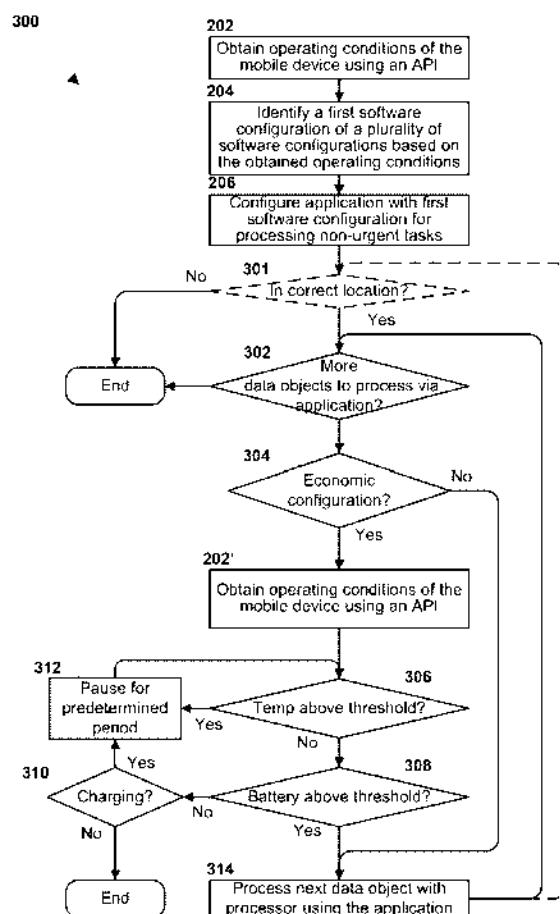


FIG. 3A

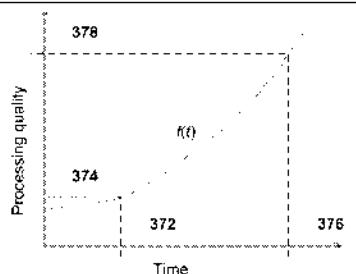
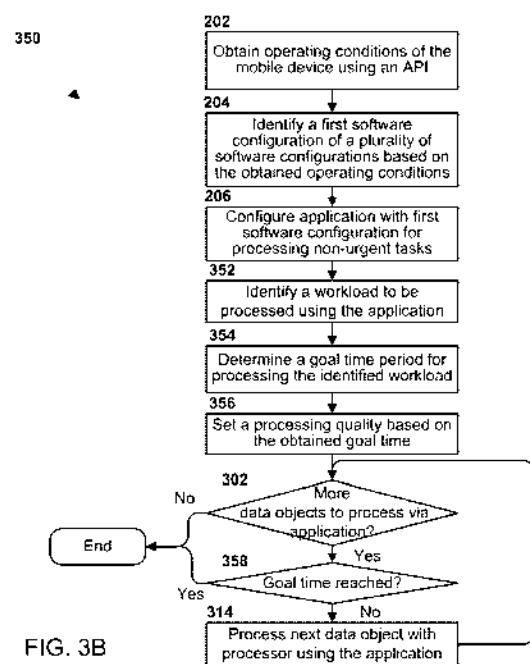


FIG. 3C

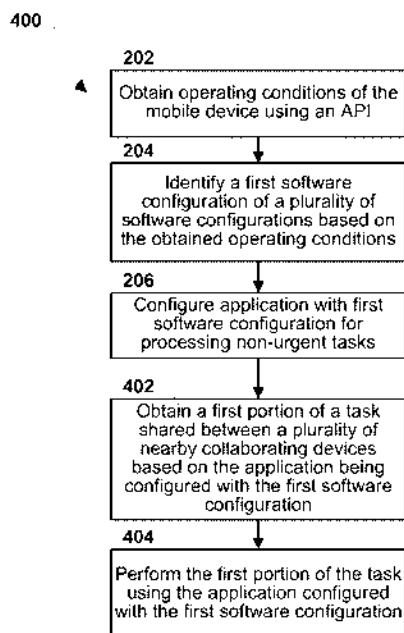


FIG. 4

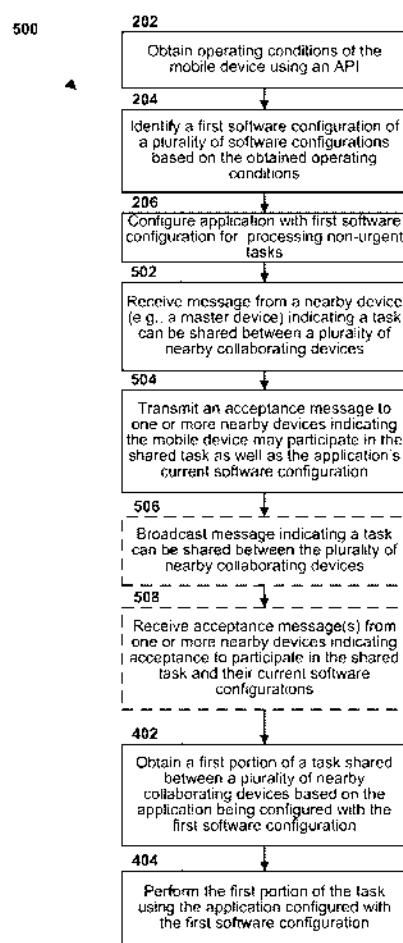


FIG. 5

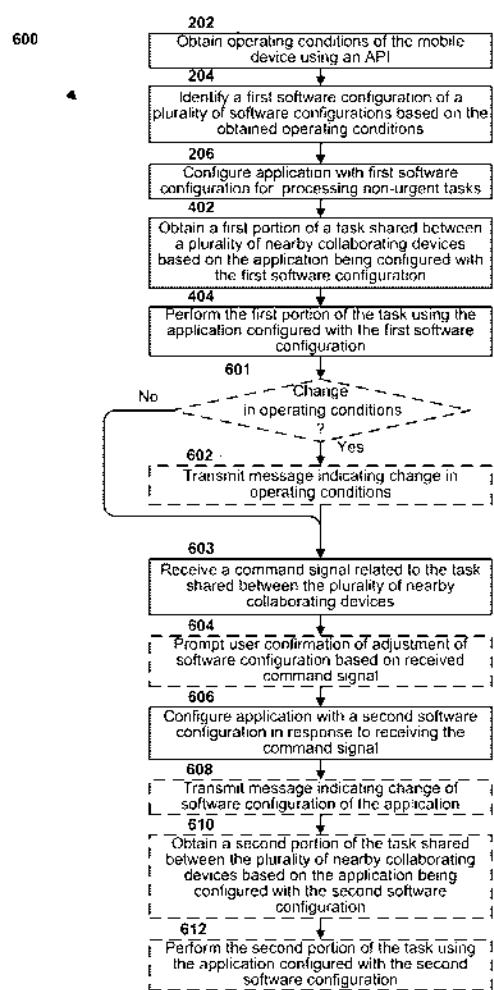


FIG. 6

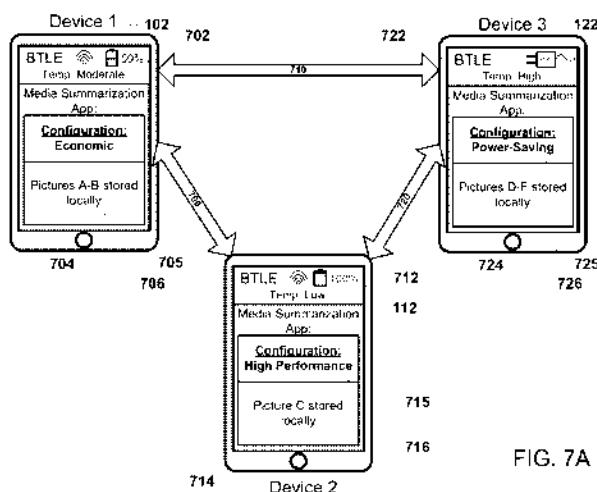


FIG. 7A

	Device	Media Assigned to be summarized	750
752	Device 1	A B	
754	Device 2	C D E	
756	Device 3	F	

FIG. 7B

	Device	Media Assigned to be summarized	760
762	Device 1	A B C	
764	Device 2	B C D E F	
766	Device 3	F	

FIG. 7C

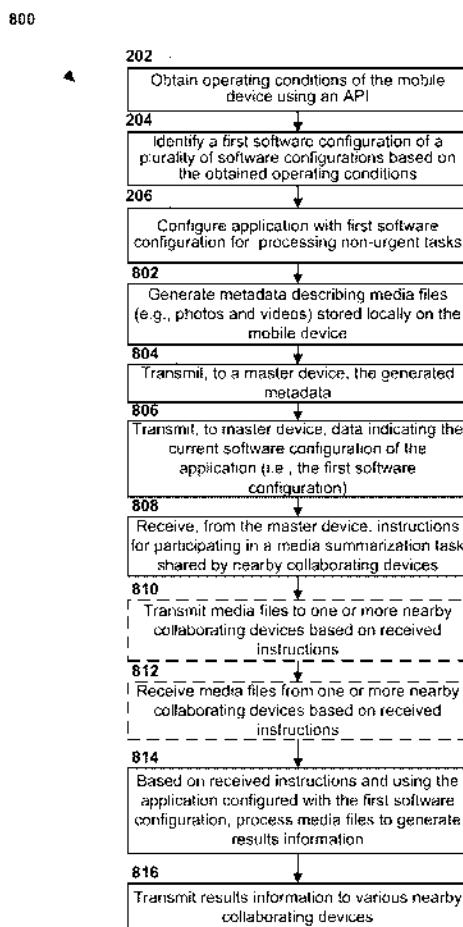


FIG. 8A

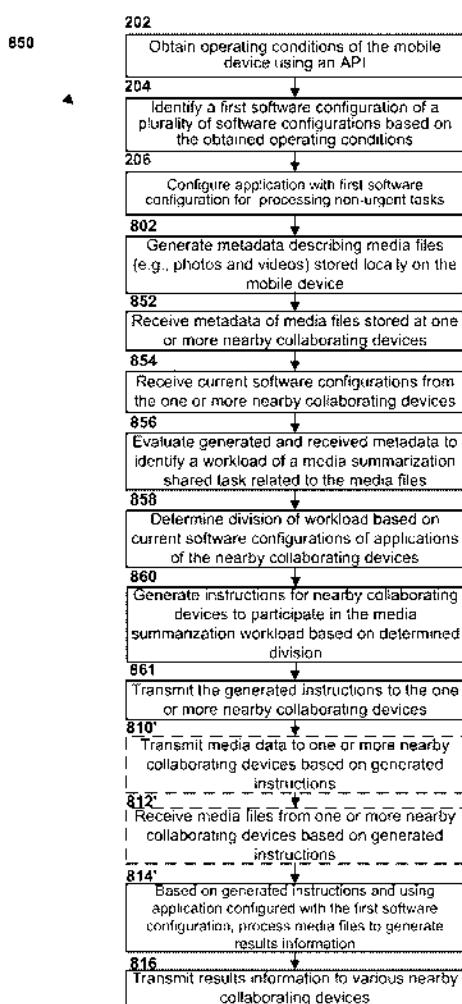


FIG. 8B

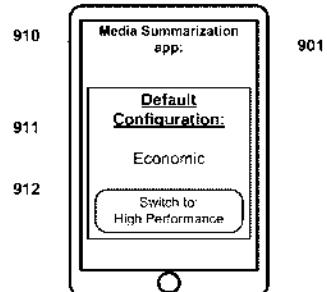


FIG. 9A

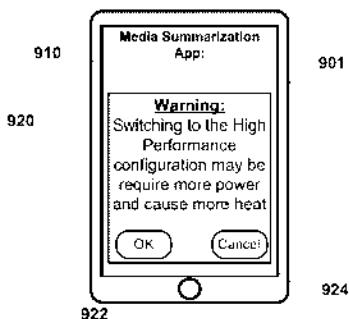


FIG. 9B

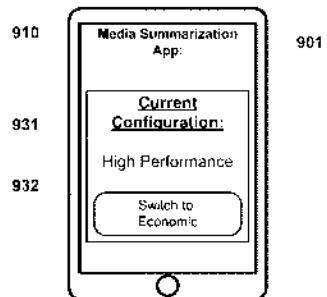


FIG. 9C

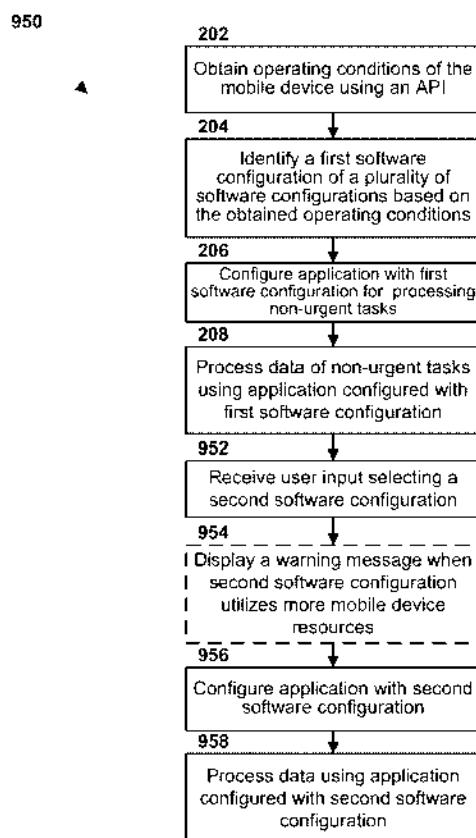


FIG. 9D

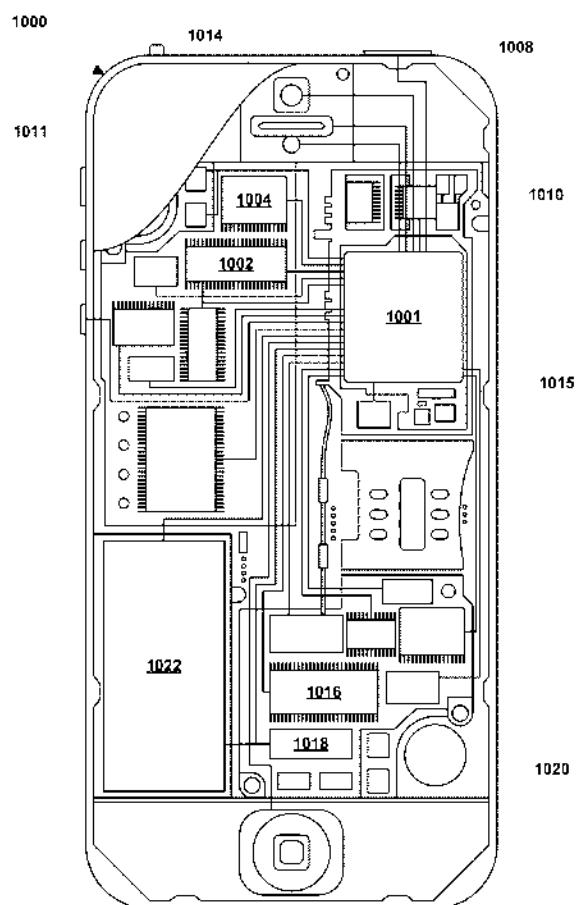


FIG. 10

**SOFTWARE CONFIGURATIONS FOR
MOBILE DEVICES IN A COLLABORATIVE
ENVIRONMENT**

BACKGROUND

[0001] Various processing tasks may be performed by applications running on mobile devices. For example, photo summarization and classification apps may be executed on a smartphone or tablet to organize a user's photos into different groups. Instead of utilizing cloud resources (e.g., cloud servers) or other remote devices, tasks may often be performed locally by applications of a mobile device to maintain privacy and/or avoid overusing a data plan. However, executing these applications to perform such tasks may be computationally expensive, causing significant power drain and/or heat creation at the mobile device, potentially causing hardware damage when experienced for prolonged periods. As some workloads may be important but not urgent (i.e., non-urgent), mobile device users may prefer to have control over how and what applications carry-out tasks.

SUMMARY

[0002] Various embodiments provide systems, methods, devices, and non-transitory processor-readable storage media for improving user experience, energy consumption, and performance of a mobile device by automatically and dynamically configurable applications for collaborative processing during suitable time windows based on device operating states. An embodiment method, performed by a processor of a mobile device executing an application, may include operations for obtaining operating conditions of the mobile device using an application programming interface, identifying a first software configuration of a plurality of software configurations based on the obtained operating conditions of the mobile device in which each in the plurality of software configurations define a set of operating parameters for the processor executing the application, activating the first software configuration with respect to the application, obtaining a first portion of a task shared between a plurality of nearby collaborating devices based on the activated first software configuration, wherein the task is processed data collected, stored, across the plurality of nearby collaborating devices, and performing the first portion of the task using the application configured with the activated first software configuration. In some embodiments, the obtained operating conditions may include one or more of available power conditions, battery conditions, temperature conditions, available communication protocols, available networking interfaces, network connectivity, past usage data, and processor workload.

[0003] In some embodiments, performing the first portion of the task using the application configured with the activated first software configuration may include performing the operations of the first portion of the task in a shortest time period, performing the operations of the first portion of the task until a predetermined threshold (e.g., a battery level, a temperature of the mobile device, and a location of the mobile device) is exceeded, and performing the operations of the first portion of the task in a fixed time period or periodically. In some embodiments, performing the first portion of the task using the application configured with the activated first software configuration includes exchanging data with one or more of the plurality of nearby collaborating devices using a preferred communication protocol.

[0004] In some embodiments, the method may further include receiving a command signal related to the task shared between the plurality of nearby collaborating devices, in which the command signal instructs a change of an active software configuration based on the operating conditions. In some embodiments, the method may further include deactivating the first software configuration with respect to the application in response to receiving the command signal, activating a second software configuration with respect to the application in response to the processor deactivating the first software configuration, and obtaining a second portion of the task shared between the plurality of nearby collaborating devices to be processed by the processor using the application configured with the second software configuration. In some embodiments, the second software configuration may utilize a different communication protocol than the first software configuration.

[0005] Further embodiments include a mobile computing device configured with processor-executable instructions for performing operations of the methods described above. Further embodiments include a non-transitory processor-readable medium on which are stored processor-executable instructions configured to cause a mobile computing device to perform operations of the methods described above. Further embodiments include a communication system including a plurality of mobile computing devices configured with processor-executable instructions to perform operations of the methods described above.

BRIEF DESCRIPTION OF THE DRAWINGS

[0006] The accompanying drawings, which are incorporated herein and constitute part of this specification, illustrate exemplary embodiments of the invention, and together with the general description given above and the detailed description given below, serve to explain the features of the invention.

[0007] FIG. 1 is a component block diagram of a communication system including a plurality of mobile devices within proximity of each other.

[0008] FIG. 2 is a process flow diagram illustrating an embodiment method for a processor of a mobile device executing an application configured with a software configuration to process data based on the mobile device's operating conditions.

[0009] FIG. 3A is a process flow diagram illustrating an embodiment method for a processor of a mobile device executing an application configured with an economic or high-performance software configuration to process data based on operating conditions of the mobile device.

[0010] FIG. 3B is a process flow diagram illustrating an embodiment method for a processor of a mobile device executing an application configured with a fixed-time software configuration to process data within a certain period of time.

[0011] FIG. 3C is an exemplary graph of values of a function plotting time against processing quality that may be achieved using a software configuration such as described in FIG. 3C.

[0012] FIG. 4 is a process flow diagram illustrating an embodiment method for a processor of a mobile device executing an application configured with a software configuration to process a portion of a task shared between a plurality of nearby collaborating devices.

[0013] FIG. 5 is a process flow diagram illustrating an embodiment method for a processor of a mobile device executing an application to exchange messages with nearby devices when participating in a task shared between a plurality of nearby collaborating devices.

[0014] FIG. 6 is a process flow diagram illustrating an embodiment method for a processor of a mobile device executing an application configured with a software configuration to receive a command signal for adjusting software configurations when processing portions of a task shared between a plurality of nearby collaborating devices.

[0015] FIG. 7A is a component block diagram illustrating applications configured with various software configurations executing on a plurality of mobile device performing a shared task.

[0016] FIGS. 7B-7C are graphs that illustrate exemplary divisions of computing workload corresponding to a task shared between processors of mobile devices executing applications configured with various software configurations.

[0017] FIGS. 8A-B are process flow diagrams illustrating embodiment methods for a processor of a mobile device executing an application to perform a portion of a task of summarizing media shared between a plurality of nearby collaborating devices.

[0018] FIGS. 9A-9C are component block diagrams illustrating example interfaces used by a mobile device that indicate and/or various software configurations of an application according to an embodiment.

[0019] FIG. 9D is a process flow diagram illustrating an embodiment method for a processor of a mobile device executing an application configured with first software configuration to configure the application with a second software configuration in response to user inputs.

[0020] FIG. 10 is a component block diagram of a mobile device suitable for use with various embodiments.

DRAFT EDDI DESCRIPTION

[0021] The various embodiments will be described in detail with reference to the accompanying drawings. Wherever possible, the same reference numbers will be used throughout the drawings to refer to the same or like parts. References made to particular examples and implementations are for illustrative purposes, and are not intended to limit the scope of the invention or the claims.

[0022] The word "exemplary" is used herein to mean "serving as an example, instance, or illustration." Any implementation described herein as "exemplary" is not necessarily to be construed as preferred or advantageous over other implementations.

[0023] The terms "mobile device" and "mobile computing device" are used herein to refer to any one or all of cellular telephones, smartphones (e.g., iPhone), web-pads, tablet computers, Internet-enabled cellular or mobile telephones, WiFi enabled electronic computing devices, personal data assistants (PDAs), laptop computers, personal computers, and similar computing devices equipped with at least a processor. In various embodiments, such computing devices may be configured with a network transceiver (or other network interface) to establish a wide area network (WAN) and/or local area network (LAN) connection (e.g., an IEEE 802.11 wireless/wireless network transceiver, a wired connection to the Internet, or WiFi). In some embodiments, such computing devices may also be configured to communicate with nearby devices via short-range transmissions, such as via a trans-

ceiver configured to exchange signals using a particular packet format, structure, and/or wireless protocol, such as Bluetooth, Zigbee, WiFi Direct, etc.

[0024] The term "non-urgent task" is used herein to refer to a task, activity, workload, routine, and/or other operation that may be performed by a mobile application and that is not critical to the functioning of the mobile device, the application, and/or the user of the mobile device. For example, a non-urgent task may be a phone processing operation. Non-urgent tasks are distinguished from system operations, such as operating system routines that control the scheduling and functioning of the various software, modules, and resources (e.g., displays, memory, interfaces, etc.) of the mobile device. Non-urgent tasks are also distinguished from emergency operations or functionalities of applications executing on the mobile device, such as emergency phone dialing operations of a phone application.

[0025] The various embodiments provide methods, devices, systems, and non-transitory process-readable storage media storing instructions for improving user experience, energy consumption, and performance of mobile devices participating in collaborative processing of tasks, for example non-urgent tasks, by automatically and dynamically configuring software configurations of applications based on device operating states of the mobile devices. Individual applications (or "apps") executing on mobile device processors may be configured with software configurations that implement different operational characteristics, additional instructions (e.g., polling operations, etc.), and/or device resource usage (collectively referred to herein as "operating parameters"). For example, when an application is configured with a particular software configuration, a processor executing the application may be constrained to a certain processing speed and/or use of a predefined number of operating system threads. As another example, when an application is configured with another software configuration, a processor executing the application may be capable of utilizing a first resource of the mobile device (e.g., a Bluetooth transceiver) but incapable of utilizing a second resource of the mobile device (e.g., a WiFi transceiver). Processors executing applications may individually evaluate operating conditions of mobile devices against predefined information, such as policies, logic rules, conditions, and/or thresholds, to determine whether various software configurations may be employed at a given time. For example, device users or device manufacturers may set heat and power thresholds that define when an application configured with a certain software configuration may cause a processor to use high power and heat if this overheats the device to violate the device's design, safety and/or regulatory limits. As another example, a user may set a policy that enables a processor executing an application configured with a software configuration to execute a task as fast as possible when it is daytime and/or the mobile device battery is close to full capacity.

[0026] With these software configurations of the embodiment techniques, users of mobile devices may have improved experiences, with non-urgent tasks being performed in less disruptive, more power-efficient ways. For example, a user may experience improved performance of a mobile device executing an application when the application can perform a job task in the shortest amount of time for given battery, power, and/or heat constraint. As another example, users of mobile devices may benefit from more efficient power consumption during execution of a shared workload, with applica-

calions only consuming a minimum energy battery, allowance for a given time constraint at one or a plurality of mobile devices contributing to the shared workload.

[0027] One or more applications installed on or that may be installed on a mobile device may be configured with one of a plurality of software configurations with specific operating parameters that may be predefined by users, developers, and/or device manufacturers. One exemplary software configuration may be a "high-performance" software configuration that enables a processor executing an application to complete tasks associated with the application as soon as possible, but that requires more resources and generates more heat. Another exemplary software configuration may be an "economic" software configuration that may cause the processor executing the application to execute tasks associated with the application in a normal (or default) fashion until a power threshold or heat threshold is exceeded, at which time the execution of the application by the processor may be paused until re-entry conditions are met. For example, while in the economic software configuration, the processor executing the application may periodically compare the current device temperature to a predefined heat threshold, and may cause a task to enter a suspended state for as long as the temperature exceeds the threshold. Another exemplary software configuration may be a "fixed-time" software configuration that controls resource usage by the processor executing the application based on a prediction of how the current resources available in the device may be utilized over a predefined time period. Such a fixed-time software configuration may balance speed and quality of processing of data associated with the application based on the availability time to process a workload. For example, the processor executing the application may determine that rendering settings in a video game engine may need to be reduced in order to maintain a constant frame rate throughout a flight. As another example, processing speeds of a processor executing an image classification application may be different for images with different resolutions, such that with higher image resolution images, the processing quality may be better but may take a longer time to classify each image. Another exemplary software configuration may be a "background power-saving" software configuration that, similar to the economic software configuration, potentially uses slower processing while producing less heat, but may use different thresholds for the processor executing the application to pause operations of a task associated with the application. For example, the background power-saving software configuration may be used when the device executing the application is plugged into a power source (or the battery is full), when it is late at night, or when no other applications or tasks are actively executing on the mobile device's processor (idle state). In some embodiments, when the processor executing the application is configured with the background power-saving software configuration (e.g., when a user selects it via an interface, etc.), a user may choose a fixed time for the processor to run the application when connected to a power source (e.g., late at night).

[0028] In various embodiments, multiple nearby mobile devices within wireless communication range of one another (referred to herein as "collaborating devices") may execute applications configured with software configurations to process a shared task in a collaborative computing environment. For example, a plurality of nearby collaborating devices placed on a charging table (e.g., a hotel table, etc.) may utilize applications configured with various software configurations

to collectively process a single data set as a computing cluster. In such environments, the nearby collaborating devices executing the applications may contribute differently to the shared task based on their different software configurations, exchanging data via wireless communication links, such as WiFi or Bluetooth links. For example, nearby collaborating devices with more resources (e.g., higher battery charge states, a faster processor, more capable communication protocols, etc.) may be configured to execute applications configured with "high-performance" software configurations and thus may be assigned larger, more frequent, and/or more complex workloads of a shared task, while other devices with fewer resources may execute applications with "economic" or "powersaving" software configurations and be assigned smaller, less frequent, and/or less complex workloads of the shared task. As another example, a particular mobile device may be configured to perform a limited number of operations to contribute to a shared workload when the particular mobile device has less available battery power than the other contributing devices. Existing peer-to-peer (P2P) communication techniques or protocols may be used to implement communications between nearby collaborating devices (e.g., existing P2P file protocols, media transfer schemes, etc.), and may include transferring metadata and media files between the collaborating devices. In some embodiments, operations of shared tasks may be assigned to nearby collaborating devices in a round-robin fashion when the devices are not configured with the same software configuration.

[0029] The following is an example to illustrate embodiment software configurations used for processing a shared task. During an out-of-town trip, smartphones of family members may be used to take digital photos during the daytime. Due to memory or other issues (e.g., poor data plan), the photos may not get immediately distributed amongst the various smartphones nor to cloud services (e.g., online file lockers, etc.). At night in a hotel the smartphones may be placed within communication range of each other, such as on a table for charging. Photo summarization applications may be actively executing on the processors of each of the smartphones. Each of the applications may be configured with different software configurations based on individual user preferences and/or operating conditions of the individual mobile devices (e.g., available battery power, device temperature, etc.). Recognizing that the devices are capable of communicating with another, the processors executing the applications on each smartphone may begin peer-to-peer communications with each other, exchanging media files (e.g., via Bluetooth connections) so that some or all of the photos taken by all the smartphones may be consolidated, organized, and summarized via the applications. Based on their individual software configurations, the applications in the smartphones may be used by the smartphones to collaborate in assigning tasks according to capabilities and resources so that the smartphones may process different numbers of photos, with more capable smartphones (e.g., faster processor, more memory, faster data connection, etc.) automatically being assigned more photos to process. After the photo summarization shared task is completed, each family member may use their respective smartphone to see the photos that were captured on the other family members' smartphones that day.

[0030] In some embodiments, one of the nearby collaborating devices may be designated as a "master" device to control the performance of a shared task by orchestrating the

operations and/or software configurations of the applications executing on other collaborating devices. For example, a master device may check the current software configuration of an application executing on another collaborating device, estimate that device's capabilities (e.g., available battery power), and assign workloads to the collaborating device's processor executing the application accordingly. The master device may evaluate processing and memory costs, communication speeds, and various other operating conditions based on the software configurations of the applications executing on the collaborating device when dividing the workload of a shared task. For example, when configured with a power-saving software configuration, the master device may assign workloads of a shared task to collaborating devices that are within a certain distance or proximity of the other collaborating devices so that data can be exchanged via short-range wireless signals (e.g., Bluetooth) in order to minimize the costs of transferring files.

[0031] The master device may organize the shared tasks among the nearby collaborating devices based on the type of task. For example, for certain simple shared tasks, no master device may be needed, as the nearby collaborating devices may easily divide or concurrently perform overlapping workloads. For some complex operations, such as unsupervised stitching or other computational photography techniques, a master device may be identified among the nearby collaborating devices as the most capable device based on its operating conditions (e.g., fastest processor, most battery life, plugged-in to a power source, most memory, etc.) and/or the current software configuration of its application, and thus may be designated the best device for expending energy organizing the shared tasks. In some embodiments, the master device may be designated based on user inputs, such as a user input to begin a shared task on a particular smartphone.

[0032] In some embodiments, applications executing on collaborating devices participating in a shared task may be configured with a common software configuration. In other words, for some shared tasks, processors executing participating applications may be required to activate a consistent software configuration prior to starting performance of the shared task. For example, a master device may transmit messages to the various collaborating devices that cause the applications executing on each to be configured with a "high-performance" software configuration, an "economize" software configuration, or another software configuration that activates communication protocols for improved file transfers between the devices. In some embodiments, collaborating devices may be configured to report or broadcast the current software configurations of applications executing on their processors so that other devices, such as the master device, may identify the current software configurations and potentially transmit messages that cause changes to their current software configurations (e.g., change thresholds, change the active software configurations, etc.). In some embodiments, users of the collaborating devices may be required to provide user inputs to authorize changes to software configurations of applications, such as by pressing an "OK" graphical user interface button rendered on his mobile device in response to receiving a command signal from a master device. In some embodiments, in response to a processor adjusting the software configuration of an application executing on the processor, the processor executing the application may be configured to report such an adjustment to nearby collaborating devices.

[0033] In some embodiments, shared tasks may be paused and/or placed on a waiting list in response to adjustments of software configurations by various applications participating in the shared tasks. Further, a warning message may be displayed to a user of a mobile device in response to the application being configured with a different or more expensive software configuration before a processor executing the application starting operations of a shared task and/or before a suspended task continues after pausing due to a software configuration adjustment.

[0034] In some embodiments, software configurations of applications may cause processors to be configured to use various resources and functionalities of mobile devices, such as different sensors and/or transceivers (e.g., Bluetooth, NFC, Zigbee, WiFi, Ultrasonic, etc.). For example, a processor executing an application configured with a first software configuration may utilize a first communication protocol and/or transceiver, but the processor executing the application may utilize a second communication protocol and/or transceiver when configured with a second software configuration. In this way, based on the operating conditions of the mobile device, processors executing applications may be configured to utilize more efficient resources and/or be capable of performing more workloads of shared tasks.

[0035] In general, processors executing applications may be configured to evaluate current operating conditions of the mobile device to identify which software configuration to use (or activate) for the applications at a given time. Processors executing applications may obtain various contextual information of the mobile device, such as a time of day (e.g., daytime, nighttime, etc.), available power (e.g., battery charge level, whether the device is plugged-in to a power source, etc.), hardware or resources used (e.g., transceivers, memory, processors), etc., and the state of other tasks or applications executing on the mobile device (e.g., no other tasks are running, system idle, etc.). Such operating conditions may be current information and/or estimated information based on current conditions of the mobile device. The processors executing the applications may compare the obtained operating conditions to predefined data stored for the device and associated with various software configurations (e.g., profiles). For example, a processor executing an application may compare a projected battery power consumption of a certain task scheduled to be processed in the near future to a stored profile to identify that the application may be configured with a particular software configuration to best conserve the mobile device battery or maintain a certain quality of service (QoS). In some embodiments, profiles may include past application performance data, such as past processor, power, and other resource usage (e.g., 3G/4G cellular network connections, WiFi transceiver, Bluetooth transceiver, etc.). In this way, the processor executing the application may estimate the potential requirements for processing a task (e.g., predicted power consumption, etc.) and determine a software configuration pre-associated as appropriate for such predicted requirements. In various embodiments, each application capable of being executed on the mobile device may be associated with individual profiles, that may be used to identify different software configurations for each application based on operating conditions of the mobile device.

[0036] Policies that set default software configurations, such as information stored in profiles, may be chosen by manufacturers, developers, and/or the user. In some embodiments, a user may utilize a user interface (e.g., a graphical

user interface of "GUI") to set or modify a default software configuration for a particular application. In some embodiments, such a GUI may provide warning messages to the user when the user or an application developer chooses a more expensive default software configuration. For example, the processor executing the application may prompt the user to authorize a proposed change from a default software configuration by rendering a pop-up window that requests user input for confirmation. Further, the user may choose the priority of each of the tasks applications to be used to choose between multiple actions awaiting execution. Priority lists may be based on the last time a certain kind of task was executed, the least amount of time a task may require to be executed, etc. [0037] In some embodiments, processors executing applications may be configured to automatically adjust user preferences or change ("sidestep") user-defined policies using context-aware logic engines. For example, when a mobile device is charging at night, an application may not be configured to enter an economic software configuration as directed by a previous user input, but instead may be configured to enter the high-performance software configuration. As another example, a processor executing an application may increase a heat threshold when plugged into a power source at night and thus unlikely to be heated by the user (e.g., a high heat threshold that won't allow the device burn a table, etc.). As another example, contrary to user preference, a processor executing an application may switch the application from an economic software configuration to a high-performance software configuration in response to determining that the available battery power is near full capacity in the daytime. Such determinations of processors executing applications may occur only when there is available information, such as past usage information stored on the mobile device, that provides adequate assurances or likelihood that such software configuration changes may not be too risky in otherwise inhibiting the future performance of the application and/or the mobile device in general. For example, changing an application from an economic software configuration to a high-performance software configuration may not occur when the processor executing the application does not have access to stored historical information that indicates whether the user is likely to plug the mobile device into a power outlet in the near future.

[0038] In various embodiments, processors executing applications may be configured to utilize an application programming interface (or API), user-level library, or a mid-tier system for implementing software configurations. Application developers may design applications with internal logic that may cause a processor to call or otherwise invoke various APIs in order to switch between software configurations for the applications, in particular without the assistance of an operating system scheduling routine or other similar central manager. API commands may be used to obtain the operating conditions of the mobile device (e.g., battery state, data plan information, device temperature, etc.), and processors executing applications may compare that obtained information to predetermined quality of service (QoS) information related to the applications in order to determine whether one or various software configurations may be enacted. Therefore, unlike techniques that may require an operating system of the mobile device in a user input to select or determine software configurations, processor executing applications may make such determinations based on the obtained information and implement selected software configurations inde-

pendently. In various embodiments, software configurations and related APIs may be enabled via the use of programming languages, such as Multicore Asynchronous Runtime Environment (MARIE) or Halide.

[0039] The various embodiments relate to application-level software configurations that may be used to control operations of a mobile device processor executing an application, particularly the processing of data for non-critical tasks shared between nearby collaborating devices. Processors executing applications may independently and individually re-configure their operating parameters based on software configurations of the applications without influence from any central or system-wide routine, operating mode, and/or other control logic of the mobile device. Thus, embodiment software configurations are not the same as conventional system-wide or computer-wide modes, logic, and/or configurations that affect all operations of a computing device, such as laptop "eco" or power-saving modes. Instead, the embodiment software configurations may be utilized to discretely adjust the operating parameters and operations of processors executing individual applications of a mobile device with regard to particular tasks. Further, unlike conventional techniques, embodiment techniques may enable processor executing applications to identify appropriate software configurations for the applications to be configured with at a given time based on evaluating operating conditions specific to mobile processing environments, such as the power implications of using different sensors (e.g., cellular, Bluetooth, NFC, Zigbee, WiFi, Ultrasound, etc.) for exchanging signals with nearby devices in a collaborative computing environment.

[0040] Additionally, unlike conventional task scheduling techniques, the embodiment techniques do not schedule processing of applications, but instead utilize various software configurations to enable processor executing applications to perform tasks in safe and convenient ways with reference to power consumption and device temperatures. The embodiment techniques do not prioritize the execution of different applications based on profiles, but instead may compare current operating conditions of a mobile device to information within profiles for individual applications in order to identify respective software configurations (and thus operating parameters) for enabling the convenient execution of each of the individual applications.

[0041] For simplicity, the following descriptions may refer to applications performing actions without reference to a processor. However, as applications necessarily utilize processors, circuitry, and other resources (e.g., operating system, memory, etc.) of a mobile computing device for loading and executing of such actions, it should be appreciated that execution of the applications by device processors, circuits and resources is intended in the following descriptions.

[0042] FIG. 1 illustrates a communication system 100 including a plurality of devices within proximity of each other. In particular, the system 100 may include a first mobile device 102, a second mobile device 112, and third mobile device 122. For example, the communication system 100 may include a plurality of smartphones associated with one or more users. Each of the mobile devices 102, 112, 122 may include various functionalities for obtaining and processing various types of data, such as application and/or communications data. For example, the mobile devices 102, 112, 122 may include various wireless transceivers, such as cellular, Bluetooth, Peanut, WiFi, RL, and/or Zigbee transceivers (or radios), coupled to their respective processors. Using such

transceivers, the mobile devices 102, 112, 122 may be configured to exchange peer-to-peer transmissions with one another via short-range communication links 103, 105, 113. For example, the first mobile device 102 may be configured to transmit Bluetooth packets via the communication link 103 for receipt by the second mobile device 112. Data exchanged between the mobile devices 102, 112, 122 via the communication links 103, 105, 113 may be associated with applications executing on the individual mobile devices 102, 112, 122. For example, the first mobile device 102 may transmit digital imagery files (e.g., photographs, movie files, etc.) to the second mobile device 112 for processing via a media summarization application.

[0043] In some embodiments, the communication system 100 may also include a wireless router 130 (e.g., a WiFi router) associated with a local area network 132 (LAN). The mobile devices 102, 112, 122 may be configured to communicate with the wireless router 130 via wireless communication links 104, 114, 124, and may be configured to exchange peer-to-peer communications via the local area network 132. In various embodiments, the mobile devices 102, 112, 122 may be configured to use different communication links for exchanging data with different devices. For example, due to operating with a software configuration with operating parameters restricting power consumption, the first mobile device 102 may only transmit data to the second mobile device 112 via the short-range communication link 103. As another example, when the third mobile device 122 has deactivated its short-range transceiver due to another software configuration, the second mobile device 112 may transmit data to the third mobile device 122 via the local area network 132 by transmitting data via the wireless router 130 using the communication link 114.

[0044] FIG. 2 illustrates an embodiment method 200 for a mobile device processor executing an application configured with a software configuration to process data based on the mobile device's operating conditions. As described above, using software configurations, processor executing applications may be configured to execute operations of tasks using various operating parameters (e.g., processing speed, quantity, etc.). Such software configurations of applications may be automatically and dynamically invoked by code within the applications based on various contextual information from the mobile device. For example, at runtime, an application written using the Microsoft Asynchronous Runtime Environment (MARL) programming language may be configured so that a processor executing the application may check for certain operating conditions within the mobile device in order to determine whether to switch software configurations of the application to improve efficiency, slow power usage rates, and/or adjust the communication resources (e.g., radios, etc.). The operations of the method 200 may enable the automatic, autonomous adjustment at the application-level of the manner in which tasks are performed by processors executing applications.

[0045] In block 202, the processor executing the application may obtain operating conditions of the mobile device using an application programming interface (API). In other words, at runtime, the processor executing the application may invoke API commands that query up-to-date information about the mobile device that may not otherwise be available to the application. The obtained operating conditions of the mobile device may include data that indicates the activities and status of the device at a given time, and may include one

or more of available power conditions, battery conditions, temperature conditions, available communication protocols, available networking interfaces (e.g., transceivers, etc.), network connectivity and communication conditions (e.g., cellular network data plan information, in the transceiver chain, available bandwidth, cellular network conditions, traffic, signal strength, etc.), past usage data, and processor workload. For example, the obtained data may indicate that the mobile device's available communication protocols include one or more of cellular, Bluetooth, NFC, Zigbee, WiFi, and Ultraband, etc.

[0046] In some embodiments, the processor executing the application may continuously or periodically obtain the operating conditions of the mobile device using the API so that up-to-date conditions (or current conditions) are available to the application during its execution on the mobile device. For example, the mobile device processor in the application may poll operating conditions every few seconds, minutes, etc. during its execution in order to determine whether to periodically or continuously configure its software configuration accordingly.

[0047] The API may be a set of library of commands, calls, and/or routines that may be invoked by the processor executing the application in order to access system-level information of the mobile device. Such an API may be developed by the manufacturer of the mobile device and/or a third-party and may be designed to enable various applications to independently interface with the operating system and/or other logic that controls the various components and software of the mobile device. For example, a certain API command may be used to cause the operating system of the mobile device to return to the processor executing the application a variable value and/or perform an operation that the processor executing the application would not otherwise have access to without the API command. In various embodiments, the API may be a user-level library or a run-time system that may be utilized by the processor when executing various applications. In various embodiments, the API may be enabled for use by the processor executing the application via the use of the MARL and/or Halede programming languages.

[0048] In various embodiments, the operating conditions may include data that indicates whether the mobile device is plugged into a power source (e.g., a power outlet), the time of day, the date (e.g., day, month, year, etc.), the idle state of the mobile device system (e.g., operating system), location information (e.g., GPS coordinates, etc.), scheduled operations (e.g., applications to be performed in the mobile device at a given time in the future, etc.), calendar data, other available functionalities or resources of the mobile device (e.g., available memory, available processor resources, software installed on the mobile device, etc.), unavailable functionalities or resources of the mobile device (e.g., deactivated transceivers, suspended routines, etc.), and the number of tasks or applications currently running on the device. In some embodiments, the operating conditions may also indicate whether there are nearby devices with which the processor executing the application can communicate, such as smartphones within Bluetooth communication range of the mobile device that have been paired via a Bluetooth protocol, etc. In some embodiments, the processor executing the application may utilize a context-aware engine or algorithm to observe the various operating conditions of the mobile device and determine weights or importance assessments of the various

conditions that may be used when identifying the most appropriate software configuration at a given time.

[0049] In some embodiments, the operating conditions may also include any inputs, preferences, settings, and/or selected options set by the user of the mobile device. For example, the operating conditions may include information indicating that the user has previously input a preference for the software configuration that an application should be configured with for a particular time of day, battery power, device temperature reading, etc. The processor executing the application may utilize such user-supplied preferences in context of other operating conditions, such as current workload on a processor, in order to determine appropriate software configurations for the application. In some embodiments, user supplied information may or may not affect the determinations of block 204 described below. In other words, user preferences or policies may be "side-stepped" by the processor executing the application based on an evaluation of some/all of the operating conditions at a given time. For example, operating conditions indicating a dangerously high device temperature may cause the processor executing the application to configure the application with an economic software configuration regardless of a user preference for high-performance software configuration.

[0050] In block 204, the processor executing the application may identify a first software configuration of a plurality of software configurations based on the obtained operating conditions of the mobile device. The first software configuration may be the set of operating parameters that is best suited or most appropriate for enabling the processor executing the application to perform a task at a given time and under the current conditions. To make the identification, the processor executing the application may analyze individual operating conditions, such as a high priority condition (e.g., very low/high battery power, very high/low temperature, etc.) or any combination of operating conditions. For example, the processor executing the application may identify a software configuration that permits aggressive processing of data when the mobile device has full battery power, when it is late at night, and/or there are no other tasks or applications executing on the mobile device. As another example, the processor executing the application may identify a software configuration that causes slower processing when the battery is not fully charged and/or the processor of the mobile device is not idle.

[0051] As described above, the first software configuration may be any of a predefined set of software configurations, such as an economic software configuration used to balance processing of data by a processor executing the application with conserving resources of the mobile device (e.g., little battery power consumption, little heat generation, etc.), a high-performance software configuration used to cause the processor executing the application to perform a task associated with the application as fast as possible without regard to the use of device resources (e.g., high power consumption, high heat generation, etc.), a fixed-time software configuration used to cause the processor executing the application to complete application tasks with variable quality of service but within a set period of time, and a power-saving software configuration used to cause the processor executing the application to perform operations associated with the application in a slow, low-heat generating manner, such as when the mobile device is plugged into a power outlet.

[0052] In some embodiments, each software configuration may be associated with different sets of operating parameters for executing the application that are each related to particular operating conditions of the mobile device. For example, the first software configuration may utilize a first battery power threshold when the mobile device is at a first location based on GPS coordinates and may utilize a second battery power threshold when the mobile device is at a second location. In this way, even when configured with the same particular software configuration, the processor executing the application may experience different behaviors based on changing operating conditions of the mobile device.

[0053] In some embodiments, to identify the first software configuration, the processor executing the application may utilize one or more profiles that correlate certain operating conditions of the mobile device with parameters, guidelines, thresholds, metrics, and other operating constraints for the processor executing the application to observe (i.e., software configuration). For example, the processor executing the application may utilize a power profile that may correlate past (or historical) power usage by the processor executing the application with a suggested set of operating parameters that are well-suited for subsequent similar power usage. Such profiles may be used by the processor executing the application to predict future activities and resource usage of the application, such as by matching current activities indicated by the obtained operating conditions with contextual information stored within the profiles (e.g., previous activities and mobile device operating conditions, etc.).

[0054] In some embodiments, the processor executing the application may identify the first software configuration by comparing the current operating conditions of the mobile device to predefined quality of service (QoS) requirements. In particular, the processor executing the application may determine the quality of service the application may achieve when configured with the operating parameters of the first software configuration and compare this determined quality of service to predefined thresholds in order to determine whether the first software configuration may be used.

[0055] In some embodiments, the processor executing the application may be configured to identify the first software configuration based on user preferences or settings. For example, the obtained operating conditions may indicate that a user has previously selected a certain software configuration to be used for configuring the application (e.g., a policy indicating a high-performance software configuration should be used at a certain time of day or available battery power, etc.). However, the processor executing the application may be configured to side-step user settings in order to identify a most appropriate software configuration for a given operating context. For example, when a user preference for the application to be configured with first software configuration is estimated to exceed an available power allowance when used to process a certain task, cause processor activity above a predefined threshold, and/or reduce the efficiency of the processor executing the application below an acceptable predefined level, the processor executing the application may ignore the user preference and identify the first software configuration based on other operating conditions alone (e.g., environment context awareness).

[0056] In block 206, the processor executing the application may configure the application with the first software configuration for processing non-urgent tasks. In other words, the processor executing the application may activate

the first software configuration. For example, via the application configuration with the first software configuration, the processor may set its processing speed or the rate at which processing cycles are utilized based on the operating parameters defined by the first software configuration. Configuring the application with the first software configuration may include setting various thresholds that may be used by the processor executing the application to determine whether to start or stop performing operations. As described below, the first software configuration may include device temperature and/or battery power threshold values that may be evaluated by the processor executing the application during its execution to determine whether the operations associated with the application may be suspended, rescheduled, or otherwise have an adjusted execution rate. In some embodiments, configuring the application with the first software configuration may include the processor activating or deactivating networking interfaces (e.g., transceivers) and/or using or discontinuing the use of certain communication protocols. For example, the first software configuration may cause the processor executing the application to stop evaluating Bluetooth packets or alternatively may cause the processor executing the application to begin utilizing a WiFi transceiver. In some embodiments, the first software configuration may increase or decrease use of the processor of the mobile device when executing the application, such as by requesting additional or fewer processing cycles or operating system threads.

[0057] In various embodiments, the mobile device processor executing the application may continually and dynamically adjust software configurations as the operating conditions may change from time to time. For example, when the mobile device is not charging, the processor may configure an application to run in power-saving software configuration, but when the battery is fully charged and still plugged in to power, the processor may change the application's software configuration to a high performance software configuration.

[0058] In block 208, the processor executing the application configured with the first software configuration may process data of non-silent tasks using the application. For example, when the application is a media summarization application (or app), the processor executing the application may begin evaluating media files stored on the mobile device in order to classify, categorize, and/or organize the media files. The manner of processing the data by the processor executing the application may be controlled by the first software configuration. In particular, the first software configuration may control the speed at which the data is processed by the processor executing the application, the quality of processing of the data, the amount of time that the processor executing the application may process the data, the total amount of data processed, the number of batches of data processed (e.g., whether the total workload is divided into individual batches of data), the interval of uninterrupted processing of the data (e.g., whether there are pauses between batch operations, etc.), and the functionalities of the mobile device that are used when processing the data (e.g., message buffers, auxiliary processors, memory units, transceivers, communication protocols, sensor units, etc.). As an example, when configured with a high-performance software configuration, the processor executing the application may perform operations for summarizing a set of digital media files as fast as possible without regard for heat generation and/or battery power levels. FIGS. 3A-C illustrate various methods for the

processor processing data using applications configured with particular software configurations.

[0059] In optional block 210, the processor executing the application may update parameters of the first software configuration based on processing the data. For example, the processor executing the application may update threshold values associated with the first software configuration (e.g., active threshold, battery threshold, heat threshold, etc.) based on the effect of processing the data on the operating conditions of the mobile device. In various embodiments, the processor executing the application may obtain additional, more up-to-date operating conditions of the mobile device, similar to the operations described above with reference to block 202, in order to perform the updating operations in optional block 210. For example, after the data associated with the application is processed with the processor via the application that is configured with the first software configuration, the processor executing the application may use API commands to query battery state, device temperature, and other operating conditions to identify whether the first software configuration is adequate for similar conditions encountered in the future.

[0060] FIGS. 3A-3B illustrate embodiment methods for a processor to process data via an application configured with different types of software configurations. In particular, based on an active software configuration of the application (e.g., economic, high-performance, etc.) the processor executing the application may perform various checking operations to determine whether to continue processing data, as well as how to process data. For example, when in the application is configured with an economic software configuration, the processor executing the application may perform soft operations in response to determining a temperature is above a threshold [0061]. For simplicity purposes, the term "data-object" is used in the following descriptions to refer to discrete data elements that may be processed by a processor executing an application as part of a task. However, it should be appreciated that references to data objects may also include instructions and/or operations to be performed by the processor executing the application as part of the task.

[0062] FIG. 3A illustrates an embodiment method 300 for a mobile device processor executing an application to process data when the application is configured with an economic or high-performance software configuration based on operating conditions of the mobile device. As described above, the application executed by the processor may be configured with an economic software configuration in order to preserve battery power and cause a lower level of heat generation in the mobile device, whereas the high-performance software configuration may simply enable the processor executing the application to process data as fast as possible.

[0063] The operations in blocks 202-206 may be similar to the operations in like numbered blocks described above with reference to FIG. 2. In optional determination block 301, the processor executing the application may determine whether the mobile device is within a predefined (or "correct") location for performing the operations of the application configured with the software configuration. The mobile device processor may evaluate various information within the obtained operating conditions against stored data associated with the application and its software configuration to determine whether the mobile device has entered (or is still within) a location appropriate for the application and/or a workload of the application. The mobile device processor may identify

locations (e.g., predefined or current) based on GPS coordinates, service set identifiers (SSID) or other access point identifiers to which the mobile device has connected or identified as within proximity. For example, the mobile device processor executing the application may compare recently obtained GPS coordinates with predefined GPS coordinates associated with the application to determine whether the mobile device is in a correct place for joining in a workload shared by a plurality of other mobile devices. As another example, the mobile device processor executing the application may be configured to share a workload of taking pictures of a sporting event with nearby devices (e.g., smartphones of attendees sitting next to the user of the mobile device, etc.) only while the mobile device is within a stadium, and thus may check whether its current location is in the stadium before commencing with the application operations. In response to determining the mobile device is not within a predefined location appropriate for the operations of the application configured with the software configuration (i.e., optional determination block 301 "No"), the processor executing the application may end the operations of the method 300.

[0064] In response to determining the mobile device is within the predefined location appropriate for the operations of the application configured with the software configuration (i.e., optional determination block 301 "Yes"), the processor executing the application may determine whether there are more data objects to process via the application configured with the first software configuration in determination block 302. For example, the processor executing the application may determine whether it has processed all data within a current workload of a task shared by a plurality of nearby collaborating devices. When the processor executing the application first performs the operations of determination block 302, it may determine whether there is a first data object to process. In response to determining there are no more data objects to process via the application (i.e., determination block 302 "No"), the processor executing the application may end the operations of the method 300. In other words, once the data objects of a task performed by the processor executing the application have all been processed, the application may be suspended, closed, or otherwise removed from the active processing queue of the mobile device processor. In some embodiments, the processor executing the application may deactivate the first software configuration in response to determining there are no more data objects to process (i.e., determination block 302 "No").

[0065] In response to determining there are more data objects to process via the application (i.e., determination block 302 "Yes"), the processor executing the application may determine whether the application is configured to operate with an economic software configuration with the first software configuration in determination block 304. For example, the processor executing the application may evaluate a stored code or variable indicating the identity of the current software configuration to determine whether the economic software configuration is active at a given time. The processor executing the application may make this determination in order to determine whether the additional operations associated with the economic software configuration (i.e., the operations in blocks 202, 306, 308, 310, 312) are needed to be performed or whether these additional operations may be bypassed due to the application being configured with the high-performance software configuration. In response to

determining that the application is not configured with the economic software configuration (i.e., determination block 304 "No"), the processor executing the application may be configured to process data objects as fast as possible and without checking the current temperature and/or battery power (i.e., operate with the high-performance software configuration). For example, when operating with the high-performance software configuration, the processor executing the application may be configured to execute operations associated with the application with high processing quality and/or high-speed processing, regardless of the computational and/or battery power expense or the amount of remaining battery power.

[0066] In response to determining that the application is configured with the economic software configuration (i.e., determination block 304 "Yes"), similar to the operations described above with reference to block 202, the processor executing the application may once again obtain the current operating conditions of the mobile device using the API in block 202. In this way, the processor executing the application may continually determine the up-to-date operating conditions of the mobile device in order to determine whether various thresholds indicated by the first software configuration have been exceeded during the processing of data. For the first execution of the operations in determination block 304, the processor executing the application may not be required to obtain the device operating conditions as the previously obtained information may still be up-to-date.

[0067] In determination block 306, the processor executing the application may determine whether a current temperature of the mobile device indicated in the obtained operating conditions is above a predefined temperature threshold. In other words, based on the operating parameters associated with the economic software configuration, the processor executing the application may determine whether the current device temperature from the obtained operating conditions (information exceeds the temperature threshold of the first software configuration). In response to determining that the current temperature of the mobile device is above the predefined temperature threshold (i.e., determination block 306 "Yes"), the processor executing the application may pause for a predetermined period, such as a number of seconds, minutes, etc. at block 312, and then may continue with the temperature checking operations in determination block 306. For example, the operations of the application may be suspended by the mobile device processor for a period of time. The predetermined period of time may be identified by the processor executing the application as a time period adequate for allowing a certain amount of heat dissipation from the device components.

[0068] In response to determining that the current temperature of the mobile device is not above the predefined temperature threshold (i.e., determination block 306 "No"), the processor executing the application may determine whether the current available battery power from the obtained operating conditions is above a predefined battery threshold based on the parameters of the first software configuration in determination block 308. For example, the processor executing the application may compare the current battery power indicated in the obtained operating conditions to the predefined battery threshold for the economic software configuration.

[0069] In response to determining that the current battery power is not above the predefined battery threshold (i.e., determination block 308 "No"), the processor executing the

application may determine whether the mobile device is currently charging in determination block 310. For example, the processor executing the application may determine whether the mobile device is currently plugged into a wall power outlet based on the obtained operating conditions. In response to determining that the mobile device is currently charging (i.e., determination block 310 "Yes"), the processor executing the application may pause the application in block 312, such as by suspending execution of the application for a predefined number of seconds, before continuing with the temperature checking operation in determination block 306. However, in response to determining that the mobile device is not currently charging (i.e., determination block 310 "No"), the processor executing the application may stop the execution of the method 300. In other words, when there is an inadequate amount of battery power and/or another available power source, the processor executing the application may determine it may not continue to process the data based on the parameters of the first software configuration.

[0070] In response to determining that the current battery power is above the predefined battery threshold (i.e., determination block 310 "Yes"), or in response to determining that the processor executing the application is not configured with the economic software configuration (i.e., determination block 304 "No"), in block 314, the processor executing the application may process a next data object using the application. For example, the processor executing the application may perform a number of operations on a next data object in a set of many objects to be processed, such as performing analysis, decryption, encryption, classification, image recognition, annotation, sorting, and/or other operations, with regard to the next data object. When the operations in block 314 are first performed, the next data object may be the first data object in task. The mobile device may continue with the operations in determination block 302. In some embodiments, the mobile device processor may optimally coordinate with the operations in optimal determination block 304 in response to processing the next data object with the operations in block 314. In this way, the mobile device processor executing the software may continually monitor whether it is within the appropriate location for performing operations, such as processes shared between multiple devices at a time-sensitive or event-centric activity (e.g., as part of event, interest group meeting, etc.).

[0071] FIG. 3B illustrates an embodiment method 350 for a processor executing an application configured with a fixed-time software configuration to process data within a certain period of time. As described above, the processor executing the application may configure itself to operate in a fixed-time software configuration in order to accomplish particular processing objectives within a time frame that is supported by the current operating conditions of the mobile device. Such a fixed-time software configuration may be a trade-off between speed and quality, such that the processor executing the application may only be capable of using a certain amount of resources, a type of algorithm, or other data or function, etc., in order to complete a workload within its allotted time window. For example, when configured to operate with the fixed-time software configuration, the processor executing the application may be required to process digital photos at a reduced resolution in order to complete an image classification before a user is scheduled to go to work, etc.

[0072] The operations in blocks 202-206 may be similar to the operations in the numbered blocks described above with

reference to FIG. 2. In block 352, the processor executing the application may identify a workload to be processed using the application. For example, the processor executing the application may evaluate a queue of files to be summarized to identify the amount of time and/or processing cycles required to process the files. In block 354, the processor executing the application may determine a goal time period for processing the identified workload. The goal time may be determined based on user preferences, such as a predetermined amount of time that the first software configuration may be used at any given time, a user input, such as a time amount entered by a user in response to a prompt from the mobile device via the application, and/or based on the obtained operating conditions of the mobile device. For example, based on available battery power, the processor executing the application may calculate an estimated amount of time for processing the workload at one or more processing qualities, processing speeds, and/or using various device resources or functionalities (e.g., codecs, transcoders, peripheral devices, etc.). In some embodiments, the processor executing the application may determine an optimized goal time and processing quality based on the identified workload. For example, the processor executing the application may determine a balance between the time to complete processing of the data and utilizing an adequate processing quality. In block 356, the processor executing the application may set a processing quality based on the obtained goal time. For example, the processor executing the application may set a computing frequency and/or may activate a particular processing routine or algorithm that may be used to process the workload within the obtained goal time.

[0073] In some embodiments, when the workload includes processing images, the processor executing the application may calculate an image resolution for processing the various images (i.e., data objects) based on a given time. For example, the processor executing the application may calculate an available processing time for each image in the workload with the following:

$$\text{Image resolution } f(t) = \frac{t}{\text{processing time per image}} \quad (1)$$

wherein $f(t)$ is a function taking time amounts (i.e., t) as inputs to output a corresponding image resolution value. Such a function may be illustrated with the graph in FIG. 3C described below.

[0074] The available processing time for each image may then be used to determine the image resolution that may be possible using the following equation:

$$\text{Image resolution } f(t) = \frac{t}{\text{processing time per image}} \quad (1)$$

wherein $f(t)$ is a function taking time amounts (i.e., t) as inputs to output a corresponding image resolution value. Such a function may be illustrated with the graph in FIG. 3C described below.

[0075] The operations in determination block 302 may be similar to the operations in like numbered blocks described above. In response to determining that there are no more data objects to process via the application configured with the first software configuration (i.e., determination block 302 "No"), the processor executing the application may end the method 350. However, in response to determining that there are more data objects to process via the application configured with the first software configuration (i.e., determination block 302 "Yes"), the processor executing the application may determine whether the goal time is reached in determination

block 358. For example, based on unforeseen resource use, such as by other applications on the mobile device, the processor executing the application may not have been able to process the workload with the first software configuration as expected. Therefore, the processor executing the application may be required to end the processing of the data objects using the application configured with the first software configuration when the obtained goal time has elapsed. In response to determining that the obtained goal time has been reached (i.e., determination block 358, "Yes"), the processor executing the application may end the method 350. However, in response to determining that the obtained goal time has not been reached (i.e., determination block 358, "No"), the processor executing the application may perform the operations for processing the next data object using the application in block 314 as described above with reference to FIG. 3A. The processor executing the application may then continue with the operations in determination block 302.

[0076] FIG. 3C illustrates an exemplary graph 370 of values of a function referred to as $f(t)$ (e.g., FIG. 3C) plotting time (t) against a processing quality that may be utilized by a processor executing an application configured with a fixed-time software configuration, such as described above in FIG. 3A. In particular, the graph 370 indicates a relationship wherein the processing quality of the processor executing the application may increase with the amount of time allotted to processing a certain workload, such as summarizing a set of media files (e.g., photos, videos, audio samples, etc.). For example, when the processor executing the application may perform a workload within a first time 372 (e.g., a small period of time, seconds, minutes, etc.), the processing quality may be a first level 374 (e.g., a low quality processing). However, when the processor executing the application may perform the same workload within a second time 376 (e.g., a larger period of time, seconds, minutes, etc.), the processing quality may be a second level 378 (e.g., a higher quality processing). As described above, the time that the processor executing the application may use, and thus the processing quality, may be dependent upon various factors, such as remaining battery power and/or other contextual operating conditions (e.g., time before next scheduled task workload, etc.).

[0077] FIG. 4 illustrates an embodiment method 400 for a mobile device processor executing an application configured with a software configuration to process a portion of a task shared between a plurality of nearby collaborating devices. The method 400 may be similar to the method 200 described above, except that the method 400 may include operations for the processor executing the application to participate in a task with a workload divided between the various applications executing on the nearby collaborating devices. In particular, the processor executing the application may perform operations for completing a portion of the shared task that is allocated to the processor executing the application based at least on the software configuration of the application. For example, a master device may instruct a first processor executing a first application on a first mobile device to perform operations to process a first, large set of data when the first application is configured with a high-performance software configuration suitable for the large data set and a second processor executing a second application on a second mobile device to process a second, small set of data when the second application is configured with an economic software configuration suitable for only a small data set. In this way, the processor executing

the application may be configured to contribute to shared workloads in a manner appropriate for the operating conditions of the mobile device.

[0078] The operations in blocks 202-206 may be similar to the operations in like numbered blocks described above with reference to FIG. 2. In block 402, the processor executing the application may obtain a first portion of a task shared between a plurality of nearby collaborating devices based on the application being configured with the first software configuration. For example, via a peer-to-peer connection (e.g., via a short-range wireless connection, via a Wi-Fi LAN, etc.), the processor executing the application may receive a selection of data objects to be analyzed. The first portion of the task may be indicated by instructions, code, scripts, commands, and/or other information transmitted to the mobile device by a master device. For example, a controlling member of the plurality of nearby collaborating devices may determine how to divide the workload of the shared task as described below, and may transmit messages to the various nearby collaborating devices, including the mobile device, that indicate which portions of the workload are assigned to each nearby collaborating device. In some embodiments, the mobile device itself may be the master device for the shared task, and thus may generate instructions for dividing the portions of the shared task to the plurality of nearby collaborating devices, including itself. For example, the processor executing the application may obtain the first portion of the shared task based on its own identification of the division of workload between the various nearby collaborating devices based on their individual software configurations.

[0079] In some embodiments, the first portion of the task may include processing data that is locally stored on the mobile device and/or data that is received from another device. For example, the processor executing the application may be assigned to process a first portion of the task that includes a first set of digital photos already stored on the mobile device and a second set of digital photos transmitted by a second mobile device. In some embodiments, the processor executing the application may send a message to one or more of the plurality of nearby collaborating devices requesting the first portion of the shared task. For example, the processor executing the application may cause a message to be transmitted (e.g., broadcast via short-range wireless signals) that indicates the processor executing the application is ready to receive subsequent transmissions that include data to be processed corresponding to the first portion of the shared task.

[0080] In block 404, the processor executing the application may perform the first portion of the task using the application configured with the first software configuration. For example, when configured with an economic software configuration as described above, the processor executing the application may process the first portion when the battery power and/or device temperature of the mobile device are within the predefined thresholds defined by the first software configuration. As another example, the processor executing the application may process the first portion as fast as possible when the first software configuration is a high-performance software configuration or alternatively may process the first portion within a certain time period when the software configuration defines a fixed-time.

[0081] FIG. 5 illustrates an embodiment method 500 for an application executing on a mobile device to exchange messages with nearby devices when participating in a task shared

between a plurality of nearby collaborating devices. The method 501 is similar to the method 400 described above, except the method 500 includes additional operations for the processor executing the application to receive various messages from nearby collaborating devices related to the shared task, such as messages indicating current software configurations of applications. For example, the processor executing the application may exchange messages with nearby collaborating devices that may include a master device configured to control or manage the processing of a photo summarization task.

[0082] The operations in blocks 202-206 may be similar to the operations in like-numbered blocks described above with reference to FIG. 2. As described above, by default, the processor executing the application configured with its software configuration may be configured to perform operations and processes data stored locally on the mobile device. For example, the application may simply be executed by the mobile device to summarize digital media files stored in a local media library or media folder. However, the processor executing the application may also be configured to monitor the messages that indicate the presence of tasks that may be shared between nearby collaborating devices, and in response to receiving such messages, may perform operations for contributing to the shared task when appropriate. Accordingly, in block 502, the processor executing the application may receive a message from a nearby device (e.g., a master device, a non-master mobile device, etc.) indicating a task is available that can be shared between a plurality of nearby collaborating devices. For example, the processor executing the application may receive the message indicating a photo summarization task is available to all nearby devices executing a certain camera application capable of causing devices to perform photo summarization. The message indicating the task may include requirements information for participating in the shared tasks, such as software configuration the application must be configured with in order to participate in the shared task. For example, the message may include instructions for the processor executing the application to configure the application with a high performance software configuration. As another example, the message may indicate the shared task may require a certain amount of time or battery power to complete. In some embodiments, the processor executing the application may configure the application with a second software configuration in response to receiving the message from the nearby device. For example, the message may include instructions that may cause the processor executing the application to deactivate a current high performance software configuration of the application and activate a fixed-time software configuration in order for the processor executing the application to process data over a certain period of time, or vice versa.

[0083] In block 504, the processor executing the application may transmit an acceptance message to one or more nearby devices indicating the mobile device may participate in the shared task. In particular, the acceptance message may be transmitted for receipt by a designated master device associated with the shared task. The acceptance message may also include the software configuration that the application is currently configured with (i.e., the first software configuration). For example, the acceptance message may include a code or other information that represents the availability or willingness of the processor executing the application to participate in the shared task, as well as data (e.g., metadata) indicating

the operating conditions of the mobile device (e.g., battery power, scheduled routines, etc.), the possible software configurations available to the application, and the software configuration with which the application is currently configured to operate. The information within the acceptance message may be used by recipient devices (e.g., a master device) to determine additional data related to the shared task, such as individual instructions for the processor executing the application and data sets to be processed by the processor executing the application. For example, a master device receiving the acceptance message may determine that the processor executing the application may process a large amount of data of the shared task based on the application being configured with a high-performance software configuration, its large amount of battery power, its processor capabilities, and/or its high bandwidth communication protocol.

[0084] In some embodiments, the acceptance message may be broadcast to all nearby devices or alternatively may be transmitted to a particular recipient device or recipient address (e.g., media access control (MAC) address, IP address, etc.) as indicated in the message received with the operations in block 502. Further the acceptance message may be transmitted using a communication protocol and/or format indicated in the message received with the operations in block 502.

[0085] In some embodiments, the mobile device may be designated to operate as a master device with regard to controlling and distributing shared tasks to nearby collaborating devices. Accordingly, in optional block 506, the processor executing the application may broadcast or otherwise transmit a message indicating a task can be shared between the plurality of nearby collaborating devices. Such a message may be similar to the message described above with reference to block 502. Further, in optional block 508, the processor executing the application may receive acceptance messages (e.g., indicating acceptance of one or more nearby devices to participate in the shared task broadcast) or otherwise transmitted by the mobile device, as well as current software configurations of the one or more nearby collaborating devices. When the mobile device is configured function as a master device for a given shared task, the processor executing the application may use information from such received acceptance messages to determine how to divide a workload of the shared task broadcast (or otherwise transmitted) by the application. For example, based on the software configurations reported in the received acceptance messages, the processor executing the application may determine the various data sets for each of the responding nearby collaborating devices to process as part of the shared task. The received acceptance messages may be similar to the acceptance message described above with reference to block 504. The processor executing the application may continue with the operations in blocks 402-404 as described above with reference to FIG. 4.

[0086] FIG. 6 illustrates an embodiment method 600 for a mobile device processor executing an application to receive a command signal for adjusting software configurations when processing portions of a task shared between a plurality of nearby collaborating devices. The method 600 is similar to the method 400 described above, except the method 600 includes additional operations for the processor executing the application to change the software configuration of the application based on instructions from a nearby device, such as a master device configured to control the processing of the shared task. For example, in response to receiving a com-

and, the processor executing the application may switch the application from a high-performance software configuration to an economic software configuration in order to accommodate a diminished battery in the mobile device.

[0087] The operations in blocks 202-206 and 402-404 may be similar to the operations in like numbered blocks described above with reference to FIG. 2 and FIG. 4, respectively.

[0088] In optional determination block 601, the processor executing the application may determine whether there is a change in the operating conditions of the mobile device, such as in response to conducting periodic or continuous polling operations as described above. For example, the mobile device processor via the application may obtain current GPS coordinates, available battery power, device temperature, etc., and compare it to previously obtained data to identify changes in values that exceed preselected threshold values (e.g., battery threshold, etc.). In response to determining that there is a change in operating conditions (i.e., optional determination block 601 “Yes”), the processor executing the application may transmit a message indicating the change in operating conditions. Such a message may be broadcast or directed transmission receivable by a master device organizing a shared workload and that may trigger the master device to reconfigure the participation of the mobile device and/or other collaborating devices as well as redistribute tasks based on the reconfigured participations. For example, based on the message indicating the battery of the mobile device is now fully charged, the master device may increase the workload assigned to the mobile device.

[0089] In response to determining that there is a change in operating conditions (i.e., optional determination block 601 “No”), or in response to the mobile device processor performing the operations in optional block 602, the processor executing the application may receive a command signal (or message) related to the task shared between the plurality of nearby collaborating devices in block 603. The command signal may be sent from a master device (e.g., via an application executing on a master device that controls the shared task), and may include a code, script, instructions, and/or other information that the processor executing the application may process or otherwise execute. For example, the command signal may be a wireless signal reporting a bit or code instructing the processor executing the application to perform an action, such as a suspend operation or an execute operation. In some embodiments, the command signal may include instructions that cause the processor to configure the application with a certain software configuration. In other words, the command signal may instruct a change of an active software configuration based on the operating conditions of the mobile device. For example, the command signal may cause the processor executing the application to deactivate a current software configuration (e.g., an economic software configuration) and activate a high-performance software configuration. In some embodiments, the command signal may include parameter values for an already configured software configuration of the application, such as new threshold values to be used with an already activated economic software configuration. For example, the command signal may include a new device temperature threshold value that may indicate when the processor executing the application configured with its economic software configuration may suspend operations related to the shared task.

[0090] In various embodiments, the command signal may be received before or after the processor executing the appli-

cation begins to process any portion of the shared task. For example, the command signal may be received prior to the processor executing the application has begun to process digital media files in a collaborative summarization task, causing the processor executing the application to change the application from an economic software configuration to a high-performance software configuration prior to processing the first portion of the shared task. As another example, the command signal may be received by the processor executing the application in response to completing the processing of a first portion of the shared task, causing the processor executing the application to reconfigure the application with a different software configuration so that the processor may be configured to perform additional portions of the shared task.

[0091] In some embodiments, the command signal may be sent to cause the processor executing the application to reconfigure the application in order to match the software configuration of applications running on the nearby collaborating devices. For example, when executing a shared task, each application on each nearby collaborating device may be required to be configured with the same software configuration that enables a particular communication protocol (e.g., Bluetooth, etc.) before the shared task may commence execution.

[0092] In optional block 604, the processor executing the application may prompt a user confirmation (or authorization) of an adjustment of software configuration for the application based on the received command signal. In other words, when the received command signal indicates that the application should be configured with a software configuration different from its current software configuration, the mobile device may render a message requesting the user of the mobile device to confirm such a change. For example, the processor executing the application may render a message indicating that the command signal requests the application be configured with a high-performance software configuration like other devices within the nearby collaborating devices. Such prompting may include providing the user with graphical user interface buttons for providing user inputs that select to confirm or reject the change of software configuration based on the received command signal. In some embodiments, the prompting may include presenting information indicating the effects of changing the software configuration of the application, such as reduced user experience and/or increases in device resource use (e.g., increased battery power use, increased device component heat, denervation activation of device components such as transceivers, etc.). For example, the mobile device may render a warning (e.g., a pop-up box) informing the user of the mobile device that executing the application configured with the second software configuration may be more expensive than executing the application with the first software configuration. In some embodiments, user inputs may not be required to confirm or authorize the change of software configuration for the application, and instead, an automated, context-aware engine (e.g., a routine, service, etc.) may evaluate the command signal to determine whether the application may be re-configured with a software configuration different from its current or default software configuration.

[0093] In block 606, the processor executing the application may configure the application to operate with a second software configuration in response to receiving the command signal, such as by setting various operating parameters (e.g., thresholds, processing quality, processing speed, etc.) based

on stored data of the second software configuration. In other words, the processor executing the application may deactivate the first software configuration and may activate the second software configuration. The operations of block 606 may be similar to those described above with reference to block 206. In some embodiments, switching software configurations may cause the processor executing the application to utilize different communication protocols and/or transceivers for processing the shared task. For example, when a more efficient communication protocol is identified by the master device orchestrating the various nearby collaborating devices regarding the shared task, the command signal may include instructions for the processor to cause the application to enter a software configuration that is associated with that efficient communication protocol. In this way, the processor executing the application may be adjusted to better operate in combination with the other nearby collaborating devices.

[0094] In optional block 608, the processor executing the application may transmit a message indicating the change of software configuration of the application. For example, in order to inform the configuration change of the application, the mobile device may transmit update messages to each of the nearby collaborating devices. In some embodiments, the processor executing the application may pause or continue processing of the first and/or second portion of the shared task, initiate the processing of the portions of the task, and/or put the portions of the shared task on a waiting list to be performed at a subsequent time.

[0095] In optional block 610, the processor executing the application may obtain a second portion of the task shared between the plurality of nearby collaborating devices based on the application being configured to operate with the second software configuration. For example, the command signal may have been sent by a master device in response to the processor executing the application completing the processing of the first portion of the data using the application configured with the first software configuration, and thus the processor executing the application may be directed to gather a new data set for processing. The second portion of the shared task may be larger (e.g., more files to summarize, etc.), smaller, more complex (e.g., larger media files to summarize, etc.), and/or more simplified than the first portion of the shared task. Further, the second portion may be particularly well-suited for processing by the processor executing the application configured with the second software configuration. For example, the processor executing the application may obtain a new portion of the shared task that includes time-sensitive data that may be appropriately processed using the application configured with a fixed-time software configuration.

[0096] Similar to the operations in block 404, in optional block 612, the processor executing the application may perform the second portion of the task using the application configured with the second software configuration. For example, the processor executing the application may perform media summarization operations with a high-performance software configuration on a second set of digital photos.

[0097] In some embodiments, the processor executing the application may only begin to process any portion of the data of the shared task in response to receiving the command signal and/or configuring the application to operate with the second software configuration. In other words, the shared task may only be started by the processor executing the appli-

cation when the application has been configured according to the master device with regard to the shared task.

[0098] FIGS. 7A-C, FIGS. 8A-BB relate to a particular application of embodiment software configurations in a collaborative computing environment for processing media stored on various nearby mobile devices. For example, during a day trip to an amusement park, each member of a family may carry their own smartphone for taking digital media of their individual experiences. A mom may take photos of a roller coaster with a first smartphone, a son may take a video of a concession stand with a second smartphone, and a dad may take photos of a person standing in a parking lot with a third smartphone. The smartphones used by the various family members may be configured with software for summarizing media files (i.e., summarization applications). But may not have an adequate means (e.g., a good data plan) for sharing digital media files during the day trip at the amusement park. Accordingly, the media files taken by each family member may reside exclusively on their individual smartphones without being stored for use with the summarization applications executing on the various devices.

[0099] However, at night after leaving the amusement park, the family may congregate in a same hotel room. Each of the smartphones may be placed within proximity to the same table, such as for charging or merely to keep them in a safe location. With an adequately robust peer-to-peer sharing on the table, such as via WiFi or short-range signaling (e.g., Bluetooth, etc.), the smartphones may determine that the various media files may be collaboratively processed by the summarization applications executing on each of the smartphones. As each of the smartphones may have different operating conditions (e.g., active transceivers, battery power, processor types, scheduled routines, etc.), their individual summarization applications may each be configured with different software configurations. For example, a first summarization application on the first smartphone may be configured with an "economic" software configuration, a second summarization application on the second smartphone may be configured with an "high-performance" software configuration, and a third summarization application on the third smartphone may be configured with an "fixed-time" software configuration. The smartphones may communicate with each other via their respective summarization applications, and may exchange media files (e.g., photos, etc.) so that all the media taken by the various mobile devices during the day at the amusement park are consolidated, organized, and have a short summary. Each of the summarization applications executing on the various smartphones may contribute different amounts of work to the summarization of the entire group of media files based on its individual software configuration. In this way, each of the smartphones (and their users) may benefit from a complete summarization of the media files, however the workload for each smartphone may only be set that individual device may handle based on its current operating conditions.

[0100] FIG. 7A illustrates an exemplary scenario in which a plurality of mobile devices 102, 112, 122 may be configured to utilize applications configured with various software configurations in order to collaboratively perform a shared task (e.g., media summarization). A first mobile device 102 (referred to in FIGS. 7A-C as "Device 1"; may include a first display 702, of some of its operating conditions, such as available transceivers (e.g., Bluetooth 104, Energy 106, or "BLU1"), WiFi, available battery power (e.g., 50%), and a

device temperature (e.g., "moderate"). The first mobile device 102 may execute a first application 704 (e.g., a media summarization app) that is configured with a first software configuration 705 (referred to as an "economic"). As described above, such an economic software configuration may be automatically configured based on the operating conditions of the first mobile device 102 as described above. For example, the first application 704 may be configured with the economic software configuration based on a moderate device temperature reading along with a half-depleted battery. The first application 704 may also be associated with a first set of media files 706 (e.g., pictures A-B) that are stored locally on the first mobile device 102.

[0101] A second mobile device 112 referred to in FIGS. 7A-C as "Device 2" may include a second display 712 of some of its operating conditions, such as available transceivers (e.g., BT/1, WiFi), available battery power (e.g., 100%), and a device temperature (e.g., "low"). The second mobile device 112 may execute a second application 714 (e.g., the media summarization app) that is configured with a second software configuration 715 (referred to as "high-performance"). The second software configuration 715 may be automatically configured based on the operating conditions of the second mobile device 112 as described above. For example, the second application 714 may be configured with the high-performance software configuration based on a low device temperature reading and/or a full battery. The second application 714 may also be associated with a second set of media files 716 (e.g., picture C) that are stored locally on the second mobile device 112.

[0102] A third mobile device 122 referred to in FIGS. 7A-C as "Device 3" may include a third display 722 of some of its operating conditions, such as an available transceiver (e.g., BT/1), a power reading (e.g., plugged into power source), and a device temperature (e.g., "high"). The third mobile device 122 may execute a third application 724 (e.g., the media summarization app) that is configured with a third software configuration 725 (referred to as "power-saving"). The third software configuration 725 may be automatically configured based on the operating conditions of the third mobile device 122 as described above. For example, the third application 724 may be configured with the power-saving software configuration based on being plugged into the power source and/or having a high device temperature reading. The third application 724 may also be associated with a third set of media files 726 (e.g., pictures D-F) that are stored locally on the third mobile device 122. In some embodiments, the various software configurations may be set in the mobile devices 102, 112, 122 based on user inputs as described below.

[0103] The mobile devices 102, 112, 122 may exchange transmissions via wireless connections 700, 710, 720. For example, the first mobile device 102 and the second mobile device 112 may exchange Bluetooth packets via the first wireless connection 700, the first mobile device 102 and the third mobile device 122 may exchange Bluetooth packets via the second wireless connection 710, and the second mobile device 112 and the third mobile device 122 may exchange Bluetooth packets via the third wireless connection 720. In various embodiments, the mobile devices 102, 112, 122 may utilize other connectivities for peer-in-peer communications, such as via communications via a router associated with a local area network as described above with reference to FIG. 1.

[0104] The exchanged transmissions may include various messages, such as messages indicating the current software configuration of each of the applications 704, 714, 724, as well as command signals and/or other instructional messages that may indicate how the workload of a shared task may be divided between the mobile devices 102, 112, 122. The transmissions via the wireless connections 700, 710, 720 may also include data to be processed via the applications 704, 714, 724. In particular, based on instructions from a "master" device configured to organize the operations of the mobile devices 102, 112, 122 with regard to the shared task based on their respective software configurations 705, 715, 725, the mobile devices 102, 112, 122 may deliver various media files to one another for processing. For example, based on instruction messages, the first application 704 may cause a media file (e.g., picture B) to be sent to the second mobile device 112 or the third mobile device 122 for processing by their respective applications 714, 724 (and processors).

[0105] In various embodiments, the master device may be any one of the mobile devices 102, 112, 122. Alternatively, the master device may be determined based on a current software configuration or operating conditions, such as the mobile device having the highest available power (or, battery power), the least burdened processor, the most locally stored data, the best networking conditions (e.g., high bandwidth, etc.), and/or a user input selecting the master device.

[0106] FIGS. 714-7C show tables 750, 760 that illustrate exemplary divisions of a workload or a task shared between nearby collaborating devices. As described above, each mobile device participating in the shared task may be assigned or otherwise instructed to perform a portion of the shared task based on their respective applications' software configurations, such as configurations automatically set by their processors executing their applications based on the operating conditions of the mobile devices. The tables 750, 760 illustrate how the media files that are locally stored on the mobile devices 102, 112, 122 may be divided for summarization by processors executing the applications 704, 714, 724 based on their software configurations 705, 715, 725. In various embodiments, such divisions of the workload may be identified and communicated by a master device as described above.

[0107] The table 750 in FIG. 7B illustrates a division of the shared media summarization task wherein the various media files stored on the mobile devices 102, 112, 122 may be processed by at most one device executing one of the applications 704, 714, 724. The table 750 includes a first row 752 associated with the first mobile device 102 illustrated in FIG. 7A. Based on its first software configuration 705, the first application 704 of the first mobile device 102 may be assigned two media files (e.g., pictures A and B). As described above with reference to FIG. 7A, the first mobile device 102 may also store the assigned media files, and so may not need to utilize the wireless connection 700, 710 to receive the media files. A second row 754 indicates that, based on the second software configuration 715, the second application 714 executing on the second mobile device 112 may be assigned three media files (e.g., pictures C-F). As the second mobile device 112 may only locally store one of these three media files (e.g., only picture C is locally stored), the second mobile device 112 via the second application 714 may need to request the other assigned media files from the other mobile devices 102, 122 collaborating on the shared media summarization task. For example, the second mobile device 112 via

the second application 714 may transmit a message to the third mobile device 122 requesting the delivery of the pictures 134. The second application 714 may be assigned a larger number of media files to process than the first application 704 or the third application 724, as the second application 714 is configured with the high-performance software configuration and thus, when executed by the processor of the second mobile device 112, may be capable of handling more workload without risking degraded performance. A third row 756 indicates that, based on the third software configuration 725, the third application 724 executing on the third mobile device 122 may be assigned one media file (e.g., picture 11). As the third mobile device 122 stores more than its assigned media files (e.g., pictures 134), the third application 724 executing on the third mobile device 122 may receive requests to deliver the unassigned media files (e.g., pictures 134) from another application that is assigned those media files (e.g., the second application 714).

[0110] The table 760 in FIG. 7C is similar to the table 750 in FIG. 7B, except that the various media files stored on the mobile devices 102, 112, 122 may be assigned for processing by one or more of the mobile devices 102, 112, 122 via their respective applications 704, 714, 724. In this way, the workload of the shared task may be distributed in an overlapping manner. Such a scheme may balance performance variances by the mobile devices 102, 112, 122, thereby minimizing potential impacts in efficiency of the shared task caused by unexpected changes in the speed and/or quality of processing using the applications 704, 714, 724. Thus, a first row 762 may indicate that the first application 704 may be assigned three media files (e.g., pictures A-C); based on its first software configuration 705, a second row 764 may indicate that the second application 714 may be assigned five media files (e.g., pictures 1-5) based on its second software configuration 715, and a third row 766 may indicate that the third application 724 may be assigned one media file (e.g., picture 1) based on its third software configuration 725. Based on its first software configuration 705, the first mobile device 102 executing the first application 704 may be capable of processing one additional file (e.g., picture 1) overlapping with the assigned media files of the second application 714 without being assigned fewer media files than in the division shown in FIG. 7B. Such an additional file may be received from the second application 714 via a transmission between the first mobile device 102 and the second mobile device 112 as described above. Further, as the second application 714 is configured with a high-performance software configuration (i.e., the second 715), it may be assigned multiple media files (e.g., pictures B and J) overlapping with the first application 704 and the third application 724. The second mobile device 112 executing the second application 714 may receive various files from the first and third mobile device 102, 122 via transmissions as described above. Based on its proven average software configuration, the third application 724 may not be assigned any additional media files.

[0111] FIGS. 8A-B illustrate embodiment methods 800, 850 for a mobile device processor executing an application to perform a portion of a task of summarizing media stored between a plurality of nearby collaborating devices. The methods 800, 850 may be similar to the methods described above, but may include operations specific to a use case of summarizing media stored on various devices within proximity of one another.

[0110] FIG. 8A illustrates an embodiment method 800 for such a media summarization shared task. The operations in blocks 202-216 may be similar to the operations in like numbered blocks described above with reference to FIG. 2. In block 802, the processor executing the application may generate (or collect) metadata of media files stored locally on the mobile device. For example, the processor executing the application may generate metadata indicating the number, file size, runtime, type, timestamps, geo-locating data, and other information of various media files, such as digital photos and videos. In block 804, the processor executing the application may transmit, to a master device, the generated metadata. In block 806, the processor executing the application may transmit to the master device data indicating the current software configuration of the application (i.e., the first software configuration). In some embodiments, the processor executing the application may transmit information indicating some or all of the obtained operating conditions as well as the current software configuration of the application. For example, the processor executing the application may transmit to the master device data that indicates the current battery power of the mobile device.

[0111] In some embodiments, the processor executing the application may cause the mobile device to broadcast or otherwise transmit the information transmitted in the blocks 804-806 so that an undisclosed or not already identified master device within proximity of the mobile device may receive the data. For example, the processor executing the application may periodically cause the mobile device to broadcast useful data that may or may not be used by nearby collaborating devices to initiate a shared task. In other embodiments, the master device may be predetermined.

[0112] In block 808, the processor executing the application may receive from the master device instructions for participating in a media summarization task shared by nearby collaborating devices. Such instructions may include commands, codes, scripts, and/or other information that indicate how the processor executing the application may distribute and process media files as part of the media summarization task. For example, the instructions may include the identifiers of a set of digital photograph files (e.g., jpgs, gds, bmps, etc.) the processor executing the application should analyze for content, authorship, represented persons, places, and/or things. In some embodiments, the instructions may indicate the algorithms, such as facial recognition processing techniques, modules, and/or filtering that may be used by the processor executing the application to process the media files.

[0113] In some embodiments, the instructions may also include information for the processor executing the application to distribute locally stored media files (or data related to the media files) to other nearby collaborating devices for processing. For example, the instructions may include the identifiers of a set of photo or video files that may be transmitted via a WiFi connection to another mobile device also participating in the media summarization shared task. The instructions may therefore also include the device identifier, network address, and/or communication protocol for the processor executing the application to transfer the media files to the other mobile devices.

[0114] In some embodiments, the instructions may also include information indicating media files that are to be received at the processor executing the application from other nearby collaborating devices. For example, the instructions may indicate a number or range of digital photographs that are

to be transmitted by a certain nearby smartphone using a particular communication protocol (e.g., Bluetooth®). [0115] In some embodiments, the instructions may also include commands that may cause the processor to configure the application with a different (or second) software configuration, such as a software configuration stored by the other nearby collaborating devices or another software configuration deemed appropriate by the master device in view of the software configurations of the other nearby collaborating devices. For example, only a certain percentage of the applications executing on the nearby collaborating devices may need to be configured with a high performance software configuration, and so the processor executing the application may receive instructions to configure the application to use a less expensive software configuration.

[0116] In optional block 810, the processor executing the application may transmit media files to one or more nearby collaborating devices based on the received instructions for distributing the media files. For example, the processor executing the application may cause a set of locally stored video files to be transmitted over a local area network connection to a nearby smartphone for further processing. In some embodiments, the media files themselves may not be transmitted, but instead the processor executing the application may transmit metadata related to the media files.

[0117] In optional block 812, the processor executing the application may receive media files from one or more nearby collaborating devices based on the received instructions. For example, due to the application's current software configuration, the processor executing the application may be capable of processing more media files than other nearby collaborating devices, and so may receive media files via a Bluetooth® connection.

[0118] In block 814, based on the received instructions and using the application configured with the first software configuration, the processor executing the application may process the received media files to generate results information. The results information may be order sorting results, identified items within the media files (e.g., known persons, places, and/or things), and/or statistical or analytical information about the media files and/or the processing of the media files. For example, the results information may indicate the amount of time that the processor executing the application performed analysis routines for each photo within the media files. In some embodiments, the results information may be a sorted listing of the media files or the media files themselves with additional/adjusted data (e.g., added tags, annotations, renamed filenames, etc.).

[0119] In block 816, the processor executing the application may transmit the results information to various nearby collaborating devices, such as the master device. For example, the processor executing the application may transmit results information related to the media files received with the operations in optional block 812 to the nearby collaborating device that initially transmitted those media files to the mobile device.

[0120] FIG. 8F3 illustrates an embodiment method 850 for a mobile device processor executing an application to perform a portion of a task of summarizing media shared between a plurality of nearby collaborating devices. The method 850 is similar to the method 800 described above, except the method 850 may be performed by a mobile device designated as a "master" device configured to control the division of the workload associated with the shared media summarization

task. For example, the processor executing the application may perform operations that include sending messages to nearby collaborating devices that direct the subsequent operations of those nearby collaborating devices with regard to the shared task of summarizing media files.

[0121] The operations in blocks 202-206 and 802 may be similar to the operations in like numbered blocks described above with reference to FIG. 2 and FIG. 8A, respectively. In block 852, the processor executing the application may receive metadata of media located at one or more nearby collaborating devices. For example, the processor executing the application may receive messages from each device participating in the shared task that indicate the number and type of media files that may be summarized via the applications executing on the devices.

[0122] In block 854, the processor executing the application may receive current software configurations from one or more nearby collaborating devices. For example, the processor executing the application may receive messages from each of the nearby collaborating devices indicating whether they are operating with an economic, high-performance, power-saving, and/or other software configuration. In some embodiments, the processor executing the application may also identify operating conditions of the various devices from received messages, such as current battery power, networking conditions, etc.

[0123] In block 856, the processor executing the application may evaluate the received metadata to identify a workload for a task related to the media files that may be shared amongst the one or more nearby collaborating devices (e.g., a summarization task). In other words, the processor executing the application may evaluate information received from the various nearby collaborating devices to determine how many total media files need to be summarized by applications executing on the nearby collaborating devices. The processor executing the application may also determine the types of processing, such as sorting, facial recognition, etc., that may be required to process the various media files related to the summarization task.

[0124] In block 858, the processor executing the application may determine a division of the workload based on the current software configurations of the applications executing on the nearby collaborating devices. For example, the processor executing the application may evaluate the resources and capabilities of the various nearby collaborating devices to determine how to best assign media files to the individual devices. As another example, the processor executing the application may determine how many media files each nearby collaborating device may be assigned in order to complete the summarization task within a certain time period. The processor executing the application may utilize various different objectives and goals when determining the division of the workload, such as maximizing the speed at which the task is completed, minimizing the amount of battery power utilized by the devices in completing the task, and minimizing the amount of file transfers required to complete the task. In some embodiments, the processor executing the application may divide the workload such that nearby collaborating devices with applications configured with high-performance software configurations and/or high amounts of available power may be assigned more media files to process.

[0125] In block 860, the processor executing the application may generate instructions for distributing the workload based on determined division. As described above, the gen-

erated instructions may indicate information that may cause the various nearby collaborating devices participating in the shared task to perform various operations, such as transmuting media files to one another and/or performing processing operations on various media files that may or may not be originally stored on the various devices. In some embodiments, the instructions may also include commands that cause the nearby collaborating devices to adjust or change the current software configurations of their applications, such as commands for adjusting thresholds of software configurations or for directing another application to activate a different software configuration entirely.

[0126] In block 861, the processor executing the application may transmit the generated instructions to the one or more nearby collaborating devices. For example, the processor executing the application may cause the mobile device to transmit the instructions to each of the nearby devices that are eligible to participate in the shared task of summarizing the media file. In various embodiments, the processor executing the application may transmit different instructions for each individual nearby device. For example, the processor executing the application may transmit to a first nearby device a first instructions message indicating a first set of media files to be processed by the first nearby device and may transmit to a second nearby device a second instructions message indicating a second set of media files to be processed by the second nearby device.

[0127] In optional block 810*, the processor executing the application may transmit media files to one or more nearby collaborating devices based on the generated instructions. In optional block 812, the processor executing the application may receive instructions from the one or more nearby collaborating devices based on the generated instructions. In block 814, based on the generated instructions and using the application configured with the first software configuration, the processor executing the application may process media files to generate results information. The operations in blocks 810, 812, and 814 may be similar to those described above with reference to blocks 810, 812, and 814, except that the processor executing the application may utilize the instructions generated locally instead of receiving the instructions from another device. The processor executing the application may then perform the operations in block 816 as described above with reference to 810, 812, 814.

[0128] FIGS. 9A-9C illustrate embodiment interfaces that indicate and/or adjust various software configurations used by an application. The current operating conditions for a mobile device may be obtained and used by processors executing applications to automatically set a software configuration for default software configuration, as described above. However, in some scenarios, a user may desire to manually set or change the default software configuration in order to achieve a different user experience when executing the application at a given time. For example, when an application is configured with an economic performance software configuration that causes the processor executing the application to suspend activity when the mobile device temperature exceeds a certain threshold, the user may want to configure the application with a high-performance software configuration in order to avoid any such suspensions. Such manual adjustments from default software configurations may be beneficial when the user has more information than the processor executing the application may obtain via the operating conditions of the mobile device. For example, when

the user knows he is about to shut the phone off or plug it in right after a task is completed, he may not care about the current low battery power and accordingly may not want to use the economic software configuration that may suspend the task.

[0129] FIG. 9A illustrates a mobile device 901 executing an application 910 (e.g., a media summarization app). The mobile device 901 executing the application 910 may render a message 911 or other information that indicates the default software configuration of the application 910. For example, the message 911 may indicate that based on the current operating conditions of the mobile device 901, the application is configured with an economic software configuration that may cause the processor of the mobile device 901 to suspend activities of the application 910 in response to detecting low battery power, high device temperature, and/or other predefined conditions.

[0130] In some embodiments, the default software configuration may be set by the user or an application provider or developer as a preferred or default configuration for the application. The mobile device 901 executing the application 910 may display a graphical user interface (GUI) element 912 for adjusting the software configuration of the application 910. For example, the element 912 may be a button that, when selected (e.g., touched by the user's finger or other input device, such as a stylus, etc.), may cause the processor executing the application 910 to switch the application 910 from the default configuration of the economic software configuration to a high-performance software configuration.

[0131] FIG. 9B illustrates a warning message 920 associated with the application 910 that is rendered by the mobile device 901 in response to the user's input to configure the application 910 with a high-performance software configuration. The warning message 920 may prompt the user to confirm the adjustment of the software configuration of the application 910, and may further indicate the effects or repercussions of such a change. For example, the warning message 920 may indicate that switching from a default economic software configuration to a high-performance software configuration may cause the mobile device 901 processor executing the application 910 to require more energy and to generate more heat. The warning message 920 may indicate various other information, such as the impact on a battery, estimated effect on other applications running on the mobile device 901 (e.g., other applications may slow down or be suspended, etc.), and effects on communications (e.g., degraded or improved quality of service via wireless transmission, etc.).

[0132] The application 910 may include a first GUI element 922 (e.g., button) for the user to confirm the change of the software configuration and a second GUI element 924 (e.g., button) for the user to reject the change of the software configuration. For example, the user may select the first GUI element 922 (e.g., touch with his/her finger) in order to cause the mobile device 901 processor executing the application 910 to configure the application 910 with the high-performance software configuration instead of the default economic software configuration. In some embodiments, the warning message 920 may be a pop-up box, a dialog box, modal or modeless window, or other graphical representation within the application 910.

[0133] FIG. 9C illustrates a new message 931 indicating the adjusted software configuration of the application 910 executing on the mobile device 901. For example, in response to the user confirming the switch from an economic software

configuration to a high-performance software configuration as shown in FIG. 9A, the new message 931 may indicate the current configuration of the application 910 is now the high-performance software configuration. Further, the application 910 may include a GUI element 932 (e.g., button) for the user to change the current software configuration to another configuration (such as a standard, default software configuration). For example, the GUI element 932 may be pressed by the user in order to cause the mobile device 910 processor to cause the application 910 to revert to the economic software configuration from the high-performance application configuration. In some embodiments, the new message 931 may be a pop-up box, a dialog box, modal or modeless window, or other graphical representation within the application 910.

[0134] FIG. 9D illustrates an embodiment method 950 for a mobile device processor executing an application to configure the application with a second software configuration in response to user inputs. The method 950 is similar to the method 210 described above with reference to FIG. 2, except that the method 950 includes operations for receiving user inputs that may cause an application to be configured with a software configuration chosen by the user as opposed to automatically chosen by the processor executing the application. For example, instead of utilizing a default software configuration that was automatically identified as appropriate for the application based on the current battery power and device temperature of the mobile device, the user may switch the application to a high-performance software configuration in order to cause the processor executing the application to quickly process a workload, regardless of the implications to the battery and/or overall device performance.

[0135] The operations in blocks 202-208 may be similar to the operations in like numbered blocks described above with reference to FIG. 2. In block 952, the processor executing the application may receive a user input selecting a second software configuration, such as via a graphical user interface (GUI), as shown in FIG. 9A above. For example, the mobile device may detect a touch input on a touchscreen of the mobile device, the touch input corresponding to a GUI button for switching software configurations for the application. The second software configuration may be an alternative configuration to the first software configuration, and thus may or may not provide better or worse performance for the mobile device and/or the application. In some embodiments, the received user input may indicate operating parameters (e.g., thresholds) or other information relevant to a software configuration. For example, when a background or power-savings software configuration is selected via the user inputs, the user may also indicate a time of day (e.g., late at night, etc.), a power condition (e.g., plugged in) or a duration for running the application.

[0136] As described above, in some embodiments, regardless of user inputs selecting software configurations, the processor executing the application itself may be configured to side-step such selections based on its awareness of the operating conditions. For example, when the mobile device is charging at night, the processor executing the application may determine the application may be configured to operate with a high-performance software configuration as selected by a user input only until the device temperature reaches a dangerous level (e.g., hot enough to burn the table on which the mobile device is resting, etc.). As another example, when the processor executing the application determines the battery may be depleted and there is no usage history stored that

indicates the user typically charges the mobile device at a certain time of day, the processor executing the application may configure itself with an economic software configuration instead of a user-chosen high-performance software configuration.

[0137] In other embodiments, the user input may indicate a priority of possible software configurations that may be used by the application. For example, for each application and/or task that may be performed by each application executing on the processor of the mobile device, the user may choose the priority of each of the plurality of software configurations for the application. In this way, when the processor executing the application determines it may change its software configuration based on obtained operating conditions, subsequent user inputs, and/or when there are multiple concurrent applications executing on the mobile device, the processor executing the application may use the prioritized list of software configurations to select a next software configuration. In some embodiments, such a priority list of the software configurations may be further refined by the processor executing the application based on stored data indicating the last time a particular type of tasks/task identity was executed by the processor executing the application and/or the least amount of time this job may require.

[0138] In some embodiments, the second software configuration may cause greater device resource consumption (e.g., power, communication interfaces, bandwidth, etc.) that may deprive other applications of resources and thus impede their performance. Accordingly, in optional block 954, the processor executing the application may display a warning message when the second software configuration may utilize more mobile device resources than the first software configuration. For example, as illustrated in FIG. 9A above, the mobile device may render a pop-up or dialog box that indicates performance of the processor executing the application and/or the mobile device may change as a result of the switch from the first to the second software configuration. Similar to the operations described above with reference to block 206, in block 956, the processor executing the application may configure the application with the second software configuration. For example, the processor executing the application may utilize new operating parameters, new thresholds used for checking operating conditions, and/or access device resources differently. In block 958, the processor executing the application may process data using the application configured with the second software configuration.

[0139] Various forms of mobile computing devices, including smartphones, tablets, and laptop computers, may be used to implement the various embodiments. Such mobile computing devices may typically include the components illustrated in FIG. 10 which illustrates an exemplary smartphone-type/mobile device 1000. In various embodiments, the mobile device 1000 may include a processor 1001 coupled to a touchscreen controller 1004 and an internal memory 1002. The processor 1001 may be one or more microprocessors designated for general or specific processing tasks. The internal memory 1002 may be volatile or non-volatile memory, and may also be secure and/or encrypted memory, or non-secure and/or unencrypted memory, or any combination thereof. The touchscreen controller 1004 and the processor 1001 may also be coupled to a touchscreen panel 1011, such as a resistive-sensing touchscreen, capacitive-sensing touchscreen, infrared sensing touchscreen, etc.

[0140] The mobile device 1000 may have one or more transceiver signal transceivers 1008 (e.g., Permitr®, Bluetooth®, Zigbee®, Wi-Fi, RF transceiver, etc.) and antennae 1010, for sending and receiving, coupled to each other and/or to the processor 1001. The transceivers 1008 and antennae 1010 may be used with the above-mentioned circuitry to implement the various wireless transmission protocol stacks and interfaces. The mobile device 1000 may include cellular network wireless modem chip 1016 that enables communication via a cellular network and is coupled to the processor 1001. The mobile device 1000 may include a peripheral computing device connection interface 1018 coupled to the processor 1001. The peripheral computing device connection interface 1018 may be similarly configured to accept one type of connection, or multiply configured to accept various types of physical and communication connections, common or proprietary, such as USB, FireWire, Thunderbolt, or PCIe. The peripheral computing device connection interface 1018 may also be coupled to a similarly configured peripheral computing device connection port (not shown).

[0141] The mobile device 1000 may also include speakers 1014 for providing audio outputs. The mobile device 1000 may also include a housing 1020, constructed of a plastic, metal, or a combination of materials, for containing all or some of the components discussed herein. The mobile device 1000 may include a power source 1022 coupled to the processor 1001, such as a disposable or rechargeable battery. The power source 1022 may also be coupled to the peripheral computing device connection interface 1018 to receive a charging current from a source external to the mobile device 1000. Additionally, the mobile device 1000 may include a GPS receiver chip 1015 coupled to the processor 1001.

[0142] The processor 1001 may be any programmable microprocessor, microcomputer or multiple processor chip or chips that can be configured by software instructions (applications) to perform a variety of functions, including the functions of the various embodiments described above. In the various computing devices, multiple processors may be provided, such as one processor dedicated to wireless communication functions and one processor dedicated to running other applications. Typically, software applications may be stored in the internal memory 1002 before they are accessed and loaded into the processor 1001. The processor 1001 may include internal memory sufficient to store the application software instructions. In many computing devices the internal memory may be a volatile or nonvolatile memory, such as flash memory, or a mixture of both. For the purposes of this description, a general reference to memory refers to memory accessible by the processor 1001 including internal memory or renewable memory plugged into the various computing devices and memory within the processor 1001.

[0143] The foregoing method descriptions and the process flow diagrams are provided merely as illustrative examples and are not intended to require or imply that the steps of the various embodiments must be performed in the order presented. As will be appreciated by one of skill in the art the order of steps in the foregoing embodiments may be performed in any order. Words such as "hereafter," "then," "next," etc. are not intended to limit the order of the steps; these words are simply used to guide the reader through the description of the methods. Further, any reference to claim elements in the singular, for example, using the articles "a," "an," "the" is not to be construed as limiting the element to the singular.

[0145] The various illustrative logical blocks, modules, circuits, and algorithm steps described in connection with the embodiments disclosed herein may be implemented as electronic hardware, computer software, or combinations of both. To clearly illustrate this interchangeability of hardware and software, various illustrative components, blocks, modules, circuits, and steps have been described above generally in terms of their functionality. Whether such functionality is implemented as hardware or software depends upon the particular application and design constraints imposed on the overall system. Skilled artisans may implement the described functionality in varying ways for each particular application, but such implementation decisions should not be interpreted as a departure from the scope of the present invention.

[0146] The hardware used to implement the various illustrative logics, logical blocks, modules, and circuits described in connection with the embodiments disclosed herein may be implemented or performed with a general purpose processor, a digital signal processor (DSP), an application specific integrated circuit (ASIC), a field programmable gate array (FPGA), or other programmable logic computing device, discrete gate or transistor logic, discrete hardware components, or any combination thereof designed to perform the functions described herein. A general-purpose processor may be a microprocessor, but in the alternative, the processor may be any conventional processor, controller, microcontroller, or state machine. A processor may also be implemented as a combination of computing devices, e.g., a combination of a DSP and a microprocessor, a plurality of microprocessors, one or more microprocessors in conjunction with a DSP core, or any other such configuration. Alternatively, some steps or methods may be performed by circuitry that is specific to a given function.

[0147] In one or more exemplary embodiments, the functions described may be implemented in hardware, software, firmware, or any combination thereof. If implemented in software, the functions may be stored on or transmitted over as one or more instructions or code on a non-transitory processor-readable storage medium (or computer-readable storage medium). The steps of a method or algorithm disclosed herein may be embodied in a processor-executable software module or processor-executable instructions (or processor-executable software instructions) which may reside on a non-transitory processor-readable storage medium (or non-transitory computer-readable storage medium). Non-transitory processor-readable storage media may be any available medium that may be accessed by a computing device. By way of example, and not limitation, such non-transitory processor-readable media may comprise RAM, ROM, EEPROM, CD-RIM or other optical disk storage, magnetic disk storage or other magnetic storage computing devices, or any other medium that may be used to store desired program code or the like for the form of instructions or data structures and that may be accessed by a computing device. Disk and disk, as used herein, includes compact disc (CD), laser disc, optical disc, digital versatile disc (DVD), floppy disk, and blu-ray disc where disks usually reproduce data magnetically, while discs reproduce data optically with lasers. Combinations of the above should also be included within the scope of non-transitory processor-readable media. Additionally, the operations of a method or algorithm may reside as one or any combination or set of codes and/or instructions on a non-transitory processor-readable storage medium, which may be incorporated into a computer program product.

[0148] The preceding description of the disclosed embodiment is provided to enable any person skilled in the art to make or use the present invention. Various modifications to these embodiments will be readily apparent to those skilled in the art, and the generic principles defined herein may be applied to other embodiments without departing from the spirit or scope of the invention. Thus, the present invention is not intended to be limited to the embodiments shown herein but is to be accorded the widest scope consistent with the following claims and the principles and novel features disclosed herein.

What is claimed is:

1. A method for improving user experience, energy consumption, and performance of a mobile device by automatically and dynamically configuring applications, comprising:
 obtaining, by a processor executing an application, operating conditions of the mobile device using an application programming interface;
 identifying, by the processor, a first software configuration of a plurality of software configurations based on the obtained operating conditions of the mobile device, wherein each in the plurality of software configurations define a set of operating parameters for the processor executing the application;
 activating, by the processor, the first software configuration with respect to the application;
 obtaining, by the processor, a first portion of a task shared between a plurality of nearby collaborating devices based on the activated first software configuration, wherein the task is processing data collectively stored across the plurality of nearby collaborating devices; and
 performing, by the processor, the first portion of the task using the application configured with the activated first software configuration.
2. The method of claim 1, wherein the obtained operating conditions include one or more of available power conditions, battery conditions, temperature conditions, available communication protocols, available networking interfaces, network connectivity, past usage data, and processor workload.
3. The method of claim 1, wherein performing, by the processor, the first portion of the task using the application configured with the activated first software configuration comprises one of:
 performing, by the processor, the operations of the first portion of the task in a shortest time period;
 performing, by the processor, the operations of the first portion of the task until a predetermined threshold is exceeded, wherein the predetermined threshold relates to one of a battery level, a temperature of the mobile device, and a location of the mobile device; and
 performing, by the processor, the operations of the first portion of the task at a fixed time period or periodically.
4. The method of claim 3, wherein performing, by the processor, the first portion of the task using the application configured with the activated first software configuration comprises exchanging data with one or more of the plurality of nearby collaborating devices using a preferred communication protocol.
5. The method of claim 4, further comprising receiving, by the processor, a command signal related to the task shared between the plurality of nearby collaborating devices, wherein the command signal instructs a change of an active software configuration based on the operating conditions.
6. The method of claim 5, further comprising:
 deactivating, by the processor, the first software configuration with respect to the application in response to receiving the command signal;
 activating, by the processor, a second software configuration with respect to the application in response to the processor deactivating the first software configuration; and
 obtaining, by the processor, a second portion of the task shared between the plurality of nearby collaborating devices to be processed by the processor using the application configured with the second software configuration.
7. The method of claim 6, wherein the second software configuration utilizes a different communication protocol than the first software configuration.
8. A mobile device, comprising:
 a processor configured with processor-executable instructions to perform operations comprising:
 obtaining, operating conditions of the mobile device using an application programming interface;
 identifying a first software configuration of a plurality of software configurations based on the obtained operating conditions of the mobile device, wherein each in the plurality of software configurations define a set of operating parameters for an application;
 activating the first software configuration with respect to the application;
 obtaining a first portion of a task shared between a plurality of nearby collaborating devices based on the activated first software configuration, wherein the task is processing data collectively stored across the plurality of nearby collaborating devices; and
 performing the first portion of the task using the application configured with the activated first software configuration.
9. The mobile device of claim 8, wherein the obtained operating conditions include one or more of available power conditions, battery conditions, temperature conditions, available communication protocols, available networking interfaces, network connectivity, past usage data, and processor workload.
10. The mobile device of claim 8, wherein the processor is configured with processor-executable instructions to perform operations such that performing the first portion of the task using the application configured with the activated first software configuration comprises one of:
 performing the operations of the first portion of the task in a shortest time period;
 performing the operations of the first portion of the task until a predetermined threshold is exceeded, wherein the predetermined threshold relates to one of a battery level and a temperature of the mobile device; and
 performing the operations of the first portion of the task in a fixed time period.
11. The mobile device of claim 10, wherein the processor is configured with processor-executable instructions to perform operations such that performing the first portion of the task using the application configured with the activated first software configuration comprises exchanging data with one or more of the plurality of nearby collaborating devices using a preferred communication protocol.
12. The mobile device of claim 8, wherein the processor is configured with processor-executable instructions to perform

- operations further comprising receiving a command signal related to the task shared between the plurality of nearby collaborating devices, wherein the command signal instructs a change of an active software configuration based on the operating conditions.
13. The mobile device of claim 12, wherein the processor is configured with processor-executable instructions to perform operations further comprising:
- deactivating the first software configuration with respect to the application in response to receiving the command signal;
 - activating a second software configuration with respect to the application in response to deactivating the first software configuration; and
 - obtaining a second portion of the task shared between the plurality of nearby collaborating devices to be processed using the application configured with the second software configuration.
14. The mobile device of claim 13, wherein the second software configuration utilizes a different communication protocol than the first software configuration.
15. A system, comprising:
- a first mobile device within a plurality of nearby collaborating devices, wherein the first mobile device comprises a first processor configured with processor-executable instructions to perform operations comprising:
 - obtaining operating conditions of the first mobile device using an application programming interface,
 - identifying a first software configuration of a plurality of software configurations based on the obtained operating conditions of the first mobile device, wherein each in the plurality of software configurations define a set of operating parameters for an application,
 - activating the first software configuration with respect to the application,
 - obtaining a first portion of a task shared between the plurality of nearby collaborating devices based on the activated first software configuration, wherein the task is processing data collectively stored across the plurality of nearby collaborating devices, and
 - performing the first portion of the task using the application configured with the activated first software configuration.
16. The system of claim 15, wherein the obtained operating conditions include one or more of available power conditions, battery conditions, temperature conditions, available communication protocols, available networking interfaces, network connectivity, past usage data, and processor workload.
17. The system of claim 15, wherein the first processor is configured with processor-executable instructions to perform operations such that performing the first portion of the task using the application configured with the activated first software configuration comprises one of:
- performing the operations of the first portion of the task in a shortest time period;
 - performing the operations of the first portion of the task until a predetermined threshold is exceeded, wherein the predetermined threshold relates to one of a battery level and a temperature of the first mobile device; and
 - performing the operations of the first portion of the task in a fixed time period.
18. The system of claim 17, wherein the first processor is configured with processor-executable instructions to perform operations such that performing the first portion of the task using the application configured with the activated first software configuration comprises exchanging data with one or more of the plurality of nearby collaborating devices using a preferred communication protocol.
19. The system of claim 15, further comprising:
- a second mobile device within the plurality of nearby collaborating devices, the second mobile device comprising a second processor configured with processor-executable instructions to perform operations comprising:
 - transmitting a command signal related to the task shared between the plurality of nearby collaborating devices, wherein the command signal instructs a change of an active software configuration based on the operating conditions;
 - wherein the first processor is configured with processor-executable instructions for performing operations further comprising:
 - receiving the command signal;
 - deactivating the first software configuration with respect to the application in response to receiving the command signal;
 - activating a second software configuration with respect to the application in response to deactivating the first software configuration; and
 - obtaining a second portion of the task shared between the plurality of nearby collaborating devices to be processed using the application configured with the second software configuration.
20. The system of claim 19, wherein the second software configuration utilizes a different communication protocol than the first software configuration.

S 4 S 3 S



US 20170083364 A1

(19) United States

(21) Patent Application Publication (10) Pub. No.: US 2017/0083364 A1
(43) Pub. Date: Mar. 23, 2017

(54) PROACTIVE RESOURCE MANAGEMENT FOR PARALLEL WORK-STEALING PROCESSING SYSTEMS

(71) Applicant: QUALCOMM Incorporated, San Diego, CA (USA)

(72) Inventors: Hua Zhan, Santa Clara, CA (USA); Dario Suarez Gracia, Israel (ES); Tushar Kumar, San Jose, CA (USA)

(21) Appl. No.: 14/862,373

(22) Filed: Sep. 23, 2015

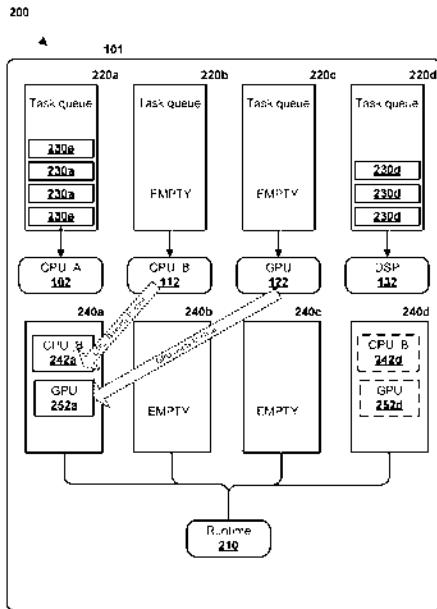
Publication Classification

(51) Int. Cl.
G06F 9/48 (2006.01)
G06F 9/50 (2006.01)

(52) U.S. Cl.
C18C G06F 9/481W (2013.01), G06F 9/488I (2013.01); G06F 9/503 (2013.01)

(57) ABSTRACT

Various embodiments proactively balance workloads between a plurality of processing units of a multi-processor computing device by making work-stealing determinations based on operating state data. An embodiment method includes obtaining static characteristics data associated with each of a victim processor and one or more of a plurality of processing units that are ready to steal work items from the victim processor (work-ready processors), obtaining dynamic characteristics data for each of the processors, calculating priority values for each of the processors based on the obtained data, and transferring a number of work items assigned to the victim processor to a winning work-ready processor based on the calculated priority values. In some embodiments, the method may include acquiring control over a probabilistic lock for a shared data structure and updating the shared data structure to indicate the number of work items transferred to the winning work-ready processor.



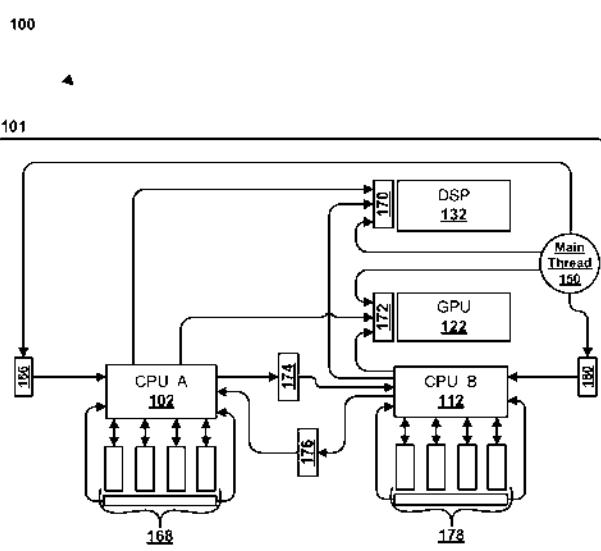


FIG. 1

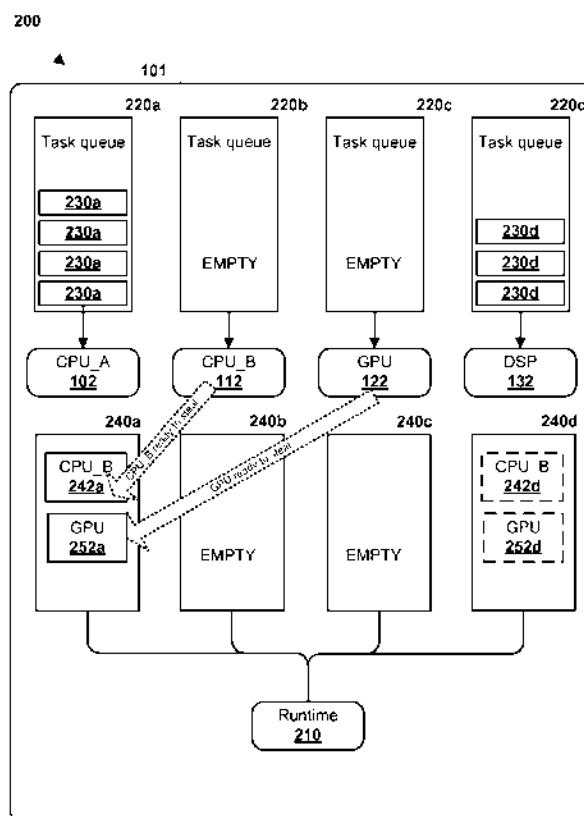


FIG. 2A

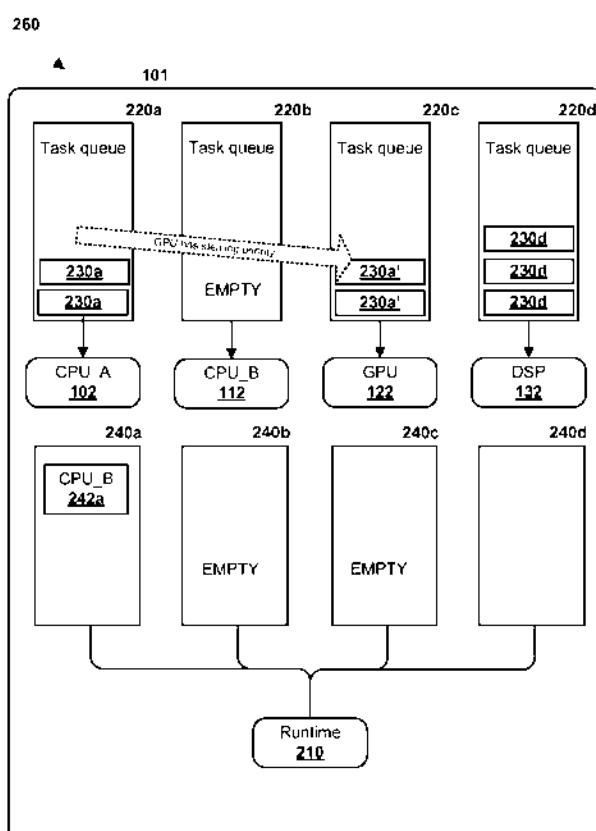


FIG. 2B

270

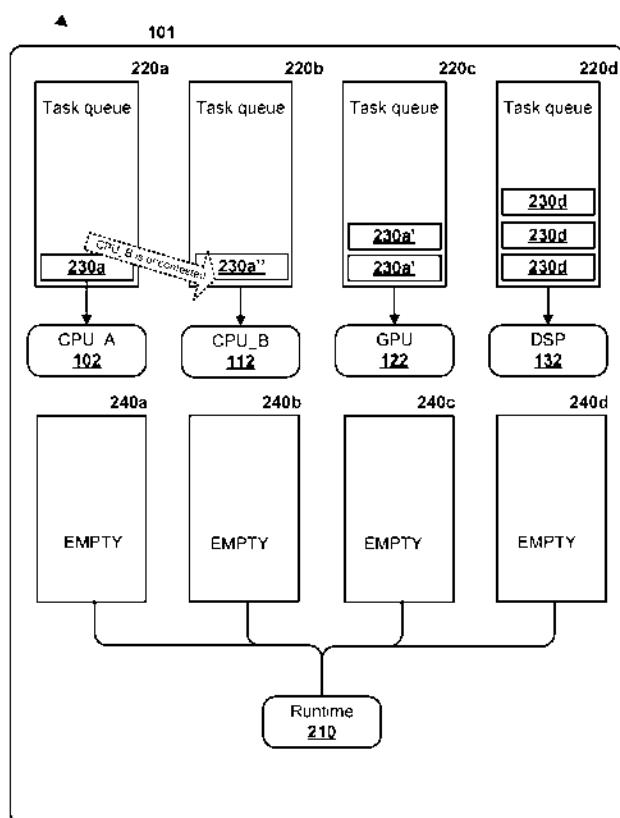


FIG. 2C

280

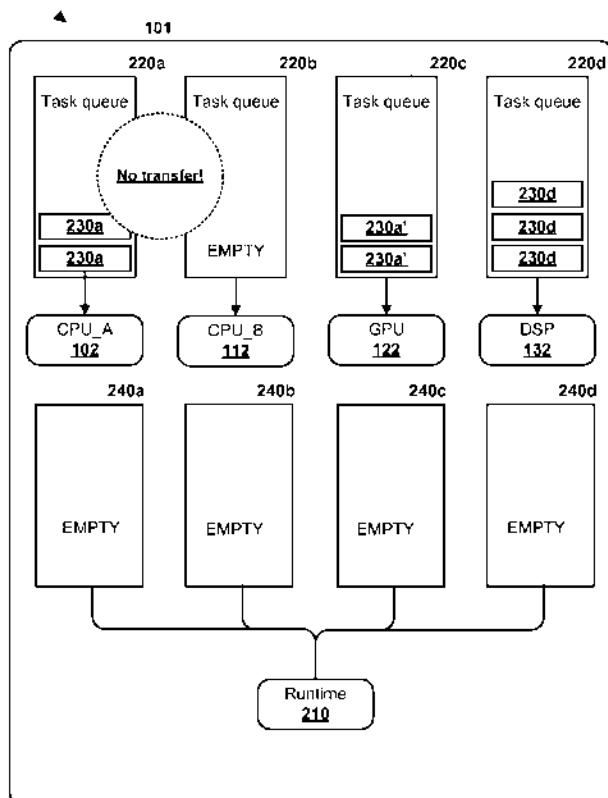


FIG. 2D

299

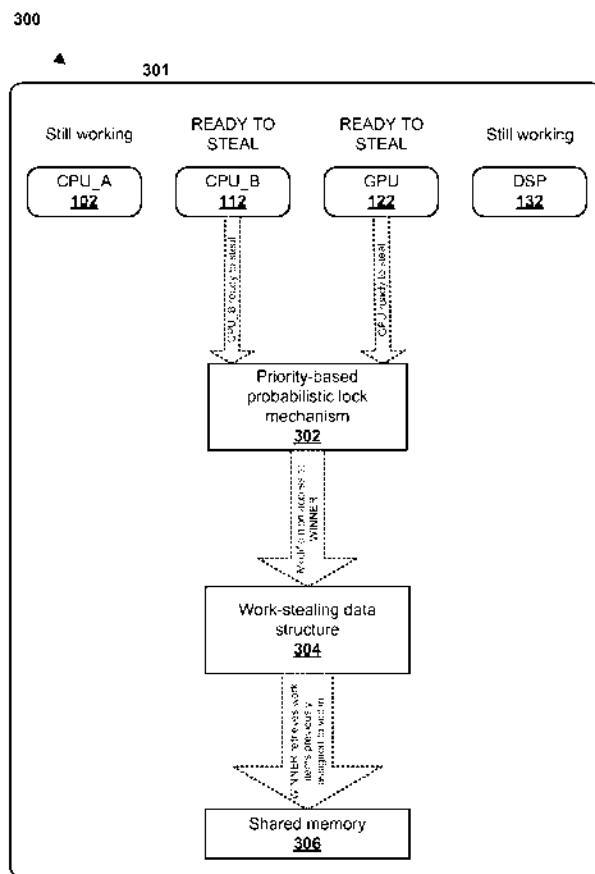


FIG. 3

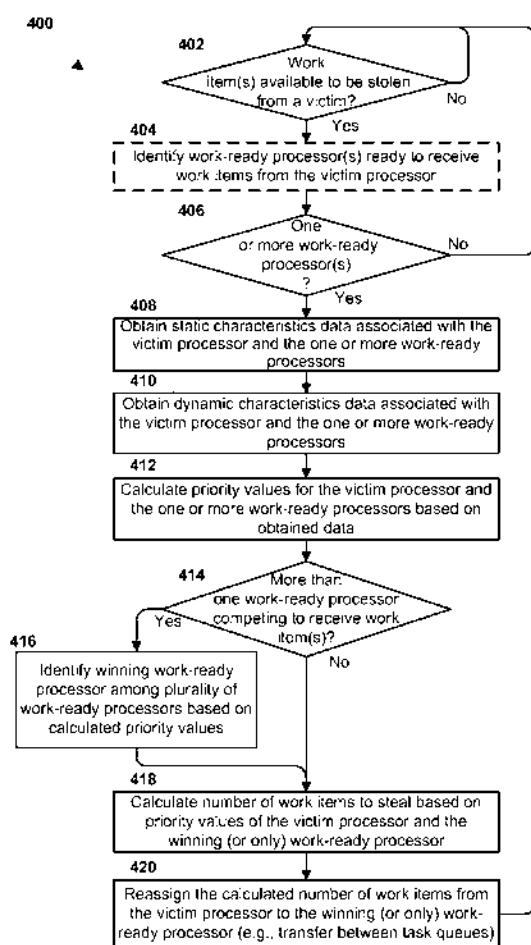


FIG. 4A

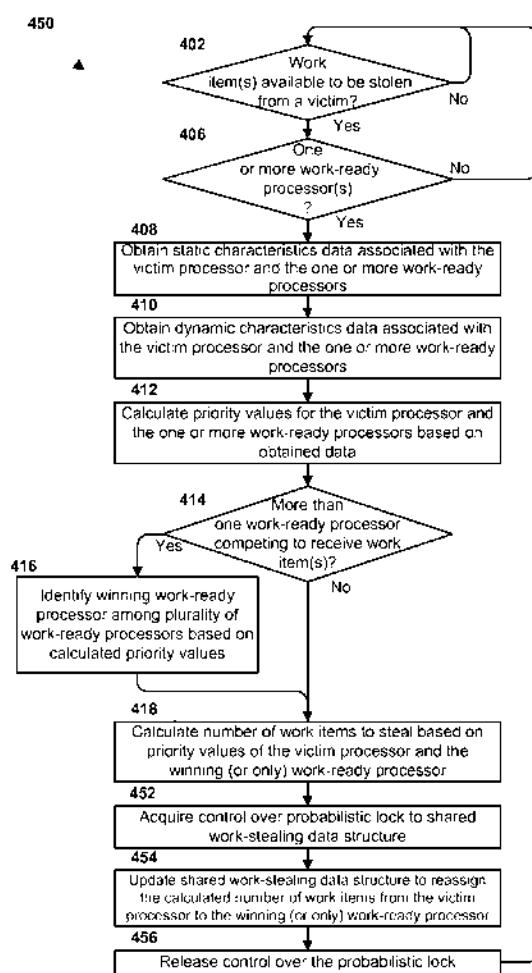


FIG. 4B

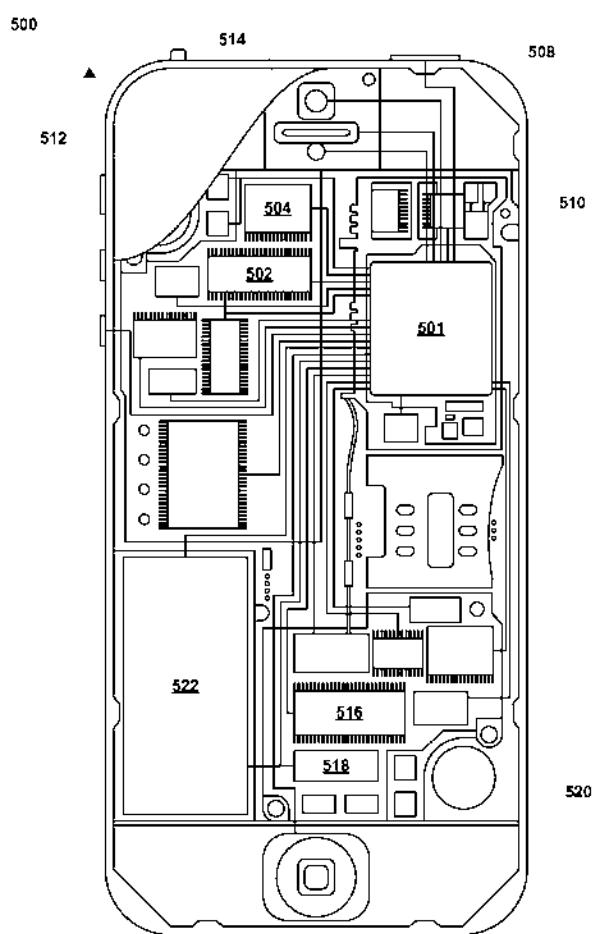


FIG. 5

PROACTIVE RESOURCE MANAGEMENT FOR PARALLEL WORK-STEALING PROCESSING SYSTEMS

BACKGROUND

[0001] Data parallel processing is a form of computing parallelization across multiple processing units. Data parallel processing is often used to perform workloads that can be broken up and concurrently executed by multiple processing units. For example, to execute a parallel loop routine having 1000 iterations, four different processing units may each be configured to perform 250 different iterations (or different sub-ranges) of the parallel loop routine. In the context of data parallel processing, a task can represent an abstraction of sequential computational work that may have a dynamic working size that is typically determined at runtime. For example, a processor can execute a task that processes a sub-range of a parallel loop routine; at some cases, a task may be created by a thread running on a first processor and dispatched to be processed via another thread on a second processor.

[0002] Different tasks may be assigned for offloaded to various processing units of a multi-core or multi-processor computing device (e.g., a heterogeneous system-on-chip (SoC)). Typically, a task-based runtime system (or task scheduler) determines in which processing unit a task may be assigned. For example, a scheduler can launch a set of concurrently-executing tasks on a plurality of processing units, each unit differently able to perform operations on data for a parallel loop routine.

[0003] Multi-processor (or multi-core) systems are often configured to implement data parallelism techniques to provide responsive and high performance software. For example, with data parallel processing capabilities, a multi-core device commonly launches a number of dynamic tasks on different processing units in order to achieve load balancing. Parallel workloads may get imbalanced between various processing units. For example, while multiple processing units may initially get equal sub-ranges of a parallel loop routine, imbalance in execution time may occur. Imbalances in workloads may occur for many reasons, such as that the amount of work per work item is not constant (e.g., some work items may require less work than other work items, etc.); if the capabilities of heterogeneous processing units may differ (e.g., high GPU CPU cores, CPU vs. GPU, etc.); rising temperature may throttle frequencies on some processing units more than others, particularly if the heat dissipation is not uniform (as is commonly the case); and other loads may cause some processors to lag more (e.g., loads from other applications, the servicing of system interrupt, and/or the effects of OS scheduling, etc.).

[0004] To improve performance while conducting data parallel processing, multi-processor systems may employ work-stealing techniques in which tasks or processing units can be configured to opportunistically take or execute work items originally assigned to other tasks or processing units. For example, when a first processing unit or task finishes its assigned subrange of a shared workload (e.g., a parallel loop routine), the first processing unit may steal additional subranges of work from other processing units that are still busy processing respective assignments of the shared workload. As local imbalances may often occur, work-stealing operations may allow dynamic load-balancing that

improves the utilization of system processing resources and reduces the time to complete parallel work.

SUMMARY

[0005] Various embodiments provide methods, devices, systems, and non-transitory processor-readable storage media for a multi-processor computing device to proactively balance workloads among a plurality of processing units by making work-stealing determinations based on operating state data. An embodiment method performed by a processor of the multi-processor computing device may include operations for obtaining static characteristics data associated with each of a victim processor and one or more of the plurality of processing units that are ready to steal work items from the victim processor (work-ready processors), obtaining dynamic characteristics data associated with each of the victim processor and the work-ready processors, calculating priority values for each of the victim processor and the work-ready processors based on the obtained data, and transferring a number of work items assigned to the victim processor to a winning work-ready processor based on the calculated priority values.

[0006] In some embodiments, the work-ready processors may be ready to steal the work items from the victim processor in response to the work-ready processors finishing respective assigned work items. In some embodiments, the static characteristics data associated with each of the victim processor and the work-ready processors may include any of data indicating a technology used, a location in a topology of the multi-processor computing device, and manufacturer data. In some embodiments, the static characteristics data may be obtained at a time of manufacture of the multi-processor computing device, at each boot-up of the multi-processor computing device, on a periodic basis, or any combination thereof. In some embodiments, the dynamic characteristics data associated with each of the victim processor and the work-ready processors may include any of data indicating a technology used, a location in a topology of the multi-processor computing device, and manufacturer data. In some embodiments, the static characteristics data may be obtained at a time of manufacture of the multi-processor computing device, at each boot-up of the multi-processor computing device, on a periodic basis, or any combination thereof. In some embodiments, the dynamic characteristics data associated with each of the victim processor and the work-ready processors may include local measurements of each of the victim processor and the work-ready processors. In some embodiments, the dynamic characteristics data associated with each of the victim processor and the work-ready processors may include values associated with temperature, power consumption, and frequency.

[0007] In some embodiments, calculating the priority values for each of the victim processor and the work-ready processors based on the obtained data may include calculating the priority values using the following equation:

$$P_i = \alpha * \text{temp}_i + \beta * \text{power}_i + \gamma * \text{freq}_i$$

where i may represent an index for one of the victim processor and the work-ready processors, P_i may represent a calculated priority value for a processing unit associated with the index for one of the victim processor and the work-ready processors, α may represent a global coefficient for temperature, β may represent a local coefficient for the temperature for the processing unit, temp_i may represent a formula over a local temperature measurement for the processing unit, β may represent a global coefficient for power consumption, γ may represent a local coefficient for the power consumption for the processing unit, Power_i may represent a formula over a local power consumption measurement for the processing unit, γ may represent a global coefficient for frequency, δ may represent a local coefficient for the frequency of the processing unit, and f_{req} may

represent a formula over a local frequency measurement for the processing unit. In some embodiments, obtaining the static characteristics data associated with each of the victim processor and the work-ready processors may include obtaining the global coefficient for temperature, the global coefficient for power consumption, and the global coefficient for frequency. In some embodiments, obtaining dynamic characteristics data associated with each of the victim processor and the work-ready processors may include obtaining the local coefficient for the temperature for the processing unit, the local temperature measurement for the processing unit, the local coefficient for the power consumption for the processing unit, the local power consumption measurement for the processing unit, the local coefficient for the frequency of the processing unit, and the local frequency measurement for the processing unit.

[0018] In some embodiments, the method may further include calculating the number of work items to transfer to the winning work-ready processor based on the calculated priority values of the victim processor and the winning work-ready processor.

[0019] In some embodiments, calculating the number of work items to transfer to the winning work-ready processor based on the calculated priority values of the victim processor and the winning work-ready processor may include calculating the number of work items to transfer using the following equation:

$$N_i = P_j - P_i + P_k$$

where i may represent a first task index associated with the victim processor, j may represent a second task index associated with the winning work-ready processor, K may represent a total number of work items remaining to be processed by the victim processor prior to stealing, P_i may represent a calculated priority value of the victim processor, and P_j may represent the calculated priority value of the winning work-ready processor, N_i may represent the number of work items to transfer to the winning work-ready processor, and N_j may represent a number of work items to remain with the victim processor.

[0020] In some embodiments, each of the victim processor and the work-ready processors may be associated with a different processing unit in the plurality of processing units. In some embodiments, each of the victim processor and the work-ready processors may be associated with a different task executing on one of the plurality of processing units. In some embodiments, the work items may be different iterations of a parallel loop routine. In some embodiments, the plurality of processing units may include two or more of a first central processing unit (CPU), a graphics processing unit (GPU), and a digital signal processor (DSP). In some embodiments, the multi-processor computing device may be one of a heterogeneous multi-core computing device, a homogeneous multi-core computing device, or a distributed computing server.

[0021] In some embodiments, the method may further include acquiring, via the winning work-ready processor, control over a probabilistic lock for a shared data structure, based on a calculated priority value of the winning work-ready processor, and updating, via the winning work-ready processor, the shared data structure to indicate the number of work items transferred to the winning work-ready processor in response to acquiring control over the probabilistic lock.

[0022] Further embodiments include a computing device configured with processor-executable instructions for performing operations of the methods described above. Further embodiments include a non-transitory processor-readable medium on which is stored processor-executable instructions configured to cause a computing device to perform operations of the methods described above.

BRIEF DESCRIPTION OF THE DRAWINGS

[0023] The accompanying drawings, which are incorporated herein and constitute part of this specification, illustrate exemplary embodiments, and together with the general description given above and the detailed description given below, serve to explain the features of the claims.

[0024] FIG. 1 is a component block diagram illustrating task queues and processing units of an exemplary heterogeneous multi-processor computing device (e.g., a heterogeneous system-on-chip (SoC)) suitable for use in some embodiments.

[0025] FIGS. 2A-2D are component block diagrams illustrating exemplary work-stealing operations by a multi-processor computing device according to some embodiments.

[0026] FIG. 3 is a component block diagram illustrating exemplary work-stealing operations by a multi-processor computing device having a shared memory and work-stealing data structure according to some embodiments.

[0027] FIGS. 4A-4B is a process flow diagram illustrating an embodiment method performed by a multi-processor computing device to proactively balance workloads between a plurality of processing units by making work-stealing determinations based on operating state data.

[0028] FIG. 5 is a component block diagram of a computing device suitable for use in some embodiments.

Detailed Description

[0029] The various embodiments will be described in detail with reference to the accompanying drawings. Wherever possible, the same reference numbers are used throughout the drawings to refer to the same or like parts. References made to particular examples and implementations are for illustrative purposes, and are not intended to limit the scope of the embodiments in the claims.

[0030] The word "exemplary" is used herein to mean "serving as an example, instance, or illustration." Any implementation described herein as "exemplary" is not necessarily to be construed as preferred or advantageous over other implementations.

[0031] The term "computing device" is used herein to refer to an electronic device equipped with at least a multi-core processor. Examples of computing devices may include mobile devices (e.g., cellular telephones, wearable devices, smart-phones, web-pads, tablet computers, internet enabled cellular telephones, Wi-Fi enabled electronic devices, personal data assistants (PDAs), laptop computers, etc.), personal computers, and server computing devices. In various embodiments, computing devices may be configured with various memory and/or data storage, as well as networking capabilities, such as network transceiver(s) and antenna(s) configured to establish a wide area network (WAN) connection (e.g., a cellular network connection, etc.) and/or a local area network (LAN) connection (e.g., a wired/wireless connection to the Internet via a Wi-Fi® router, etc.).

[0022] The terms "multi-processor computing device" and "multi-core computing device" are used herein to refer to computing devices configured with two or more processing units. Multi-processor computing devices may execute various operations (e.g., routines, functions, tasks, calculations, data structure sets, etc.) using two or more processing units. A "homogeneous multi-processor computing device" may be a multi-processor computing device (e.g., a system-on-chip (SoC)) with a plurality of the same type of processing unit, each configured to perform workloads. A "heterogeneous multi-processor computing device" may be a multi-processor computing device (e.g., a heterogeneous system-on-chip (SoC)) with different types of processing units that may each be configured to perform specialized and/or general-purpose workloads. Processing units of multi-processor computing devices may include various processor devices, a core, a plurality of cores, etc. For example, processing units of a heterogeneous multi-processor computing device may include an application processor(s) (e.g., a central processing unit (CPU)) and specialized processing devices, such as a graphics processing unit (GPU), and a digital signal processor (DSP), any of which may include one or more internal cores. As another example, a heterogeneous multi-processor computing device may include a mixed cluster of big and little cores (e.g., ARM big.LITTLE architecture, etc.) and various heterogeneous systems devices (e.g., GPU, DSP, etc.).

[0023] The terms "work-ready processor" and "work-ready processors" are generally used herein to refer to processing units and/or tasks executing on the processing units that are ready to receive one or more workloads via a work-stealing policy. For example, a "work-ready processor" may be a processing unit capable of receiving individual work items from other processing units or tasks executing on the other processing units. Similarly, the term "victim processor(s)" is generally used herein to refer to a processing unit and/or a task executing on the processing unit that has one or more workloads (e.g., individual work items, tasks, etc.) that may be transferred to one or more work-ready processors. The term "work-ready task" is used herein to refer to a work-ready processor that is a task executing on a processing unit. The term "work-ready processing unit" refers to a work-ready processor that is a processing unit. The term "victim task" is used herein to refer to a victim processor that is a task executing on a processing unit. The term "victim processing unit" refers to a victim processor that is a processing unit. However, such terms are not intended to limit any embodiments or claims to specific types of work-ready processor or victim processor.

[0024] In general, work stealing can be implemented in various ways, depending on the nature of the parallel processing computing system. For example, a shared memory multi-processor system may employ a shared data structure (e.g., a tree representation of the work sub-ranges) to represent the sub-division of work across the processing units. In such a system, stealing may require work-ready tasks to concurrently access and update the shared data structure via locks or atomic operations. As another example, a processing unit may utilize associated work-queues such that, when the queues are empty, the processing unit may steal work items from another processing unit and add the stolen work items the work queues of the first processing unit. In a similar manner, another processing unit may steal work

items from the first processing unit's work-queues. Conventional work-stealing schemes are often rather simplistic, such as merely enabling one processing unit to share (or steal) an equally-subdivided range of a workload from a victim processing unit.

[0025] Traditionally, work-stealing has been assumed to be beneficial. However, in many situations, conventional work-stealing can result in counter-productive, sub-optimal results for computing devices executing under tight power and/or thermal constraints. For example, when a task is ready to accept (or steal) work but is executing on a processor that is already close to a thermal limit, stealing too much work (e.g., too many iterations of a parallel long task) may not be helpful to the overall performance of a shared workload. Some traditional work-stealing techniques may employ a "reactive" approach that looks only at immediate states without considering implications on near-future states of the computing system. For example, conventional ad-hoc power management thermal throttling may be used in a reactive manner to mitigate thermal issues in multi-processor systems. Further, some conventional work-stealing techniques often employ shallow "first-come, first-served" schemes to award work items to a work-ready processor (e.g., the first processing unit to request or be ready to steal). Other techniques may use atomic operations to resolve data races between competing work-ready processors such that only one work-ready processor may succeed in taking over some work.

[0026] Such conventional work-stealing techniques fail to acknowledge that dynamic tasks scheduled on different power-hungry processing units often encounter different power consumption and thermal states, which may affect the benefit achieved by transferring work among processors. For example, conventional power management techniques may have information gaps between runtime assessments of power and thermal conditions and data collected centrally at an operating system architecture, making prioritization and load balancing suboptimal in parallel scheduling situations.

[0027] Various embodiments provide methods, devices, systems, and non-transitory processor-readable storage media for using data from a task-based runtime system to improve data parallelism and load balance in a multi-processor computing device (e.g., a heterogeneous system, etc.). In general, the multi-processor computing device may identify operating state data of different processing units executing tasks. The operating state data may include static device information and/or up-to-date measurements (e.g., temperature, frequency, power consumption) for the processing units. The multi-processor computing device may also maintain or otherwise calculate other information related to the operating conditions of the processing units, such as coefficients that indicate temperature, frequency, and/or power consumption levels of the processing units. With various operating state data, the multi-processor computing device may calculate priority values that may be used to enable comprehensive work-stealing decisions. For example, when work-ready processors are available to receive work items from a victim processor, the multi-processor computing device may use the calculated priorities of the various tasks to decide the most appropriate manner to redistribute work items. By assigning work based on at least current power consumption and thermal conditions, the multi-processor computing device may regulate the size of dynamic tasks in

data parallel processing and proactively alleviate power consumption and hot spot issues.

[0028] In various embodiments, the multi-processor computing device may calculate a priority value for each processing unit (and/or associated task) using various types of measured or sampled data, in order to compare or make relative priorities between processing units. The multi-processor computing device may utilize operating state data specific to each processing unit (e.g., local data) as well as information that is applicable to all processing units (e.g., global data). In other words, the multi-processor computing device may utilize a global awareness of the states of all processing units to calculate priorities at a given time.

[0029] In various embodiments, the multi-processor computing device may utilize dynamically sampled data that may be used to prioritize tasks. For example, each processing unit executing a task may periodically provide to a runtime system (or scheduler) measurement data indicating current thermal output, power use (or power consumption rate), and/or operating frequency. Such data may be stored as a power profile for each processing unit (or an associated task).

[0030] In some embodiments, the multi-processor computing device may use a state query statement when a task begins execution on a processing unit in order to establish profiles or otherwise associate operating state data with processing units and/or tasks. Executing such a state query statement (e.g., an API call, etc.) may cause the processing unit to return data of a variety of local properties (e.g., frequency, leakage current, or active power). For example, when a task is dispatched to the DSP and ready to execute work items of a parallel loop routine (or workload), the multi-processor computing device may use a state query statement to tag the task with power consumption, frequency, and thermal operating state data for use in subsequent priority assessments.

[0031] In some embodiments, the multi-processor computing device may use a deterministic approach to calculate priority values and thus identify a work-ready processor that will receive work items from a victim processor. In particular, the multi-processor computing device may use various deterministic priority equations, such as schemes that emphasize different task properties (e.g., favoring the task with lower leakage power, or lower active power to maximize battery life, etc.). Such deterministic approaches may reduce communication overhead and may balance local power thermal conditions by identifying work-ready processors that best balance work efficiency with other operating states of the system. In some embodiments, the multi-processor computing device may calculate a priority value for each individual processing unit for stealing work items using the following deterministic equation:

$$\text{Priority}_i = \alpha_i \cdot \text{Power}_i + \beta_i \cdot \text{Freq}_i + \gamma_i \cdot \text{Temp}_i \quad [Eq. 1]$$

In Equation 1, i represents an identity for index for a processing unit configured to execute dynamic tasks. P_i represents an overall stealing priority of the processing unit (relative to the stealing priorities of other processing units). α_i represents a global coefficient for temperature. t_i represents a local coefficient for temperature (or temperature value) for the processing unit. Temp_i represents a term formula incorporating a local temperature measurement for the processing unit. β_i represents a global coefficient for power consumption. g_i represents a local coefficient for

power consumption (or power consumption value) for the processing unit. Power_i represents a term formula incorporating a local power consumption measurement for the processing unit. γ_i represents a global coefficient for frequency. f_i represents a local coefficient for frequency (or compute rate) for the processing unit. Freq_i represents a term formula incorporating a local frequency measurement for the processing unit. Larger numeric values generated using Equation 1 may indicate higher priorities.

[0032] The global coefficients of Equation 1 (i.e., $\alpha_i, \beta_i, \gamma_i$) may contribute in Equation 1 to uniformly balance the importance or impact of the local measurements (i.e., $\text{Temp}_i, \text{Power}_i, \text{Freq}_i$). Additionally, for each aspect (e.g., temperature, power, and frequency), the global coefficients may allow differentiation between units (and) (e.g., by adjusting α_i vs. α_j). In particular, the global coefficients may incorporate static characteristics of the processing units to differentiate between units. For example, the global coefficients may be generated to reflect static process technology variations (e.g., differing process used per core, such as big.LITTLE, differing leakage, frequency, etc.); packaging, and/or topology that may affect thermal dissipation (e.g., which cores can heat up or cool down quickly; etc.). As another example, the global coefficients may be generated to reflect workload-specific variations of processing units, such as temperature change rates due to various instruction-mixes and heterogeneity (e.g., CPU vs. DSP vs. GPU). As another example, the global coefficients may be generated to reflect system use (or system topology awareness), such as by providing data of components (e.g., a modest, etc.) that are busy and getting hot in order to de-prioritize related or nearby processing units. In some embodiments, the global coefficients may indicate whether processing units are suited for certain types of workloads (e.g., instructions for "big" architecture vs. "little" architecture, floating-point operations, etc.). In this way, the global coefficients may differentiate between processing units based on the characteristics of the parallel workload.

[0033] In some embodiments, a global scheduler may be configured to collect information across all processing units of the multi-processor computing device for comparisons that may adjust the global coefficient values. Further, the global coefficient values may incorporate comparisons across various processing units. For example, the multi-processor computing device may evaluate and/or incorporate relative factors between processing units in order to generate $\alpha_i, \beta_i, \gamma_i$ for each processor. For example, the multi-processor computing device may calculate that a very hot processor with index "i" has a smaller compared a cooler processor with index "j" (e.g., $\alpha_i < \alpha_j$).

[0034] In some embodiments, data used to generate the global coefficients may be obtained via periodic feedback from the processing units and/or from compilation data indicating characteristics of to-be processed workloads. In some embodiments, the global coefficients may be set, adjusted, and/or otherwise determined offline (e.g., at manufacture, etc.), at boot-up of the multi-processor computing device, and/or periodically, such as via a global scheduler functionality (e.g., an OS-level thread, etc.).

[0035] The local coefficients of Equation 1 (i.e., α_i, β_i) may be coefficients determined based on estimations local to each processing unit (e.g., estimations typically done by a local scheduler for each processing unit). In particular, α_i may be task-specific (or processing unit-specific) values generated

by the multi-processor computing device based on data measured locally at the processing units. For example, the local coefficients may be based on the local properties of a certain processing unit and properties exhibited when the work items execute on that processing unit. The local coefficients (i.e., b_i) may be estimates of local processing unit properties over periods of time, such as average throughput observed over a number of prior work items, some estimations about observed temperature trends, and/or instruction-mix over a sequence of work items. In some embodiments, the local coefficients may be individually adjusted by a local scheduler functionality based only on data associated with each individual processing unit.

[0036] In some embodiments, the b_i coefficient may be calculated to be a small value for certain processing units that are expected to heat up quickly based on the workload, such as if compute-bound, floating-point heavy work items execute on a CPU. Compile-time knowledge of the instruction-mix for the application code for a processing unit may provide such information. When work items are observed or expected to be similar as prior work items, the execution of the prior work items may help estimate a better b_i for subsequent work items. The g_i coefficient may be inversely proportional to an estimated energy per work item for a particular processing unit. For example, a first CPU core with a higher frequency may utilize more energy to process one work item compared to a second CPU core with a lower frequency, and thus the first CPU core may have a lower g_i value, helping prioritize power-efficiency. It that is the desired goal. A local scheduler may compute g_i based on estimated energy-per-work item over a recent sequence of work items at a processing unit indexed "i", making g_i tuned to be specific to characteristics of the work items. The h_i coefficient may be proportional to a throughput of work items for compute tasks at a particular processing unit. In some embodiments, the higher the h_i coefficient value, the higher the calculated priority may be for a task or processing unit. In some embodiments, a GPU may be calculated to have a higher priority with regard to performance, but a DSP may be calculated to have a higher priority with regard to power-saving.

[0037] In various embodiments, the local coefficients (b_i , g_i , h_i) may represent "local" decisions by a local scheduler running on and using location information for each processing unit (e.g., on a processing unit with index "i").

[0038] The local measurements in Equation 1 (i.e., Temp, Power, Freq) may be formulas over normalized measurements of the current operating conditions at individual processing units. In general, the local measurements (per-core measured metrics) may allow faster adjustment than a local scheduler. The normalization may typically be with regard to some maximum temperature limit, power level or allowed frequency setting. In some embodiments, the priority of a processing unit (or task) may be lower when a total temperature is higher. For example, the overall priority calculation for a certain processing unit may be lower when the processing unit's measured temperature at a given time is higher. The local measurements of Equation 1 (i.e., Temp, Power, Freq) may be based on instantaneous measurements of various operating conditions.

[0039] In some embodiments, the local measurements may be generated using the following equations:

$$\text{Temp}_i = T_i / T_{\text{max}} \quad (1)$$

$$\text{Power}_i = P_i / P_{\text{max}} \quad (2)$$

$$\text{Freq}_i = f_i / f_{\text{max}} \quad (3)$$

where T_i represents an estimated or measured temperature and T_{max} represents a maximum temperature for limit at a processing unit with the index i ;

$$\text{Power}_i = P_i / P_{\text{max}} \quad (4)$$

where P_i represents an estimated or measured core power burn and P_{max} represents a maximum power consumption level (or limit) at a processing unit with the index i , and

$$\text{Freq}_i = f_i / f_{\text{max}} \quad (5)$$

where f_i represents an estimated or measured frequency state and f_{max} represents a maximum frequency (or limit) at a processing unit with the index i . Regardless of how work items may have contributed to power consumption, Power, may be based on instantaneous power measurements of the processing unit index " i ".

[0040] In some embodiments, the multi-processor computing device may utilize per-work item estimations when calculating priorities. For example, the multi-processor computing device may estimate a throughput per work item on particular processing units, in which a measured completion rate of a work item for each processing unit may be normalized to a maximum frequency for each processing unit. Per-work item throughput estimations may be valuable to overall priority calculations when evaluated work items are similar in execution time. Per-work item throughput estimations may also be valuable when work items are not similar, but the variations are spread evenly throughout the iteration-space of the work items. In some embodiments, such estimations may be performed by the local schedulers of processing units and may be reflected in local prioritization.

[0041] In some embodiments, the multi-processor computing device may estimate an energy per work item on particular processing units. In such embodiments, an observed instruction mix (e.g., types(s) of instructions to be performed, amount of floating-point operations, etc.) may be identified at compile-time or runtime in order to estimate potential impacts on power consumption and/or temperature (e.g., more floating-point operations may increase thermal level faster, etc.).

[0042] In some embodiments, the multi-processor computing device may weigh or otherwise bias temperature, power consumption, and/or frequency measurements. For example, for a particular work-stealing decision, the multi-processor computing device may calculate priorities for a plurality of work-ready processors based on performance (e.g., frequency), but may not fully utilize data related to power consumption and/or temperatures related to the work-ready processors. In other words, priorities may be calculated using any combination and/or extent of the factors related to temperature, power consumption, and/or frequency. In some embodiments, the temperature, power consumption, and frequency factors may be highly correlated for particular processing units, allowing the use of only a subset of the factors to be adequate for calculating priority values.

[0043] In some embodiments, the multi-processor computing device may select work-ready processors for work-stealing ("winning" work-ready processors) using a probabilistic approach. In particular, the multi-processor computing device may utilize one or more "probabilistic locks" that may be accessed by work-ready processor-based on corresponding priority values. In general, with some

$\text{temp}_i = T_i / T_{\text{max}}$, $\text{power}_i = P_i / P_{\text{max}}$, $\text{freq}_i = f_i / f_{\text{max}}$,

parallel processing implementations, the multi-processor computing device may utilize shared memory. In typical cases, locks may be employed to restrict access to certain data within the shared memory. In such cases, work-stealing protocols may utilize a shared work-stealing data structure to enable winning work-ready processors to directly steal work items from a victim processor. The work-stealing data structure (e.g., a work-stealing tree data structure, etc.) may describe the processor list is responsible for what range of work items of a certain shared task for competition.

[0044] In some embodiments, work ready processors may contend in parallel to acquire a probabilistic lock on the shared data structure, typically by one winner processing unit succeeding and stealing at a time. Acquiring the probabilistic lock may be based on priorities of the competing work-ready processors. For example, the work-ready processor with the highest priority is calculated by the equation: $I_{\text{base}} \geq \text{min}(\text{P}_j, \text{P}_k)$. I_{base} may have a higher probability of acquiring the lock over other work-ready processors with lower priority values. In other words, two work-ready processing units (e.g., processing unit with index "j" and another processing unit with index "k") that are simultaneously contending for a probabilistic lock may have a probability of acquiring the probabilistic lock determined by priorities P_j and P_k respectively. The work-ready processor that acquires the probabilistic lock may have exclusive access to examine and modify data within the shared work-stealing data structure, such as by updating the data of the work-stealing data structure to reflect the work-ready processor that is stealing work from a victim processor, as well as how many work items are being stolen. When the probabilistic lock is released, additional work-ready processors may receive a chance to acquire the probabilistic lock. Although probabilistic lock embodiments may effectively create a competition or "race" between work-ready processors, the corresponding embodiment techniques may not be reactive as priorities are evaluated based on global and local coefficients and do not depend on particular work-stealing frameworks.

[0045] In some embodiments, acquiring the probabilistic lock may include a processing unit issuing for calling a function that accepts the processing unit's calculated priority value (e.g., P_j). In some embodiments, the rate at which a processing unit may retry acquiring a probabilistic lock may be based on a priority value. For example, a processing unit "i" may be allowed to call a lock-acquiring function at an interval based on a priority P_i.

[0046] In some embodiments, workload distribution decisions by the multi-processor computing device may be two-fold; a first decision to determine a winning work-ready processor, and a second decision to determine how much work to distribute to the winning work-ready processor. When a winning work-ready processor is determined, the multi-processor computing device may decide the amount of work for how many work items that is to be reassigned and transferred to the winning work-ready processor and the victim processor. For example, the multi-processor computing device may determine a percentage or split of work between a victim task and a winning work-ready task based on relative power thermal states at the time the decision is made. In some embodiments, the multi-processor computing device may calculate a priority for the victim processor and the winning work-ready processor to identify the number or percentage of available work items that may be stolen kept.

For example, based on priority values calculated using an equation as described herein, the multi-processor computing device may determine that the victim task should keep most of a set of available work items due to the winning task having only a marginal priority over the victim task based on the relative speeds of the two tasks.

[0047] In some embodiments, the multi-processor computing device may use the following equations to calculate a work item allocating between a victim processor and a winning work-ready processor:

$$\begin{aligned} X_i &= \text{min}(P_j, P_k) \\ X_j &= K - X_i \end{aligned}$$

In Equations 6 and 7, i represents a first task index for ID_i, j represents a second task index for ID_j, N_i represents the number of work items to be awarded to the first task, N_j represents the number of work items to be awarded to the second task, K represents a total number of work items remaining to be processed by the victim processor prior to stealing, P_j represents a priority of the second task, and P_i represents a priority of the first task. In various embodiments, the number of work items stolen by a work-ready processor may be in proportion to priorities calculated via Equation 1.

[0048] As a non-limiting illustration using exemplary Equations 8-10, a victim processor (i.e., processing unit or task with index i) and a work-ready processor (i.e., processing unit or task with index j) may be configured to execute work items for a parallel loop routine. At a given time, the work-ready processor may be identified as a winning work-ready processor over other work-ready processors ready to receive work items from the victim processor. At that time, there may be 10 iterations (i.e., K work items 10) remaining to be processed by the victim processor for the parallel loop routine. Further, the victim processor may be tagged with a first priority (P_i, 1), and the winning work-ready processor may be tagged with a second priority (P_j, 2). As a result, the multi-processor computing device may calculate that the victim processor may keep 3 iterations of the parallel loop routine (i.e., via Equation 5, N_i = 3.33 * 10*(1/1+2)); and that the winning work-ready processor may be allocated 7 iterations of the parallel loop routine (i.e., via Equation 6, N_j = 6.67 * 10/3, 3.33).

[0049] In some embodiments, a plurality of work-ready processors may be allocated work items from a victim processor. In such a case, the multi-processor computing device may utilize a similar approach to determine percentages of allocations as described herein with reference to Equation 5-6. In some embodiments, the multi-processor computing device may collectively examine a number of work-ready tasks in a work-ready queue and distribute stolen work items across multiple work-ready tasks in proportion to respective priorities.

[0050] In some embodiments, the multi-processor computing device may be configured to assign more work items to a slower processing unit and/or task (e.g., a more energy-efficient winner work-ready processor, such as a DSP). In such settings, the slower energy-efficient processor would be assigned a higher priority than faster processors due to having larger β_i or β_j coefficients. In this way, processing units and tasks with lower power leakage or lower active power may be favored in order to maximize battery life, with the energy efficiency policy being implemented simply by

an appropriate choice of the global and/or local coefficient values across the different processing units.

[0051] In various embodiments, the multi-processor computing device may execute one or more runtime functionalities (e.g., runtime service, routine, thread, or other software element, etc.) to perform various operations for scheduling or dispatching tasks. Such one or more functionalities may be generally referred to herein as a “runtime functionality.” The runtime functionality may be executed by a processing unit of the multi-processor computing device, such as a generic purpose or applications processor configured to execute operating systems, services, and/or other system-relevant software. For example, a runtime functionality executing on an application processor may be configured to calculate priorities of tasks running on one or more processing units. Other runtime functionalities may also be used, such as dedicated functionalities for handling tasks at individual processing units.

[0052] The processing units of the multi-processor computing device may be configured to execute one or more tasks that each may be associated with one or more work items. Some tasks may be configured to participate in iteration data parallel processing efforts. For example, a GPU¹ may be configured to perform a certain task for processing a set of work items for iterations of a parallel loop routine (or workload) also shared by a DSP and/or CPU. However, although the embodiment techniques described herein may be clearly used by the multi-processor computing device to improve such common-data parallel processing workloads on a plurality of processing units, any workloads capable of being shared on various processing units may be improved with the embodiment techniques. For example, work-stealing policies based on priority calculations as described herein may be used for improving the multi-processor computing device's overall performance of multi-threaded tasks and/or general-purpose calculations that may be discretely performed on a plurality of processing units.

[0053] Work-ready processors may be determined as “work-ready” (i.e., ready to steal work items from a victim processor) in various ways. In some embodiments, the multi-processor computing device may be configured to continually (or periodically) identify processing units and associated tasks that are ready to receive work items from other processing units (i.e., work-ready processors). The identification of work-ready processors may be accomplished in various manners, such as detecting application programming interface (API) calls from a processing unit to a runtime functionality indicating readiness to steal, setting a system variable associated with a processing unit, entering a processing unit's identifier in a general work stealing list, sending interrupts, etc. In some other embodiments, that utilize shared memory, each processing unit of the multi-processor computing device may independently realize a respective work-ready status (e.g., when a processing unit detects there are no more originally-assigned work items) and may concurrently perform operations to access a shared work-stealing data structure to steal work items as described herein (e.g., acquire a probabilistic lock, etc.).

[0054] For simplicity, references may be made to “work-ready processor queues” that may be listings of work-ready processors identified as ready to receive work items from task queues. Such a work-ready processor queue or other data structure may be associated with a particular processing

unit in the multi-processor computing device, and may be updated as work-ready processors are awarded work items. For example, data related to a pending work-ready task may be popped, dequeued, or otherwise removed from a work-ready processor queue when there is a successful transfer from a task queue. As shown with reference to FIGS. 2A-2D, a work-ready processor queue may be a data structure (e.g., an array, queue, linked list, etc.) stored in memory that may be used to track identifying data for any processing units (or associated tasks) that are available to receive work items from another processing unit.

[0055] However, references to such work-ready processor queues are not intended to limit the embodiments or claims to any particular implementation for identifying and/or otherwise managing one or more ready work-ready tasks. For example, in a shared-memory environment, processing units may simply compete for access to work-stealing data structures. In some embodiments, a runtime functionality may be configured to manage work-ready processor queues and/or any other similar mechanism for identifying work-ready tasks ready to receive work items from any particular processing unit. For example, instead of using work-ready processor queues, processing units may utilize various data structures, data containers, and/or other stored data to manage a collection of consecutive work items (e.g., iterations in a loop).

[0056] The embodiment techniques tag or otherwise assess the operating states (e.g., power consumption, frequency, temperature) for dynamic tasks in order to determine workload distributions and proactively balance power consumption and thermal dissipation in multi-processor systems (e.g., mobile devices using ARM's big.LITTLE architecture). These improvements work to fill a gap between parallel program execution and architecture level power and thermal management, and thus provide work-stealing solutions that also balance power consumption, thermal dissipation, and other operating parameters. Various embodiment techniques (e.g., via a control authority or runtime system) may use data regarding power consumption, frequency, and thermal states of the various tasks processing units to proactively balance workload distribution. The embodiment techniques calculate fine-grained, comprehensive priorities that include both local values, global values utilizing state information from across the processing units, and local measurements at a given time, such as via local schedulers with local core data (e.g., metrics, sampled data).

[0057] Some conventional thermal management schemes take actions to mitigate thermal conditions in response to detecting unacceptable temperatures (e.g., temperatures exceeding predefined thresholds). The embodiment techniques prevent thermal issues prior to encountering unacceptable conditions due to the work-stealing techniques that use temperature (and other operating state) data of processing units and/or tasks to identify where work items may be processed. Further, conventional techniques may utilize pre-defined priorities for various tasks or operations. The embodiment techniques utilize fine-grained data about any arbitrary tasks to identify relative priorities based on dynamic operating states of the tasks.

[0058] Thus, the various embodiments provide proactive task management techniques to effectively trade-off performance and power through work-stealing techniques in multi-core processor computing devices. Various embodi-

ments provide intelligent methods for selecting work-ready processors to receive work items from a victim processor in order to achieve overall load-balancing in a manner that considers execution speed, temperature and/or power states of the various processors.

[0059] Various embodiments may use global state information from across the processing units and local information on each processing unit to make more intelligent work-stealing decisions. Such embodiment techniques may be employed to improve performance due to load imbalances in different computing systems capable of implementing various work-stealing techniques. For example, embodiment methods may be used to improve performance in heterogeneous multi-core computing devices, homogeneous multi-core computing devices, servers or systems configured for distributed processing (e.g., a distributed computing server), etc. The various embodiment techniques are not specific to any type of parallelization system and/or implementation, but instead provide a manner of adding priorities to various work-stealing decisions, regardless of how numerous may implement work-stealing. Therefore, reference to any particular type or structure of multi-processor computing device (e.g., heterogeneous multi-processor computing device, etc.) and/or general work stealing implementation described herein is merely used for simplicity and is not intended to limit the embodiments or claims.

[0060] FIG. 1 is a diagram 100 illustrating various components of an exemplary heterogeneous multi-processor computing device 101 suitable for use with various embodiments. The heterogeneous multi-processor computing device 101 may include a plurality of processing units, such as a first CPU 102 referred to as "CPU_A" 102 in FIG. 1; a second CPU 112 referred to as "CPU_B" 112 in FIG. 1; a GPU 122, and a DSP 132. In some embodiments, the heterogeneous multi-processor computing device 101 may utilize an "ARM Big.Little" architecture, and the first CPU 102 may be a "big" processing unit having relatively high performance capabilities but also relatively high power requirements, and the second CPU 112 may be a "little" processing unit having relatively low performance capabilities but also relatively low power requirements compared to the first CPU 102.

[0061] The heterogeneous multi-processor computing device 101 may be configured to support parallel-processing, "work sharing", and/or "work stealing" between the various processing units 102, 112, 122, 132. In particular, any combination of the processing units 102, 112, 122, 132 may be configured to create and/or receive discrete tasks for execution.

[0062] Each of the processing units 102, 112, 122, 132 may utilize one or more queues (or task queues) for temporarily storing and re-executing tasks (and/or data associated with tasks) to be executed by the processing units 102, 112, 122, 132. For example, the first CPU 102 may retrieve tasks and/or task data from task queue 166, 168, 176 for local execution by the first CPU 102 and may place tasks and/or task data in queue 170, 172, 174 for execution by other devices. The second CPU 112 may retrieve tasks and/or task data from queue 174, 178, 180 for local execution by the second CPU 112 and may place tasks and/or task data in queue 170, 172, 176 for execution by other devices. The GPU 122 may retrieve tasks and/or task data from queue 172. The DSP 132 may retrieve tasks and/or task data from queue 170. In some embodiments, some task queues (70,

172, 174, 176) may be so-called multi-producer, multi-consumer queues, and some task queues 166, 168, 178, 180 may be so-called single-producer, multi-consumer queues. In some embodiments, tasks may be generated based on indicators within code, such as designations by programmers of workloads that split certain computations.

[0063] In some embodiments, a runtime functionality (e.g., runtime engine, task scheduler, etc.) may be configured to determine destinations for dispatching tasks to the processing units 102, 112, 122, 132. For example, in response to identifying a new general-purpose task that may be offloaded to any of the processing units 102, 112, 122, 132, the runtime functionality may identify a processing unit suitable for executing the task and may dispatch the task accordingly. Such a runtime functionality may be executed on an application processor or main processor, such as the first CPU 102. In some embodiments, the runtime functionality may be performed via one or more operating system-enabled threads (e.g., "main thread" 150). For example, based on determinations of the runtime functionality, the main thread 150 may provide task data to various task queues 166, 170, 172, 180.

[0064] FIGS. 2A-2D illustrate non-limiting examples of work-stealing operations by a heterogeneous multi-processor computing device 101 according to various embodiments. As described with reference to FIG. 1, the heterogeneous multi-processor computing device 101 may include a plurality of processing units, including a first CPU 102 (i.e., CPU_A in FIG. 2A-2D), a second CPU 112 (i.e., CPU_B in FIG. 2A-2D), a GPU 122, and a DSP 132. Each of the processing units 102, 112, 122, 132 may be associated with a task queue 220a-220d for managing tasks and related data (e.g., work items). For example, the first CPU 102 may be associated with a first task queue 220a, the second CPU 112 may be associated with a second task queue 220b, the GPU 122 may be associated with a third task queue 220c, and the DSP 132 may be associated with a fourth task queue 220d.

[0065] Each of the processing units 102, 112, 122, 132 may also be associated with a work-ready processor queue 240a-240d for managing tasks and related data. For example, the first CPU 102 may be associated with a first work-ready processor queue 240a, the second CPU 112 may be associated with a second work-ready processor queue 240b, the GPU 122 may be associated with a third work-ready processor queue 240c, and the DSP 132 may be associated with a fourth work-ready processor queue 240d.

[0066] Each of the work-ready processor queues 240a-240d may be used to indicate the identities of processing units (and/or related tasks) that are currently waiting to receive work items from particular task queues. For example, processing unit identities (or indices) within the first work-ready processor queue 240a may indicate that the processing units that are currently waiting contending to receive work items from the first task queue 220a associated with the first CPU 102.

[0067] In some embodiments, the work-ready processor queues 240a-240d may be stored within various data sources or structures (e.g., system variables, arrays, etc.) that may or may not be directly associated with the individual processing units 102, 112, 122, 132. For example, the work-ready processor queues 240a-240d may be represented by the same queue-type data structure that is used for organizing any processing unit or task currently ready to receive work items from other tasks. In some embodiments, the task

queues 220*a*-220*b* and/or the work-ready processor queues 240*a*-240*d* may be discrete components (e.g., memory units) corresponding to the processing units 102, 112, 122, 132 and/or ranges of memory within various memory units (e.g., system memory, shared memory, virtual memory, etc.).

[0068] In some embodiments, the heterogeneous multi-processor computing device 101 may include a runtime functionality module 210, such as an operating system (OS) level process, service, application, routine, instruction set, or thread that executes via an application processor (e.g., CPU₁-3 102). The runtime functionality module 210 may be configured to at least perform operations for managing work-stealing policies within the heterogeneous multi-processor computing device 101. For example, the runtime functionality module 210 may be configured to add and/or remove identifiers from the work-ready processor queues 240*a*-240*d* in response to determining tasks queues 220*a*-220*b* are empty, as well as perform operations for evaluating operating states and determining relative priorities between processing units and/or associated tasks.

[0069] FIG. 2A shows a diagram 200 illustrating exemplary work-stealing operations in the heterogeneous multi-processor computing device 101. At an arbitrary first time, the task queue 220*a* for the first CPU 102 has a plurality of work items 230*a* and the task queue 220*b* for the DSP 132 has a plurality of work items 230*b*. However, both the second CPU 112 (i.e., CPU B in FIGS. 2A-2D) and the GPU 122 have no work items within respective task queues 220*c*, 220*d*, thus making the second CPU 112 and the GPU 122 ready (or eligible) to receive work items from other processing units. For example, when the processing units 102, 112, 122, 132 each were assigned work items for processing iterations of a parallel loop running the second CPU 112 and the GPU 122 may have already completed assigned iterations and may be ready to receive iterations from the first CPU 102 and/or the DSP 132. Upon becoming ready to receive work items from other processing units, data identifying the second CPU 112 and the GPU 122 may be entered into the work-ready processor queues of other processing units that have work items waiting to be processed. For example, identifying data for the second CPU 112 (i.e., data 242*a*) and identifying data for the GPU 122 (i.e., data 252*a*) may be placed in the work-ready processor queue 240*a* of the first CPU 102.

[0070] In various embodiments, work-ready processors ready to receive work items may contend to receive work items from one or a plurality of task queues of processing units of a system. In other words, there may be more than one potential victim processor at a given time. For example, instead of competing to receive work items from the task queue 220*a* of the first CPU 102, the second CPU 112 and the GPU 122 may instead compete to receive work items from the task queue 220*b* of the DSP 132 by being entered in the work processor queue 240*a* of the DSP 132. As another example, instead of competing to receive work items from the task queue 220*a* of the first CPU 102, the second CPU 112 may be entered in the work-ready processor queue 240*a* of the first CPU 102 and the GPU 122 may be entered in the work-ready processor queue 240*b* of the DSP 132. However, in various embodiments, a work-ready processor may only compete in one work-ready processor queue at any given time. Further, in various embodiments, the results of a competition between a plurality of work-ready processor to receive work items from a task queue of a victim processor

may not affect the results of subsequent competitions for the same victim processor and/or other victim processors. For example, if the second CPU 112 is awarded work items 230*a* from the first task queue 220*a* over the GPU 122 at a first time, the second CPU 112 may not necessarily be assigned work items 230*a* from the fourth task queue 220*d* over the GPU 122 at a second time.

[0071] FIG. 2B shows a diagram 260 illustrating an example of work-stealing operations at a second time after the first time of FIG. 2A. At the second time, the runtime functionality module 210 may transfer work items 230*a* from the task queue 220*a* of the first CPU 102 to the task queue 220*b* of the GPU 122 based on priority calculations as described herein (e.g., based on temperature, power consumption, and/or frequency). For example, in response to detecting that the second CPU 112 and the GPU 122 are both ready to receive from the first CPU 102 (i.e., identifying data 242*a*, 252*a* in the first work-ready processor queue 240*a*), the runtime functionality module 210 may calculate a relative priority between the second CPU 112 and the GPU 122 that addresses (or is based at least partly upon) the temperature, power consumption, and/or frequency for each processing unit 112, 122. The runtime functionality module 210 may determine that the GPU 122 is the winning work-ready processor over the second CPU 112 based on a higher priority value for the GPU 122 at the second time. Thus, in the illustrated example, as the second CPU 112 is calculated to have a lower priority for stealing purposes, no work items may be transferred to the task queue 220*b* of the second CPU 112 at the second time. However, due to the dynamic nature of the stealing priority calculations, it is possible that the second CPU 112 may have a higher stealing priority than the GPU 122 at other times.

[0072] Once the GPU 122 is determined the winning work-ready processor, the runtime functionality module 210 may also make a determination as to the amount of work to be reassigned to the GPU 122. For example, the runtime functionality module 210 may also calculate relative priorities of the first CPU 102 and the GPU 122 as described herein. Based on these relative priorities, the runtime functionality module 210 may determine that two of the original four pending work items 230*a* in the task queue 220*a* of the first CPU 102 may be transferred to the task queue 220*b* of the GPU 122.

[0073] After the transfer (or reassignment) of work items, the runtime functionality module 210 may remove the identifying data 252*a* associated with the winning work-ready processor (i.e., the GPU 122) from the work-ready processor queue 240*a* of the first CPU 102. In some embodiments, the runtime functionality module 210 may remove all identifying data associated with the winning work-ready processor from all work-ready processor queues 240*a*, 240*b* in response to transferring work items 230*a*. In some embodiments, all identifying data 242*a*, 252*a*, 242*b*, 252*b* for any competing work-ready processors may be removed from work-ready processor queues 240*a*, 240*b* in response to awarding work items to a winning work-ready processor, thus forcing ready work-ready processors to reassess readiness to receive work items. For example, after the GPU 122 wins the work items 230*a* based on priority calculations by the runtime functionality module 210, the second CPU 112 may enter new identifying data 242*a*, 242*b* into the work-ready processor queues 240*a*, 240*b* of processing units 102, 132 having available work items to steal.

[0074] FIG. 2C shows a diagram 270 illustrating exemplary work-stealing operations at a third time after the second time of FIG. 2B. At the third time, the runtime functionality module 210 may transfer work items 230a' from the task queue 220a of the first CPU 102 to the second CPU 112. In other words, after a first award of work items 230a to the GPU 122, the runtime functionality module 210 may perform subsequent work-stealing calculations to determine whether remaining work-ready processors in the work-ready processor queues 240a, 240b may also be awarded work items. For example, in response to detecting identifying data 242a of the second CPU 112 in the work-ready processor queue 240b of the first CPU 102, the runtime functionality module 210 may calculate relative priorities of the processing units 102, 112 to determine that the unawarded second CPU 112 may receive a work item 230a.

[0075] FIG. 2D shows a diagram 280 illustrating an alternative embodiment in the work-stealing operations at the third time in which the runtime functionality module 210 may not award any work item 230a from the task queue 220a of the first CPU 102 to the second CPU 112 based on priority calculations. For example, the runtime functionality module 210 may perform priority calculations based on power consumption, temperature, and/or frequency and determine a ratio for steals/stealing work items 230a that provides no work items to the second CPU 112. In this case, based on priority, the second CPU 112 may have had a lower probability of winning work items 230a from the first CPU 102. Thus, the runtime functionality module 210 may refuse to transfer work items 230a to the second CPU 112. For example, based on a high current temperature at the second CPU 112, the runtime functionality module 210 may determine that first CPU 102 is a better option over the second CPU 112 for furnishing the work items 230a remaining in the task queue 220a.

[0076] Different work-stealing implementations may benefit from the embodiment priority calculations as described. For example, some computing environments that use queues (e.g., as shown in FIGS. 2A-2D) may employ embodiment techniques for calculating work-ready processor priorities and associated allocations of work items from task queues. As another example, in a shared memory environment, a multi-processor computing device may be configured to manage the reallocation of parallel processing workloads using a work-stealing data structure that corresponds to work items associated with various segments of the shared memory. FIG. 3 illustrates example work-stealing operations by a multi-processor computing device 301 having a shared memory 306 and work-stealing data structure 304 according to some embodiments. Unlike the example system illustrated in FIGS. 2A-2D, the multi-processor computing device 301 of FIG. 3 may be a high-performance implementation of work-stealing that does not use queues, but instead uses the shared work-stealing data structure 304 that controls access to various work items of a shared task.

[0077] As shown in FIG. 3, whenever processing units complete assigned workloads for the shared task, the processing unit units may become ready to steal from the other processing units (e.g., become work-ready processor). For example, both the second CPU 112 and GPU 122 may independently determine themselves to be "ready to steal". The second CPU 112 and the GPU 122 may then compete for next to steal work items from either the first CPU 102 or the DSP 132, which are still processing original work

items of the shared task. In some embodiments, both the second CPU 112 and the GPU 122 may compete to acquire control over the work-stealing data structure 304 via a priority-based probabilistic lock mechanism 302. The probability of either processing unit 112, 122 gaining control over the priority-based probabilistic lock mechanism 302 may depend on the respective priority values (e.g., values calculated via I equation 1 as described). The one processing unit to gain control over the probabilistic lock may thus gain modification access to the work-stealing data structure 304 as the "winning" work-ready processor. With access to update or otherwise change the data within the work-stealing data structure 304, the winning work-ready processor may assign work items from other processing units (e.g., first CPU 102, DSP 132) and then retrieve work items directly from the shared memory 306.

[0078] FIG. 4A illustrates a method 400 for pivotively balancing workloads between a plurality of processing units by making work-stealing determinations based on operating state data. As described, a multi-processor computing device (e.g., multi-processor computing device 101, 301 in FIGS. 1-2), 3 may be configured to enable improved work-stealing by calculating priorities of work-ready processors and victim processors with respect to power consumption, temperature, and/or frequency operating state data. Once one or more work-ready processors are identified as ready for stealing work items from a victim processor, the multi-processor computing device may calculate priorities by evaluating static characteristics of the processing units involved (e.g., processing units of work-ready tasks, processing unit of a victim task) and dynamic operating conditions of the processing units. Such comprehensive priority values may be used to determine relative rankings between work-ready processor as well as work item distributions to a winning work-ready processor. In this way, the multi-processor computing device may employ fine-grained operating data about different processing units and associated tasks to direct dynamic work-stealing decisions, improving system efficiency and performance.

[0079] In various embodiments, the method 400 may be performed for each processing unit of the multi-processor computing device. For example, the multi-processor computing device may concurrently execute one or more instances of the method 400 (e.g., one or more threads for executing the method 400) to handle work stealing for tasks of each processing unit. In some embodiments, various operations of the method 400 may be performed by a runtime functionality (e.g., runtime functionality module 210 of FIGS. 2A-2D) executing in a processing unit of the multi-processor computing device (e.g., via an application processor, the CPU 102 of the multi-processor computing device 101 of FIGS. 1-2), etc.).

[0080] Referring to the method 400, the multi-processor computing device may determine whether there are any work items available to be transferred or "stolen" from a first processing unit and/or an associated task in determining block 402. In other words, the multi-processor computing device may determine whether the first processing unit and/or the associated task is to be treated as a victim processor at a given time. For example, the multi-processor computing device may determine whether the first processing unit and/or the associated task is executing work items that are capable of being executed elsewhere (e.g., iterations of a parallel loop routine) and/or whether there are any work

means that the first processing unit is not already processing (or scheduled to process in the near future). Such a determination may be made by the multi-processor computing device by evaluating the task queue of the first processing unit, such as by querying the number and/or type of currently pending work items. In some embodiments, the first processing unit and/or the associated task may only be a victim processor when the number of pending work items in the corresponding task queue exceeds a threshold number. For example, the multi-processor computing device may identify a maximum number of work items needed to make any migration of work items from the first processing unit to another processing unit beneficial. Such a threshold may be calculated based on a comparison of an estimated first time required to perform a computation (e.g., process a certain number of work items) locally at the first processing unit and an estimated second time required to perform the computation at another processing unit.

[0081] In some embodiments, the multi-processor computing device may calculate a priority as described above to determine whether the first processing unit and/or the associated task may be stolen from. For example, when the first processing unit's relative priority is higher than other processing units, the multi-processor computing device may determine that no other processing unit is capable of stealing from the first processing unit regardless of the availability of work items, and thus may forgo any further operations to allow work-stealing from the first processing unit for a time period.

[0082] In response to determining that there are no work items available to be stolen from the first processing unit and/or the associated task (i.e., determination block 402 "No"), the first processing unit and/or the associated task may not be a victim processor at the current time, and thus the multi-processor computing device may continue with the victim processor determination operations in determination block 402. In some embodiments, the victim processor determinations of determination block 402 may be performed immediately or after a predefined period of time (e.g., after the elapse of a predetermined wait period).

[0083] In response to determining that there is at least one work item available to be stolen from the first processing unit and/or the associated task (i.e., determination block 402 "Yes"), the first processing unit and/or the associated task may be determined to be a victim processor at the current time. In optional block 404, the multi-processor computing device may identify any work-ready processors (e.g., work-ready processing unit(s), work-ready task(s)) that are ready to receive work items from the victim processor. For example, the multi-processor computing device may evaluate various API calls, interrupts, and/or stored data (e.g., system variables, data within work-ready processor queues, etc.) that indicate one or more work-ready processors have requested or are otherwise ready to be assigned work items from the victim processor. The operations in optional block 404 may be optional as the operations may apply to embodiments that rely upon "work-ready queues" as described herein, and thus the operations of optional block 404 may not be applicable in other embodiments that utilize a shared memory and work-stealing data structure as described.

[0084] In some embodiments, work-ready processors may specifically request to receive work items from particular victim processor and/or may request to receive from any

available victim processor. For example, a ready work-ready processor may be placed within a work-ready processor queue for a specific processing unit (e.g., a GPU's work-ready processor queue, etc.). As another example, a ready work-ready task may be placed within a general queue for receiving work items from any task queue of any processing unit.

[0085] The multi-processor computing device may determine whether there are one or more work-ready processors ready to receive work items from the victim processor in determination block 406. In response to determining that there are no work-ready processors ready to receive from the victim processor (i.e., determination block 406 "No"), the multi-processor computing device may continue with the victim processor determination operations in determination block 402.

[0086] In response to determining there is at least one work-ready processor ready to receive from the victim processor (i.e., determination block 406 "Yes"), the multi-processor computing device may obtain static characteristics data associated with the victim processor and the one or more work-ready processors in block 408. Such static characteristics may include information describing attributes/capabilities of the processing units corresponding to the work-ready processors and victim processor. For example, static characteristics data for each work-ready processor and the victim processor may include data indicating a technology used (e.g., architecture, special abilities, operating specifications and/or thresholds, etc.), a location in the topology of the multi-processor computing device (e.g., near a modem, etc.), and/or manufacturer data (e.g., age/install date, etc.). In some embodiments, the static characteristics data may be obtained by the multi-processor computing device once at the time of manufacture of the various processing units or the multi-processor computing device, at each boot-up of the multi-processor computing device, on a periodic basis (e.g., daily, weekly, monthly, etc.), or any combination thereof. In some embodiments, obtaining the static characteristics may include identifying values for global coefficients (e.g., $\alpha_1, \beta_1, \gamma_1$) used in Equation 1 (i.e., $P_i = \alpha_1 * Temp + \beta_1 * Power + \gamma_1 * Threq$) as described above.

[0087] In block 410, the multi-processor computing device may obtain dynamic characteristics data associated with the victim processor and the one or more work-ready processors. The dynamic characteristics data may include information related to the current, recent, and/or historical execution of operations on the various processing units. For example, the dynamic characteristics may include average, minimum, maximum, current, and/or projected values for power consumption, frequency, and/or temperature for the various processing units associated with the victim processor and the one or more work-ready processors. In some embodiments, obtaining the dynamic characteristics may include identifying values for local coefficients (e.g., $\alpha_2, \beta_2, \gamma_2$) and/or local measurements (e.g., Temp, Power, Freq) used in Equation 1 (i.e., $P_i = \alpha_2 * Temp + \beta_2 * Power + \gamma_2 * Freq$) as described above.

[0088] In block 412, the multi-processor computing device may calculate priority values for the victim processor and the one or more work-ready processors based on the obtained data. For example, the multi-processor computing device may use obtained values for global coefficients, local coefficients, and local measurements related to the various

processing units of the victim processor and one or more work-ready processors to calculate a relative priority value for each using the Equation 1 (i.e., $P_i = \alpha_1^{1.7}(\text{temp}_i + 15.5)^{1.7}(\text{Power}_i)^{1.7}(\text{freq}_i)$) as described above. In other words, the multi-processor computing device may balance data of operating states (e.g., temperature, power consumption, etc. frequency factors) to calculate relative priorities of the victim processor and one or more work-ready processors.

[0089] In determination block 414, the multi-processor computing device may determine whether more than one work-ready processor is competing to receive work items from the victim processor. For example, when there are more than one work-ready processors, the resulting competition for the victim processor's work items may require a selection or determination of a winning work-ready processor that may receive work items from the victim processor.

[0090] In response to determining that there is more than one work-ready processor (i.e., determination block 414 "Yes"), the multi-processor computing device may identify a winning work-ready processor among the plurality of work-ready processors based on the calculated priority values in block 416. For example, the multi-processor computing device may simply compare the calculated relative priority values to find the highest value (i.e., the highest priority). Determining a winning work-ready processor with the calculated priorities may not rely on randomness or timing (e.g., the time at which the plurality of work-ready processors become ready to steal). Instead, the winning work-ready processor may be determined based on the ability of the processor to process work items given various operating conditions of the system. In some embodiments, the multi-processor computing device may identify more than one winning work-ready processor, such as a first winning work-ready processing unit and a second winning work-ready processing unit. A plurality of winning work-ready processors may be possible when the victim processor has an abundance of available work items to steal, the calculated priorities of the work-ready processors are within a certain threshold of each other, and/or other factors that make multiple recipients of work items feasible at a given time. In some embodiments, the multi-processor computing device may determine percentages of allocations for the winning work-ready processors, such as by using Equations 5-6 as described that may assign work items in proportion to priorities.

[0091] In response to determining that there is only one work-ready processor (i.e., determination block 414 "No"), or in response to performing the identified-in operations in block 416, the multi-processor computing device may calculate a number of work items to transfer to the winning work-ready processor (or the only work-ready processor) based on the calculated priority values of the victim processor and winning work-ready processor for only work-ready processor in block 418. For example, when the victim processor has a plurality of work items available to be stolen, the multi-processor computing device may determine a first number of the plurality of work items that may remain for execution by the victim processing unit and a second number of work items that may be transferred to the work-ready processor. In some embodiments, the multi-processor computing device may calculate the number of work items to transfer to the winning work-ready processor using Equations 5-6 as described above.

[0092] In block 420, the multi-processor computing device may reassign the calculated number of work items from the victim processor to the winning work-ready processor (or only work-ready processor). For example, the multi-processor computing device may transfer one or more work items and/or related data to the task queue(s) of a winning work-ready processing unit from the task queue(s) of the victim processor. As another example, the multi-processor computing device may enable the winning work-ready processor to adjust data in a shared work-stealing data structure to indicate that work items in a shared memory that were previously assigned to the victim processor are now assigned to the winning work-ready processor. In some embodiments, the reassignment of work items between task queues may include data transfers between queues and/or assignments of access to particular data, such as via a check-out or assignment procedure for a shared memory unit.

[0093] The multi-processor computing device may perform the operations of the method 400 in a continuous or periodic loop, returning again to determining whether there are any work items available to be stolen from a victim processor at determination block 402. With each iteration, the processor units that are work-ready and the victim processor may change as the workload of each processing unit very with time. Thus, a work-ready processing unit at one time may later become a victim processor as its queue of work items increases.

[0094] As described, embodiment methods may be implemented in various work-stealing and/or data parallelism frameworks. For example, embodiment priority calculations may be used for processing units that utilize task queues and/or access shared memory to manage assigned work items. FIG. 4B illustrates an embodiment method 450 that includes operations for a multi-processor computing device that utilizes a shared memory and shared work-stealing data structure implementation.

[0095] Some operations of the method 450 may be similar to the method 400 and may be performed for each processing unit of the multi-processor computing device. For example, the operations of blocks 402, 406-418 may be similar to the operations of like numbered blocks described above with reference to FIG. 4A. Further, in some embodiments, the multi-processor computing device may concurrently execute one or more instances of the method 450 (e.g., one or more threads for executing the method 400) to handle work stealing for tasks of each processing unit. In some embodiments, various operations of the method 450 may be performed by a runtime functionality (e.g., runtime functionality module 210 of FIGS. 2A-2D), executing in a processing unit of the multi-processor computing device (e.g., via an application processor, such as the CPU 102 of the multi-processor computing device 101 of FIGS. 1-2), etc.).

[0096] In general, whenever a task (or associated processing unit) becomes free by completing originally-assigned work items of a shared task workload, that processing unit (or work-ready processor) may then contend to steal work items by updating a shared work-stealing data structure. A victim processor may be identified from the work-stealing data structure as part of such an access modification process. In various embodiments, probabilistic locks (or atomics) may be used to deal with concurrent accesses to the shared data structure by multiple work-ready processing units.

[0097] Referring to FIG. 4B, in response to performing the operations of determination block 414 (i.e., determination block 414 "Not" or block 416), the multi-processor computing device may acquire, via a winning processor, control over a probabilistic lock for a shared data structure based on the calculated priority value of the winning work-ready processor in block 452. In block 454, the multi-processor may update, via the winning processor, the shared data structure to indicate the number of work items transferred to the winning work-ready processor in response to regaining the control over the probabilistic lock.

[0098] In block 456, the multi-processor may release, via the winning processor, control over the probabilistic lock, and thus also free the work-sharing data structure to be potentially accessed by other processing units. The multi-processor computing device may then continue with the determination operations in determination block 402.

[0099] Various forms of computing devices, including personal computers, mobile devices, and laptop computers, may be used to implement the various embodiments. Such computing devices may typically include the components illustrated in FIG. 5, which illustrates an example multi-processor mobile device 500. In various embodiments, the mobile device 500 may include a processor 501 coupled to a touch screen controller 504 and an internal memory 502. The processor 501 may include a plurality of multi-core IC's designated for general or specific processing tasks. In some embodiments, other processing units may also be included and coupled to the processor 501 (e.g., GPU, DSP, etc.).

[0100] The internal memory 502 may be volatile and/or non-volatile memory, and may also be secure and/or encrypted memory, or unsecure and/or unencrypted memory, or any combination thereof. The touch screen controller 504 and the processor 501 may also be coupled to a touch screen panel 512, such as a resistive-sensing touch screen, capacitive-sensing touch screen, infrared sensing touch screen, etc. The mobile device 500 may have one or more radio signal transceivers 508 (e.g., Bluetooth®, Zig-Bee®, Wi-Fi®, RF radio) and antenna 510, for sending and receiving, coupled to each other and/or to the processor 501. The transceivers 508 and antenna 510 may be used with the above-mentioned circuitry to implement the various wireless transmission protocol stacks and interfaces. The mobile device 500 may include a cellular network wireless modem chip 516 that enables communication via a cellular network, and is coupled to the processor 501. The mobile device 500 may include a peripheral device connection interface 518 coupled to the processor 501. The peripheral device connection interface 518 may be singularly configured to accept one type of connection, or multiple, configured to accept various types of physical and communication connections, common or proprietary, such as USB, FireWire, Thunderbolt, or PCIe. The peripheral device connection interface 518 may also be coupled to a similarly configured peripheral device connection port (not shown). The mobile device 500 may also include speakers 514 for providing audio outputs. The mobile device 500 may also include a housing 520, constructed of a plastic, metal, or a combination of materials, for containing all or some of the components described herein. The mobile device 500 may include a power source 522 coupled to the processor 501, such as a disposable or rechargeable battery. The rechargeable battery may also be

coupled to the peripheral device connection port to receive a charging current from a source external to the mobile device 500.

[0101] Various processors for processing units(s) described herein may be any programmable microprocessor, micro-computer or multiple processor chip or chips that may be configured by software instructions (application(s)) to perform a variety of functions, including the functions of the various embodiments described herein. In the various devices, multiple processors may be provided, such as one processor dedicated to wireless communication functions and one processor dedicated to running other applications. Typically, software applications may be stored in internal memory before they are accessed and loaded into the processors. The processors may include internal memory sufficient to store the application software instructions. In many devices the internal memory may be a volatile or nonvolatile memory, such as flash memory, or a mixture of both. For the purposes of this description, a general reference to memory refers to memory accessible by the processors, including internal memory or removable memory plugged into the various devices and memory within the processors.

[0102] The foregoing method descriptions and the process flow diagrams are provided merely as illustrative examples and are not intended to require or imply that the operations of the various embodiments must be performed in the order presented. As will be appreciated by one of skill in the art the order of operations in the foregoing embodiments may be performed in any order. Words such as "thereafter," "then," "next," etc., are not intended to limit the order of the operations; these words are simply used to guide the reader through the description of the methods. Further, any reference to claim elements in the singular, for example, using the articles "a," "an" or "the," is not to be construed as limiting the element to the singular.

[0103] The various illustrative logical blocks, modules, circuits, and algorithm operations described in connection with the embodiments disclosed herein may be implemented as electronic hardware, computer software, or combinations of both, to clearly illustrate this interchangeability of hardware and software, various illustrative components, blocks, modules, circuits, and operations have been described above generally in terms of their functionality. Whether such functionality is implemented as hardware or software depends upon the particular application and design constraints imposed on the overall system. Skilled artisans may implement the described functionality in varying ways for each particular application, but such implementation decisions should not be interpreted as causing a departure from the scope of the present claims.

[0104] The hardware used to implement the various illustrative logics, logical blocks, modules, and circuits described in connection with the embodiments disclosed herein may be implemented or performed with a general purpose processor, a digital signal processor (DSP), an application specific integrated circuit (ASIC), a field programmable gate array (FPGA) or other programmable logic device, discrete gate or transistor logic, discrete hardware components, or any combination thereof designed to perform the functions described herein. A general-purpose processor may be a microprocessor, but, in the alternative, the processor may be any conventional processor, controller, micro-controller, or state machine. A processor may also be

implemented as a combination of computing devices, e.g., a combination of a DSP and a microprocessor; a plurality of microprocessors, one or more microprocessors in conjunction with a DSP core, or any other such configuration. Alternatively, some operations or methods may be performed by circuitry that is specific to a given function.

[0105] In one or more exemplary embodiments, the functions described may be implemented in hardware, software, firmware, or any combination thereof. If implemented in software, the functions may be stored on or transmitted over as one or more instructions or code on a non-transitory processor-readable, computer-readable, or server-readable medium, or a non-transitory processor-readable storage medium. The operations of a method or algorithm disclosed herein may be embodied in a processor-executable software module, or processor-executable software instructions, which may reside on a non-transitory computer readable storage medium, or a non-transitory server-readable storage medium, and/or a non-transitory processor-readable storage medium. In various embodiments, such instructions may be stored processor-executable instructions or stored processor-executable software instructions. Tangible, non-transitory computer-readable storage media may be any available media that may be accessed by a computer. By way of example, and not limitation, such non-transitory computer-readable media may comprise RAM, ROM, EPROM, CD-ROM or other optical disk storage, magnetic disk storage, or other magnetic data storage devices, or any other medium that may be used to store desired program code in the form of instructions or data structures and that may be accessed by a computer. Disk and disc, as used herein, includes compact disc (CD), laser disc, optical disc, digital versatile disc (DVD), floppy disk, and Blu-ray Disc®, where disks initially reproduce data magnetically, while discs reproduce data optically with lasers. Combinations of the above should also be included within the scope of non-transitory computer-readable media. Additionally, the operations of a method or algorithm may reside as one or any combination of set of codes and/or instructions on a tangible, non-transitory processor-readable storage medium, and/or computer-readable medium, which may be incorporated into a computer program product.

[0106] The preceding description of the disclosed embodiments is provided to enable any person skilled in the art to make or use the embodiment techniques of the claims. Various modifications to these embodiments will be readily apparent to those skilled in the art, and the generic principles defined herein may be applied to other embodiments without departing from the scope of the claims. Thus, the present disclosure is not intended to be limited to the embodiments shown herein but is to be accorded the widest scope consistent with the following claims and the principles and novel features disclosed herein.

1. A method implemented by a processor of a multi-processor computing device to principally balance workloads among a plurality of processing units by making work-stealing determinations based on operating state data, comprising:

obtaining static characteristics data associated with each of a victim processor and one or more of the plurality of processing units that are ready to steal work items from the victim processor (work-ready processors);

obtaining dynamic characteristics data including values associated with temperature and power consumption, associated with each of the victim processor and the work-ready processors;

calculating priority values for each of the victim processor and the work-ready processors based on the obtained data; and

transferring a number of work items assigned to the victim processor to a winning work-ready processor based on the calculated priority values.

2. The method of claim 1, wherein the work-ready processors are ready to steal the work items from the victim processor in response to the work-ready processors finishing respective assigned work items.

3. The method of claim 1, wherein the static characteristics data associated with each of the victim processor and the work-ready processors include any of data indicating a technology used, a location in a topology of the multi-processor computing device, and manufacturer data.

4. The method of claim 1, wherein the static characteristics data is obtained at a time of manufacture of the multi-processor computing device, at each bootup of the multi-processor computing device, on a periodic basis, or any combination thereof.

5. The method of claim 1, wherein the dynamic characteristics data associated with each of the victim processor and the work-ready processors includes local measurements of each of the victim processor and the work-ready processors.

6. The method of claim 1, wherein the dynamic characteristics data associated with each of the victim processor and the work-ready processors further includes values associated with frequency.

7. The method of claim 1, wherein calculating the priority values for each of the victim processor and the work-ready processors based on the obtained data comprises calculating the priority values using the following equation:

$$P_i = \alpha_1 \cdot \text{Temp}_i + \alpha_2 \cdot \text{Power}_i + \alpha_3 \cdot \text{Freq}_i$$

wherein i represents an index for one of the victim processor and the work-ready processors, P_i represents a calculated priority value for a processing unit associated with the index for one of the victim processor and the work-ready processors, α_1 represents a global coefficient for temperature, α_2 represents a local coefficient for the temperature for the processing unit, Temp_i represents a formula over a local temperature measurement for the processing unit, α_3 represents a global coefficient for power consumption, Power_i represents a local coefficient for the power consumption for the processing unit, Power represents a formula over a local power consumption measurement for the processing unit, α_4 represents a global coefficient for frequency, Freq_i represents a local coefficient for the frequency of the processing unit, and Freq represents a formula over a local frequency measurement for the processing unit.

8. The method of claim 7, wherein obtaining the static characteristics data associated with each of the victim processor and the work-ready processors comprises obtaining the global coefficient for temperature, the global coefficient for power consumption, and the global coefficient for frequency.

9. The method of claim 7, wherein obtaining dynamic characteristics data associated with each of the victim pro-

cessor and the work-ready processors comprises obtaining the local coefficient for the temperature for the processing unit, the local temperature measurement for the processing unit, the local coefficient for the power consumption for the processing unit, the local power consumption measurement for the processing unit, the local coefficient for the frequency of the processing unit, and the local frequency measurement for the processing unit.

10. The method of claim 1, further comprising calculating the number of work items to transfer to the winning work-ready processor based on the calculated priority values of the victim processor and the winning work-ready processor.

11. The method of claim 10, wherein calculating the number of work items to transfer to the winning work-ready processor based on the calculated priority values of the victim processor and the winning work-ready processor comprises calculating the number of work items to transfer using the following equations:

$$N_1 = P_1 / (P_1 + P_2)$$

$$N_2 = N - N_1$$

wherein i represents a first task index associated with the victim processor, j represents a second task index associated with the winning work-ready processor, K represents a total number of work items remaining to be processed by the victim processor prior to steading, P_1 represents a calculated priority value of the victim processor, and P_2 represents the calculated priority value of the winning work-ready processor, N_1 represents the number of work items to transfer to the winning work-ready processor, and N_2 represents a number of work items to remain with the victim processor.

12. The method of claim 1, wherein each of the victim processor and the work-ready processors is associated with a different processing unit in the plurality of processing units.

13. The method of claim 1, wherein each of the victim processor and the work-ready processors is associated with a different task executing on one of the plurality of processing units.

14. The method of claim 1, wherein the work items are different iterations of a parallel loop routine.

15. The method of claim 1, wherein the plurality of processing units includes two or more of a first central processing unit (CPU), a graphics processing unit (GPU), and a digital signal processor (DSP).

16. The method of claim 1, wherein the multi-processor computing device is one of a heterogeneous multi-core computing device, a homogeneous multi-core computing device, a distributed computing server.

17. The method of claim 1, further comprising: acquiring, via the winning work-ready processor, control over a probabilistic lock for a shared data structure based on a calculated priority value of the winning work-ready processor; and updating, via the winning work-ready processor, the shared data structure to indicate the number of work items transferred to the winning work-ready processor in response to acquiring control over the probabilistic lock.

18. A computing device, comprising:
a memory; and
a processor of a plurality of processing units, wherein the processor is coupled to the memory and is configured with processor-executable instructions to perform operations comprising:
obtaining static characteristics data associated with each of a victim processor and one or more of the plurality of processing units that are ready to steal work items from the victim processor (work-ready processors);
obtaining dynamic characteristics data including values associated with temperature and power consumption associated with each of the victim processor and the work-ready processors;
calculating priority values for each of the victim processor and the work-ready processors based on the obtained data; and
transferring a number of work items assigned to the victim processor to a winning work-ready processor based on the calculated priority values.

19. The computing device of claim 18, wherein the static characteristics data associated with each of the victim processor and the work-ready processors include any of data indicating a technology used, a location in a topology of the computing device, and manufacturer data.

20. The computing device of claim 18, wherein the static characteristics data is obtained at a time of manufacture of the computing device, at each boot-up of the computing device, on a periodic basis, in any combination thereof.

21. The computing device of claim 18, wherein the dynamic characteristics data associated with each of the victim processor and the work-ready processors includes local measurements of each of the victim processor and the work-ready processors.

22. The computing device of claim 18, wherein the dynamic characteristics data associated with each of the victim processor and the work-ready processors further includes values associated with frequency.

23. The computing device of claim 18, wherein the processor is configured with processor-executable instructions to perform operations such that calculating the priority values for each of the victim processor and the work-ready processors based on the obtained data comprises calculating the priority values using the following equation:

$$P_i = \alpha_1 T_{local,i} + \alpha_2 P_{global,i} + \alpha_3 f_{local,i}$$

wherein i represents an index for one of the victim processor and the work-ready processors, P_i represents a calculated priority value for a processing unit associated with the index for one of the victim processor and the work-ready processors, α_1 represents a global coefficient for temperature, α_2 represents a local coefficient for the temperature for the processing unit, $T_{local,i}$ represents a formula over a local temperature measurement for the processing unit, α_3 represents a global coefficient for power consumption, α_3 represents a local coefficient for the power consumption for the processing unit, $P_{global,i}$ represents a formula over a local power consumption measurement for the processing unit, $f_{local,i}$ represents a global coefficient for frequency, $f_{local,i}$ represents a local coefficient for the frequency of the processing unit, and $f_{local,i}$ represents a formula over a local frequency measurement for the processing unit.

24. The computing device of claim **23**, wherein the processor is configured with processor-executable instructions to perform operations such that obtaining the static characteristics data associated with each of the victim processor and the work-ready processors comprises obtaining the global coefficient for temperature, the global coefficient for power consumption, and the global coefficient for frequency.

25. The computing device of claim **23**, wherein the processor is configured with processor-executable instructions to perform operations such that obtaining dynamic characteristics data associated with each of the victim processor and the work-ready processors comprises obtaining the local coefficient for the temperature for the processing unit, the local temperature measurement for the processing unit, the local coefficient for the power consumption for the processing unit, the local power consumption measurement for the processing unit, the local coefficient for the frequency of the processing unit, and the local frequency measurement for the processing unit.

26. The computing device of claim **18**, wherein the processor is configured with processor-executable instructions to perform operations further comprising calculating the number of work items to transfer to the winning work-ready processor based on the calculated priority values of the victim processor and the winning work-ready processor.

27. The computing device of claim **26**, wherein the processor is configured with processor-executable instructions to perform operations such that calculating the number of work items to transfer to the winning work-ready processor based on the calculated priority values of the victim processor and the winning work-ready processor comprises calculating the number of work items to transfer using the following equations:

$$N_i = K^* P_i / P_w P_v$$

$$N_j = N - N_i$$

wherein i represents a first task index associated with the victim processor, j represents a second task index associated with the winning work-ready processor, K represents a total number of work items remaining to be processed by the victim processor prior to steading, P_i represents a calculated priority value of the victim processor, and P_w represents the calculated priority value of the winning work-ready processor, N represents the number of work items to transfer to the winning work-ready processor, and N_j represents a number of work items to remain with the victim processor.

28. The computing device of claim **18**, wherein the processor is configured with processor-executable instructions to perform operations further comprising:

acquiring, via the winning work-ready processor, control over a probabilistic lock for a shared data structure based on a calculated priority value of the winning work-ready processor; and
updating, via the winning work-ready processor, the shared data structure to indicate the number of work items transferred to the winning work-ready processor in response to acquiring control over the probabilistic lock.

29. A non-transitory processor-readable storage medium having stored thereon processor-executable instructions configured to cause a processor of a plurality of processing units of a computing device to perform operations comprising:

obtaining static characteristics data associated with each of a victim processor and one or more of the plurality of processing units that are ready to steal work items from the victim processor (work-ready processors);
obtaining dynamic characteristics data including values associated with temperature and power consumption, associated with each of the victim processor and the work-ready processors;
calculating priority values for each of the victim processor and the work-ready processors based on the obtained data; and
transferring a number of work items assigned to the victim processor to a winning work-ready processor based on the calculated priority values.

30. A computing device, comprising:

means for obtaining static characteristics data associated with each of a victim processor and one or more of a plurality of processing units that are ready to steal work items from the victim processor (work-ready processors);
means for obtaining dynamic characteristics data including values associated with temperature and power consumption, associated with each of the victim processor and the work-ready processors;
means for calculating priority values for each of the victim processor and the work-ready processors based on the obtained data; and
means for transferring a number of work items assigned to the victim processor to a winning work-ready processor based on the calculated priority values.

* * *



US 20170083827A1

(19) United States

(21) Patent Application Publication (10) Pub. No.: US 2017/0083827 A1
Robatmili et al. (14) Pub. Date: Mar. 23, 2017

(54) DATA-DRIVEN ACCELERATOR FOR
MACHINE LEARNING AND RAW DATA
ANALYSIS

(71) Applicant: QUALCOMM Incorporated, San
Diego, CA (USA)

(72) Inventors: Behnam Robatmili, San Jose, CA
(USA); Matthew Leslie Budin, San Jose,
CA (USA); Darío Suárez Gracia, Intel
(USA); Gheorghe Calin Caseaval, Palo
Alto, CA (USA); Nayem Islam, Palo
Alto, CA (USA)

(21) Appl. No.: 14/862,408

(22) Filed: Sep. 23, 2015

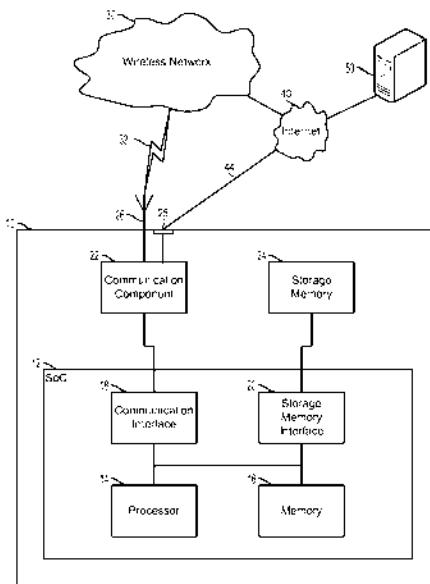
Publication Classification

(51) Int. Cl.
G06N 99/00 (2006.01)

(52) U.S. Cl.
CPC G06N 99/003 (2013.01)

(57) ABSTRACT

Embodiments include computing devices, apparatus, and methods implemented by the apparatus for accelerating machine learning on a computing device. Raw data may be received in the computing device from a raw data source device. The apparatus may identify key features as two dimensional matrices of the raw data such that the key features are mutually exclusive from each other. The key features may be translated into key feature vectors. The computing device may generate a feature vector from at least one of the key feature vectors. The computing device may receive a first partial output resulting from an execution of a basic linear algebra subprogram (BLAS) operation using the feature vector and a weight factor. The first partial output may be combined with a plurality of partial outputs to produce an output matrix. Receiving the raw data on the computing device may include receiving streaming raw data.



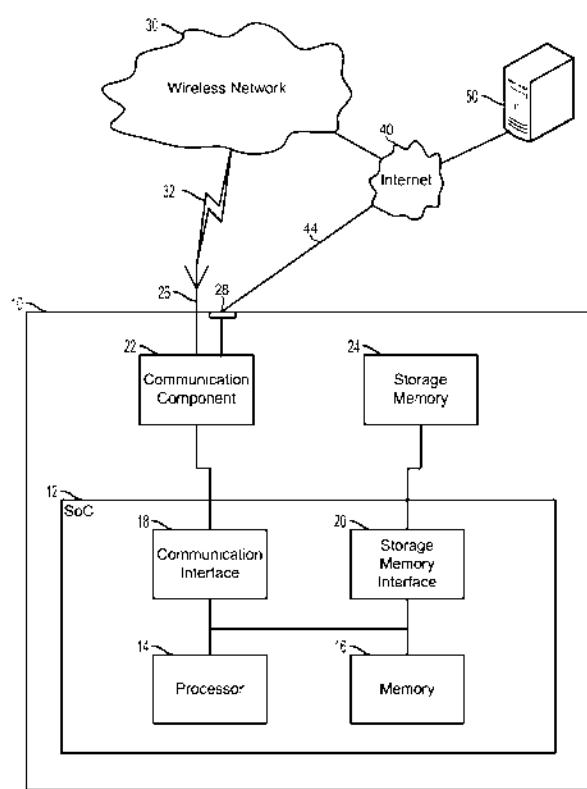


FIG. 1

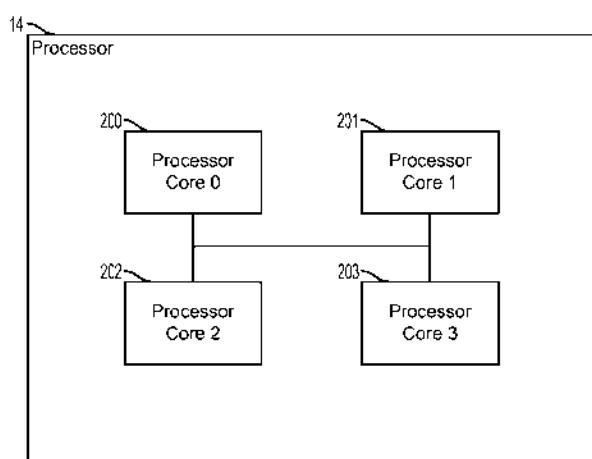


FIG. 2

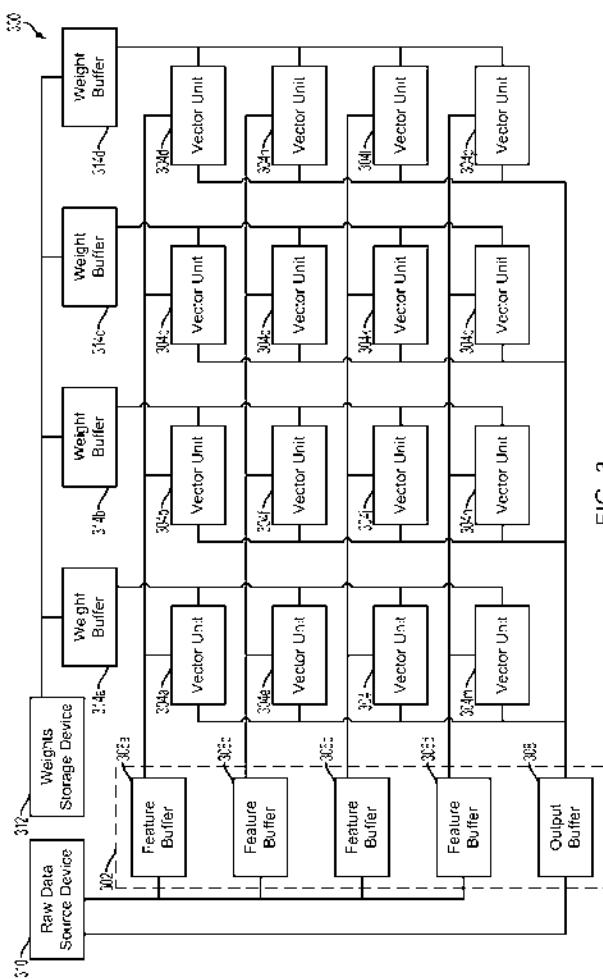


FIG. 3

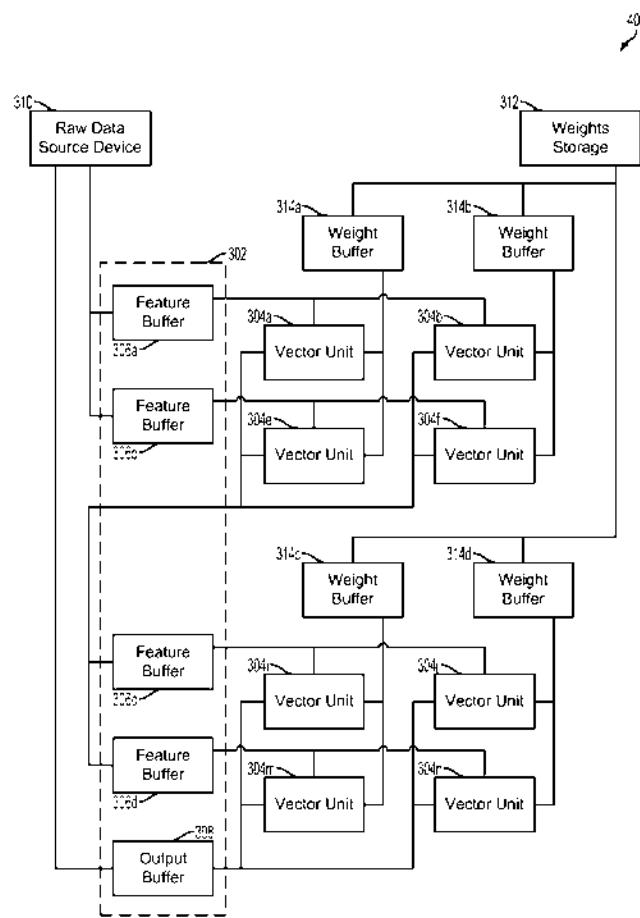


FIG. 4

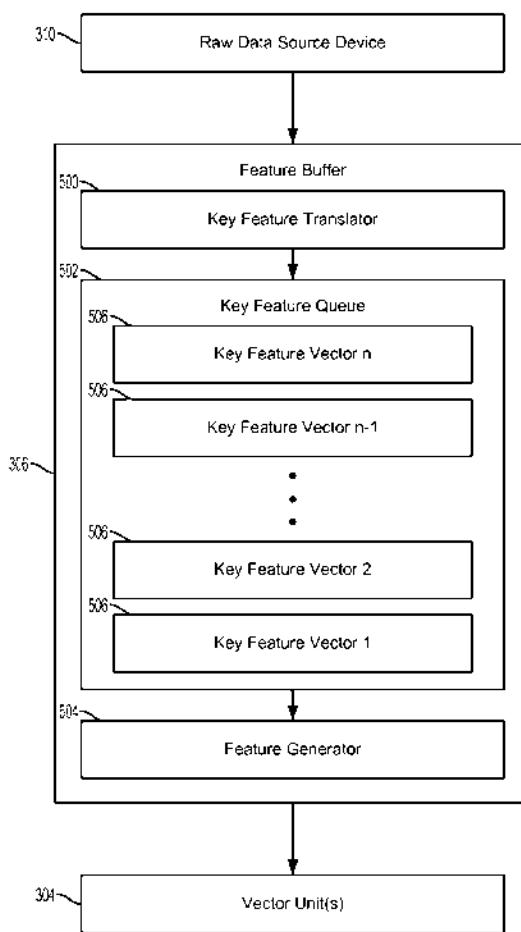


FIG. 5

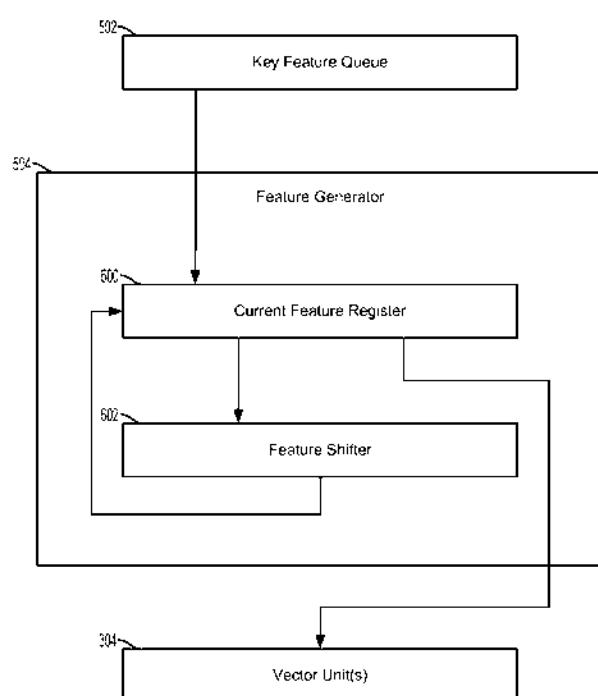


FIG. 6

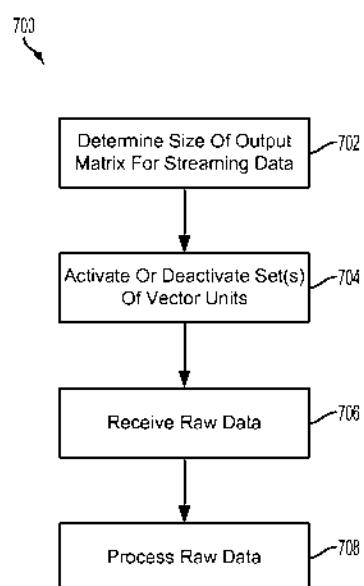


FIG. 7

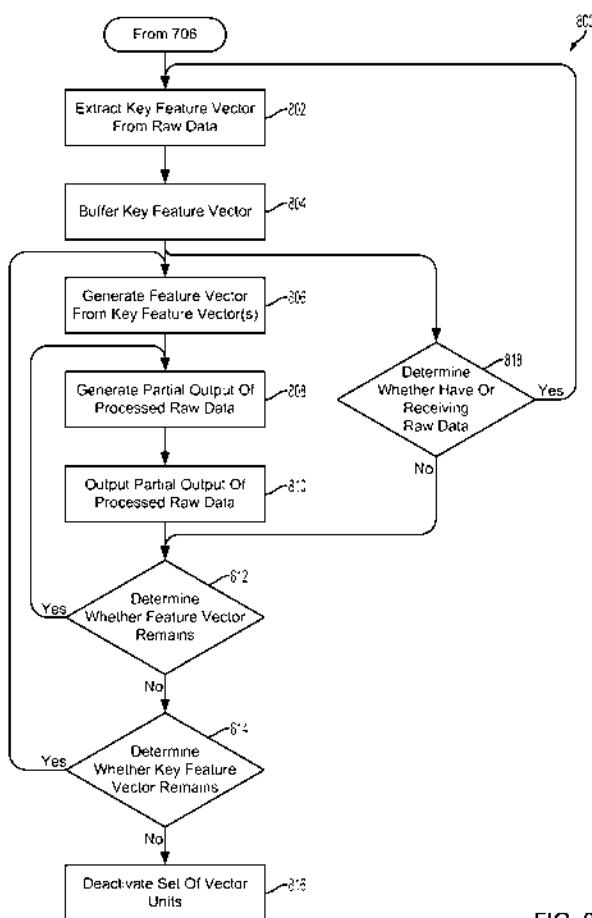


FIG. 8

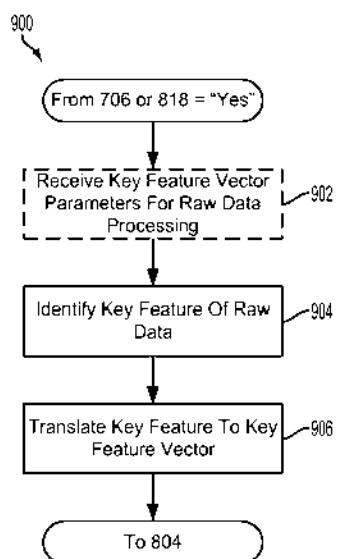


FIG. 9

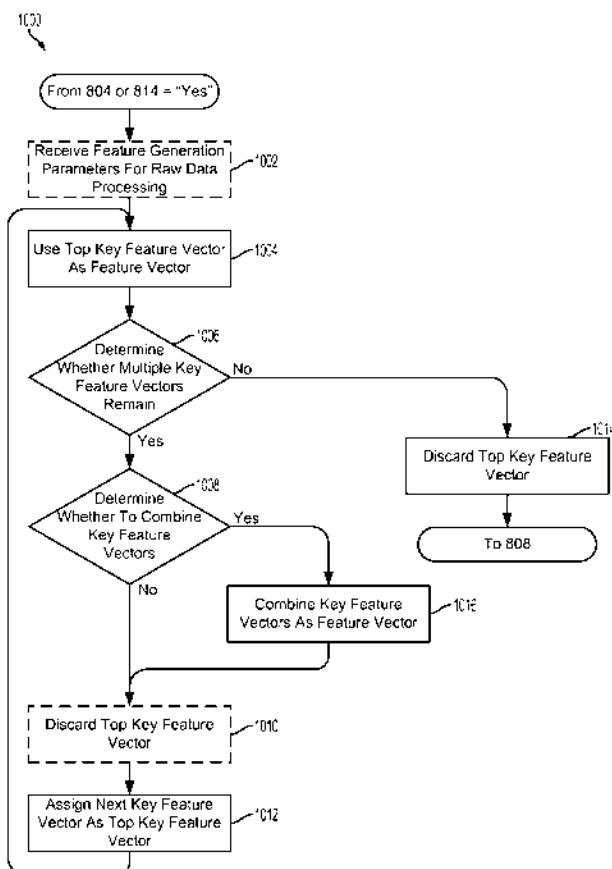


FIG. 10

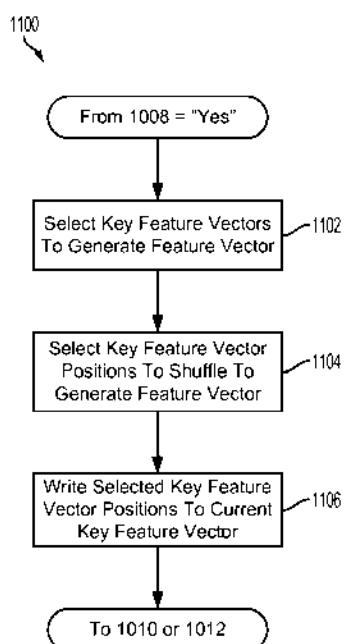


FIG. 11

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23

FIG. 12A

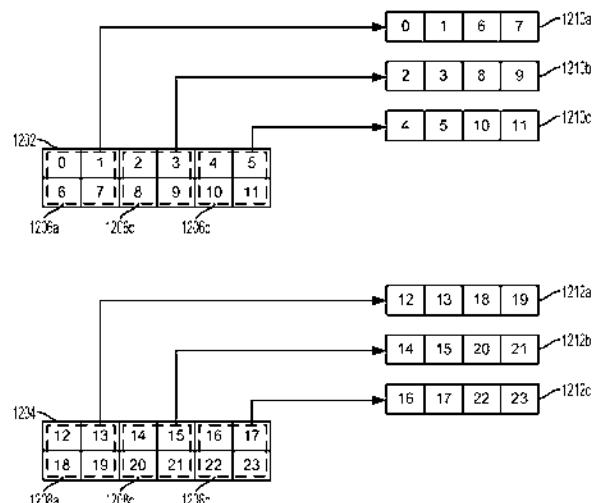


FIG. 12B

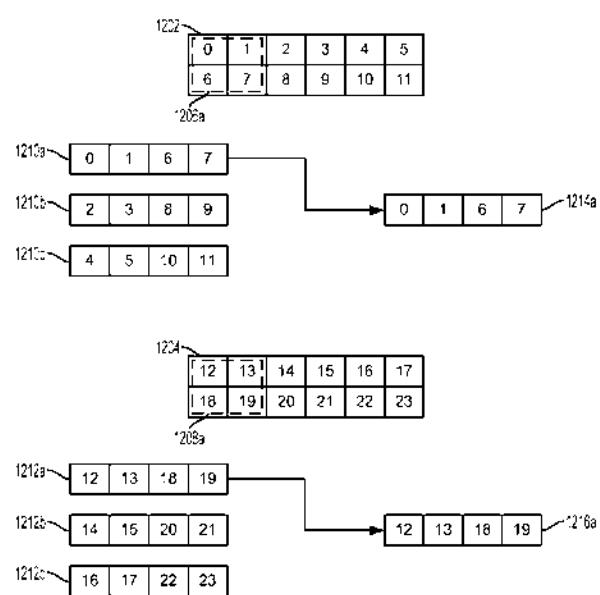


FIG. 12C

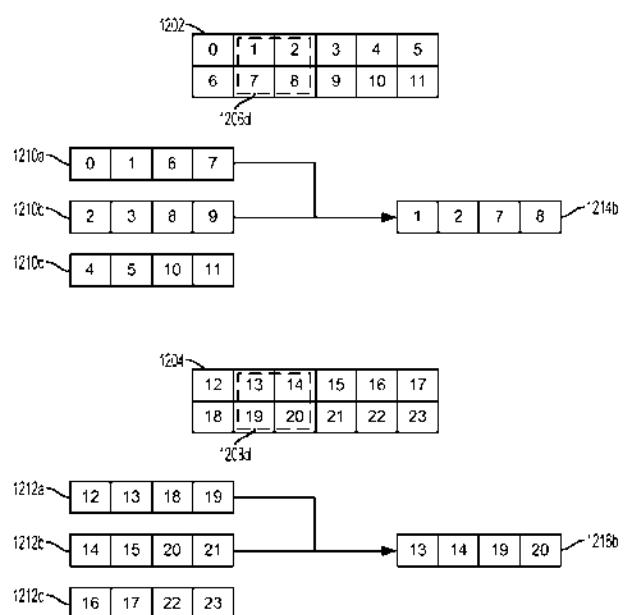


FIG. 12D

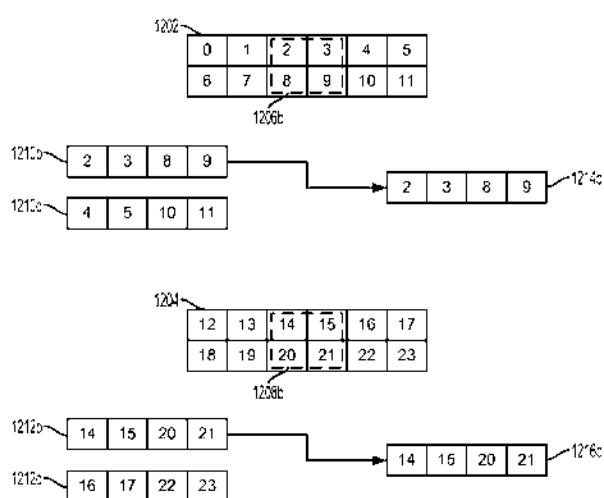


FIG. 12E

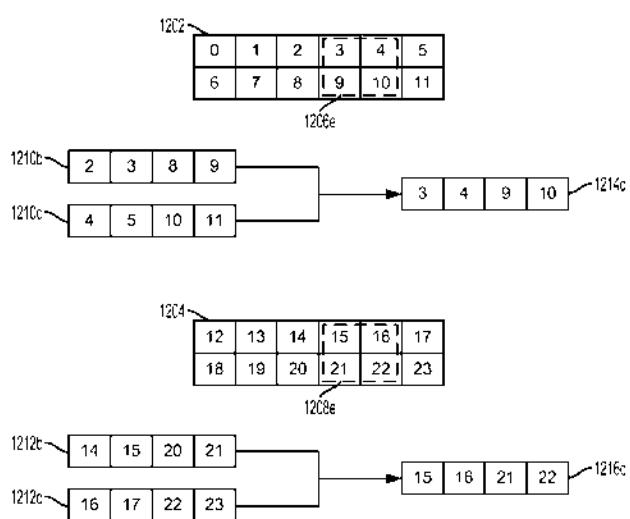


FIG. 12F

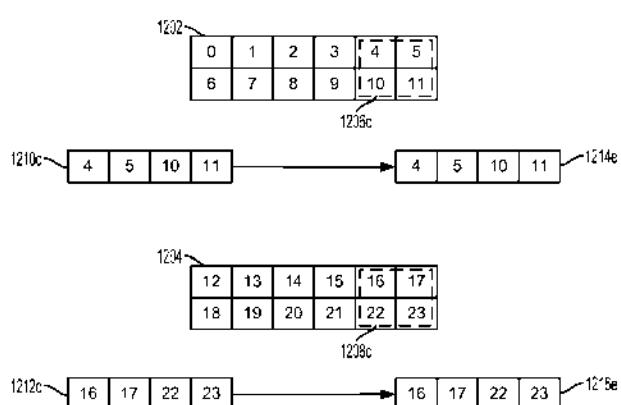


FIG. 12G

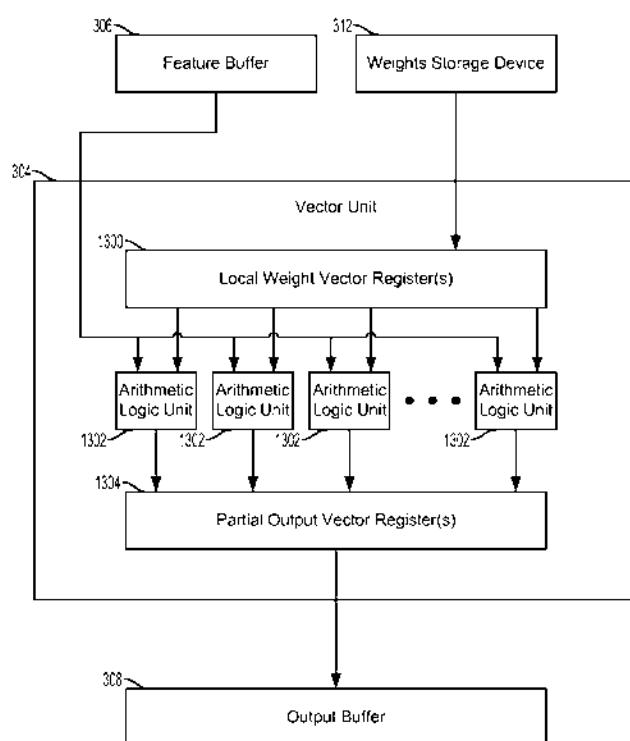


FIG. 13

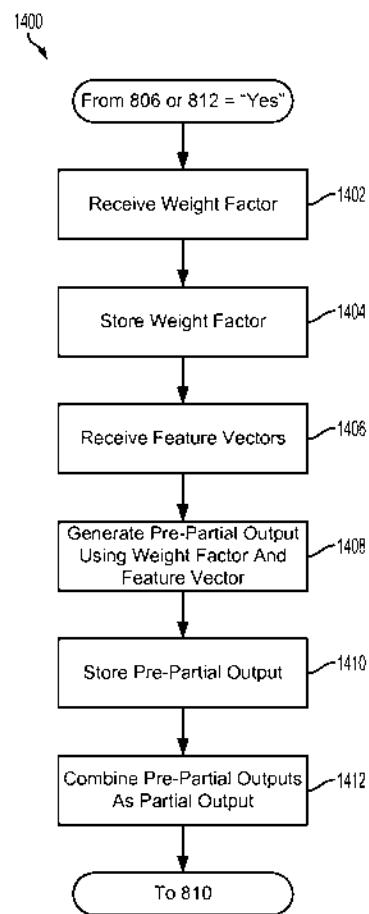


FIG. 14

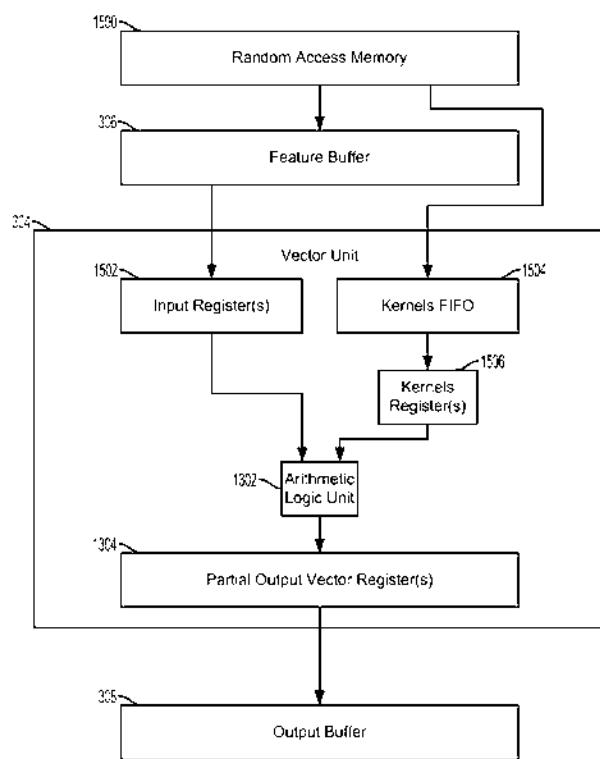


FIG. 15

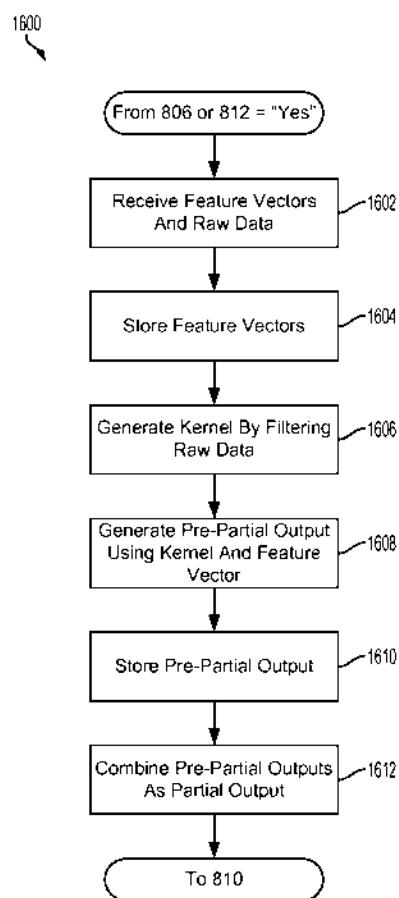


FIG. 16

0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7	
8	8	9	9	9	9	10	10	10	11	11	12	12	13	13	14	14
16	16	16	16	17	17	17	18	18	18	19	19	20	20	21	21	22
24	24	24	24	25	25	25	26	26	26	27	27	27	28	28	29	29
32	32	32	32	33	33	33	34	34	34	35	35	36	36	37	37	38

1700

F0	F0	F0	F1	F1	F1	F2	F2	F2	F3	F3	F3				
1702															

1702

FIG. 17A

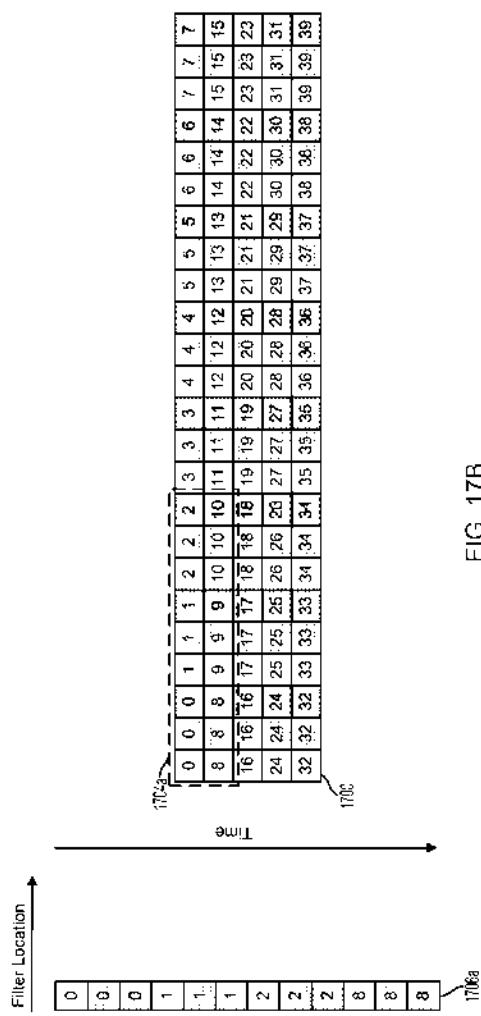


FIG. 17B

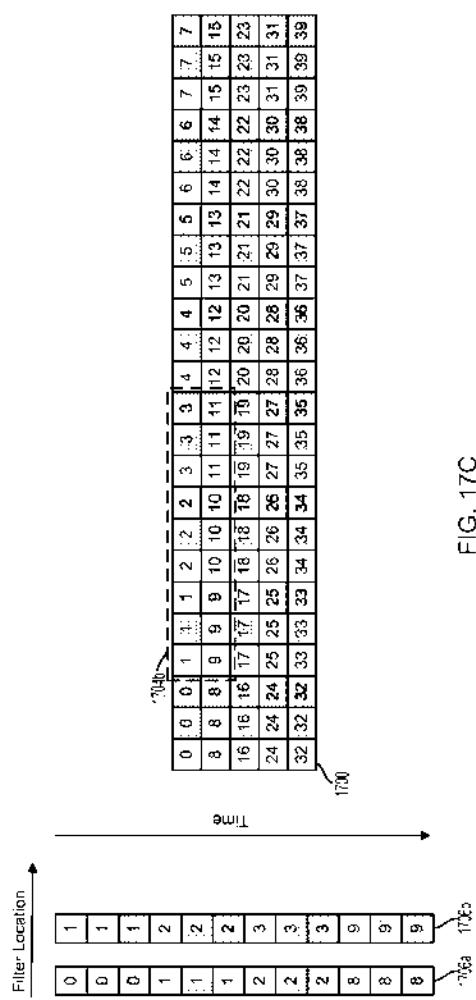


FIG. 17C

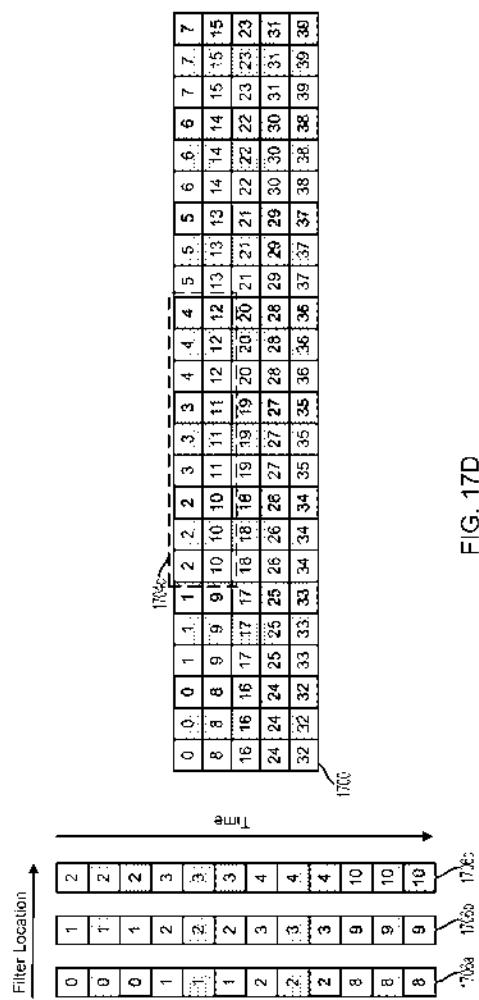


FIG. 17D

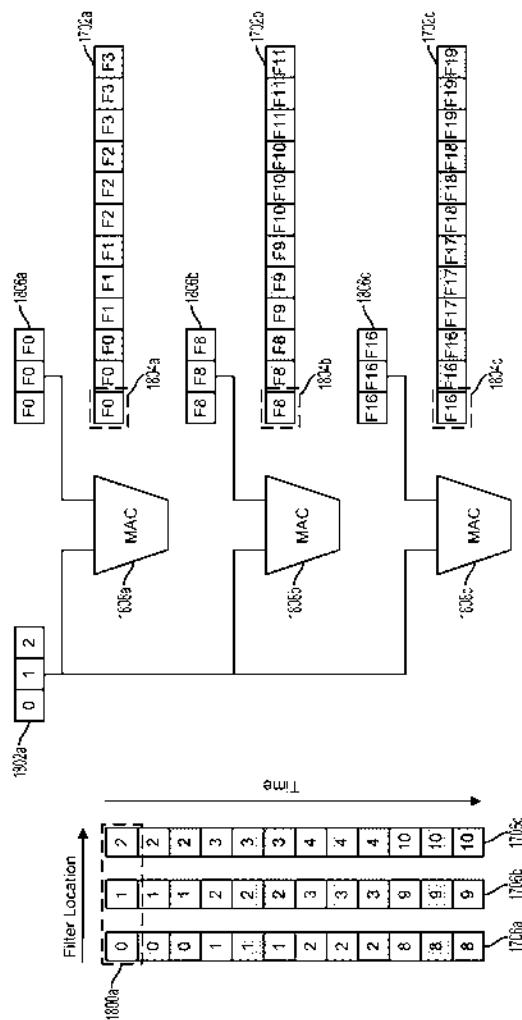


FIG. 18A

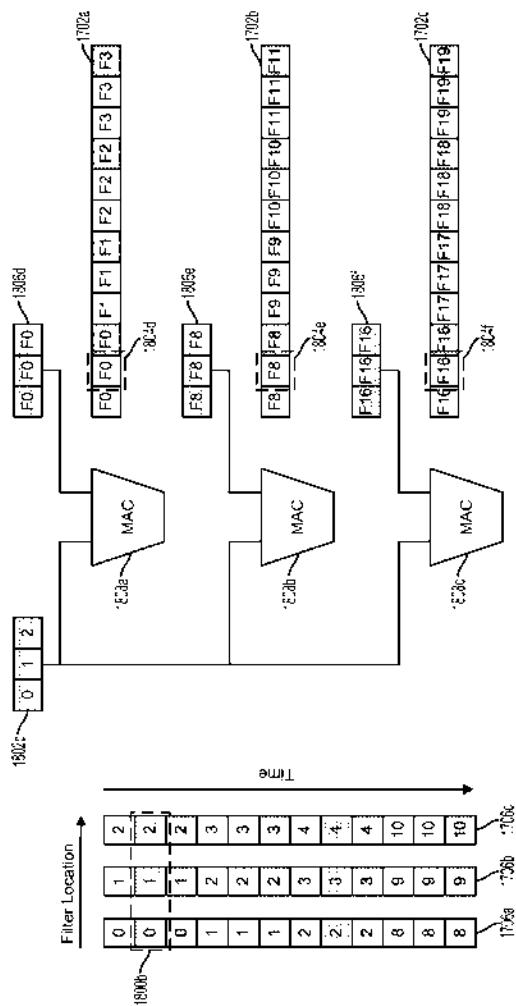


FIG. 18B

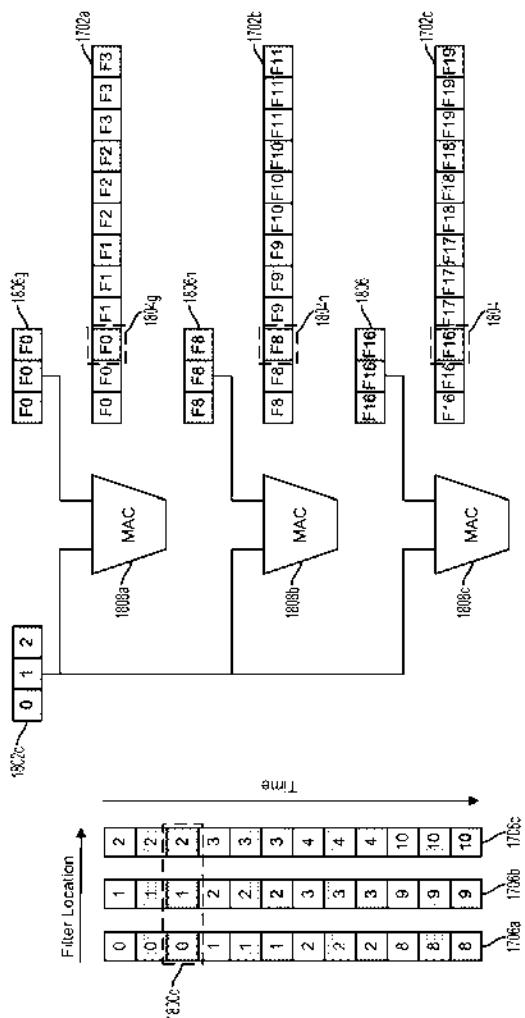


FIG. 18C

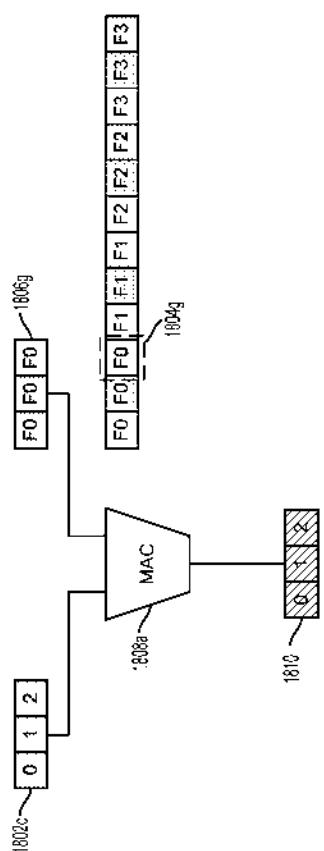


FIG. 18D

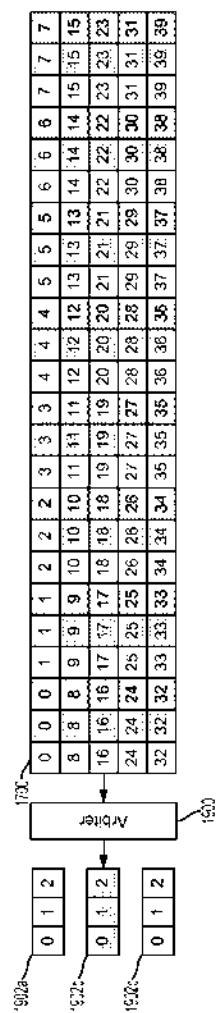


FIG. 19

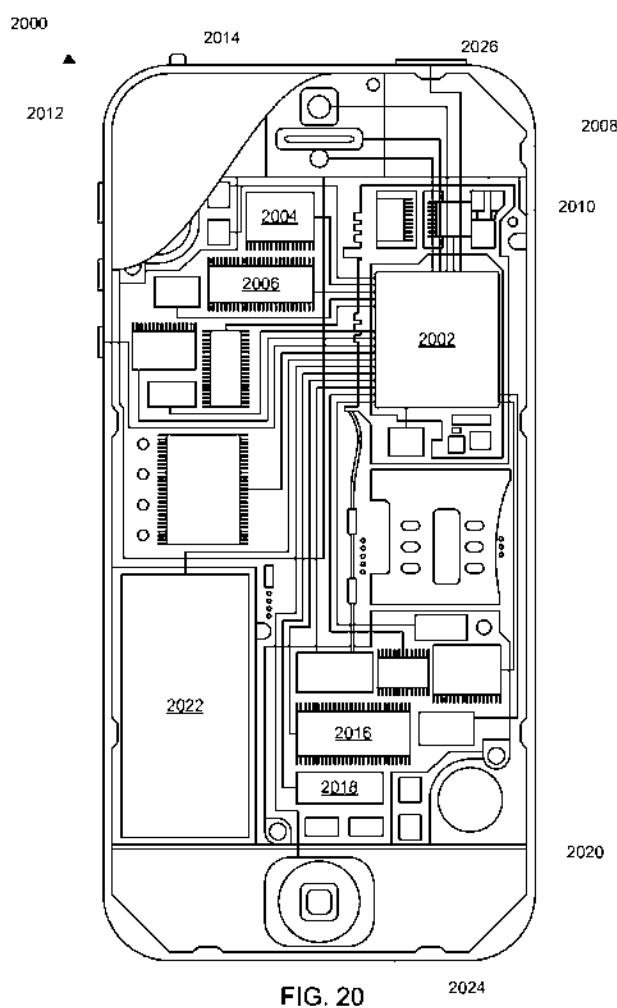


FIG. 20

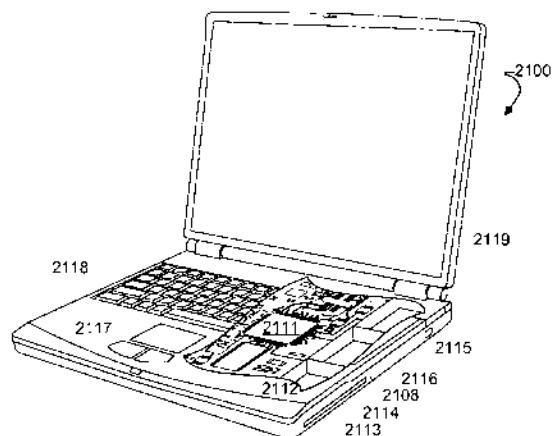


FIG. 21

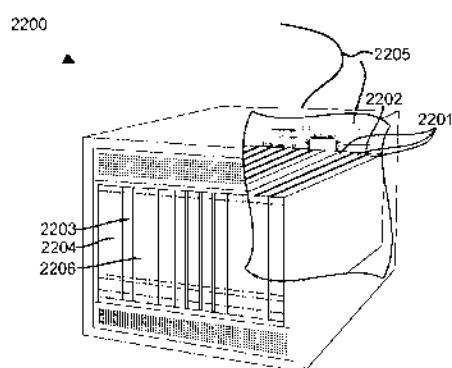


FIG. 22

DATA-DRIVEN ACCELERATOR FOR MACHINE LEARNING AND RAW DATA ANALYSIS

BACKGROUND

[0001] Most machine learning accelerators reformulate learning algorithms to define them as matrix or vector-dot product operations and then execute the machine learning using basic linear algebra subprograms (BLAS). While this approach can be considered fast, it does not reduce all of the overhead associated with data translation or data movement starting from raw data and feature extraction. Before doing machine learning in BLAS, the raw data must be read, stored, and translated to extract features needed for the machine learning or BLAS operations. Extracting key features from the stored data requires multiple memory access to retrieve the stored data and to store the extracted key features. Key features are often derived from overlapping data sets resulting in multiple memory accesses for duplicate copies of data. Thus, reformulating learning algorithms to define them as matrix or vector dot product operations and then execute the machine learning using BLAS is still inefficient given the large amount of data movement in and out of memory required before such accelerated learning is applied to the data.

SUMMARY

[0002] The methods and apparatuses of various embodiments provide circuits and methods for accelerating machine learning on a computing device. In various embodiments, the methods may include receiving raw data from a raw data source device, identifying key features as two-dimensional matrices of the raw data such that the key features are mutually exclusive from each other, translating the key features into key feature vectors, generating a feature vector from at least one of the key feature vectors, receiving a first partial output resulting from an execution of a basic linear algebra subprogram (BLAS) operation using the feature vector and a weight factor, and combining the first partial output with a plurality of partial outputs to produce an output matrix.

[0003] In some embodiments, identifying key features as two-dimensional matrices of the raw data such that the key features are mutually exclusive from each other may include identifying a first key feature as a first two-dimensional matrix of a designated size, and identifying a second key feature as a second two-dimensional matrix of the designated size a designated number of units from the first key feature.

[0004] In some embodiments, generating a feature vector from at least one of the key feature vectors may include selecting a top key feature vector from a key feature vector queue, and using the top key feature vector as the feature vector.

[0005] In some embodiments, generating a feature vector from at least one of the key feature vectors may include selecting a top key feature vector from a key feature vector queue, selecting a next key feature vector from the key feature vector queue, selecting top key feature vector positions and next key feature vector positions, and combining the selected top key feature vector position and the selected next key feature vector positions into the feature vector. In some embodiments, selecting top key feature vector positions and next key feature vector positions may include

selecting the top key feature vector positions and the next key feature vector positions such that each of the selected top key feature vector position and the selected next key feature vector positions represent mutually exclusive locations from each other in the raw data and represent an unidentified key feature of raw data that spans a plurality of the identified key features of the raw data, and combining the selected top key feature vector position and the selected next key feature vector positions into the feature vector may include combining the selected top key feature vector position and the selected next key feature vector positions into the feature vector such that the feature vector is emulated like a key feature vector of the unidentified key feature.

[0006] Some embodiments may further include activating a set of vector units upon receiving the raw data at a feature buffer associated with the set of vector units, in which the set of vector units is mapped to the output matrix, executing the BLAS operation by each vector unit of the set of vector units, and outputting at least one partial output by each vector unit. Some embodiments may further include determining whether any feature vectors remain for use in an execution of the BLAS operation by the set of vector units, and deactivating the set of vector units in response to determining that no feature vectors remain for use in an execution of the BLAS operation by the set of vector units.

[0007] In some embodiments, receiving raw data from a raw data source device may include receiving streaming raw data from the raw data source device.

[0008] Various embodiments may include an apparatus configured to accelerate machine learning on a computing device. The apparatus may include a raw data source device, and a vectorization unit communicatively connected to the raw data source and configured to perform operations of one or more embodiment methods described above.

[0009] Various embodiments may include an apparatus configured to accelerate machine learning on a computing device. The apparatus may include means for performing functions of one or more of the aspect methods described above.

[0010] Various embodiments may include a non-transitory processor-readable storage medium having stored thereon processor-executable instructions to cause a processor of a computing device to perform operations of the methods described above.

BRIEF DESCRIPTION OF THE DRAWINGS

[0011] The accompanying drawings, which are incorporated herein and constitute part of this specification, illustrate example embodiments of various embodiments, and together with the general description given above and the detailed description given below, serve to explain the features of the claims.

[0012] FIG. 1 is a component block diagram illustrating a computing device suitable for implementing an embodiment.

[0013] FIG. 2 is a component block diagram illustrating an example multi-core processor suitable for implementing an embodiment.

[0014] FIG. 3 is a component block diagram illustrating an example machine learning accelerator suitable for implementing an embodiment.

[0015] FIG. 4 is a component block diagram illustrating an example machine learning accelerator suitable for implementing an embodiment.

- [0016] FIG. 5 is a component block diagram illustrating an example feature buffer suitable for implementing an embodiment.
- [0017] FIG. 6 is a component block diagram illustrating an example feature generator suitable for implementing an embodiment.
- [0018] FIG. 7 is a process flow diagram illustrating an embodiment method for implementing acceleration of machine learning and raw data analysis.
- [0019] FIG. 8 is a process flow diagram illustrating an embodiment method for accelerating machine learning and raw data analysis.
- [0020] FIG. 9 is a process flow diagram illustrating an embodiment method for extracting a key feature vector from raw data.
- [0021] FIG. 10 is a process flow diagram illustrating an embodiment method for generating a feature from a key feature vectors.
- [0022] FIG. 11 is a process flow diagram illustrating an embodiment method for combining a top key feature vector and a next key feature vector as a feature.
- [0023] FIGS. 12A-12G are schematic diagrams illustrating an example of a process flow for extracting a key feature vector from raw data and generating a feature vector from the key feature vector for implementing an embodiment.
- [0024] FIG. 13 is a component block diagram illustrating an example vector unit suitable for implementing an embodiment.
- [0025] FIG. 14 is a process flow diagram illustrating an embodiment method for generating a partial output of a processed raw data.
- [0026] FIG. 15 is a component block diagram illustrating an example vector unit suitable for implementing an embodiment.
- [0027] FIG. 16 is a process flow diagram illustrating an embodiment method for generating a partial output of a processed raw data.
- [0028] FIGS. 17A-17D are schematic diagrams illustrating an example of a process flow for generating a kernel using filtered raw data.
- [0029] FIGS. 18A-18D are schematic diagrams illustrating an example of a process flow for generating a pre-partial output using a kernel and a feature vector.
- [0030] FIG. 19 is a schematic diagram illustrating an example of a process flow for generating a feature vector using an arbiter to assign addresses to raw data.
- [0031] FIG. 20 is a component block diagram illustrating an example mobile computing device suitable for use with the various embodiments.
- [0032] FIG. 21 is a component block diagram illustrating an example mobile computing device suitable for use with the various embodiments.
- [0033] FIG. 22 is a component block diagram illustrating an example server suitable for use with the various embodiments.
- DETAILED DESCRIPTION**
- [0034] The various embodiments will be described in detail with reference to the accompanying drawings. Wherever possible, the same reference numbers will be used throughout the drawings to refer to the same or like parts. References made to particular examples and implementations are for illustrative purposes, and are not intended to limit the scope of the claims.
- [0035] The terms "computing device" and "mobile computing device" are used interchangeably herein to refer to any one or all of cellular telephones, smartphones, personal or mobile multi-media players, personal data assistants (PDAs), laptop computers, tablet computers, convertible laptops, tablets (2-in-1 computers), notebooks, ultrabooks, netbooks, palm-top computers, wireless electronic mail receivers, multimedia Internet enabled cellular telephones, mobile gaming consoles, wireless gaming controllers, and similar personal electronic devices that include a memory, and a multi-core programmable processor. While the various embodiments are particularly useful for mobile computing devices, such as smartphones, which have limited memory and battery resources, the embodiments are generally useful in any electronic device that implements a plurality of memory devices and a limited power budget in which reducing the power consumption of the processor can extend the battery-operating time of a mobile computing device. The term "computing device" may further refer to stationary computing devices including personal computers, desktop computers, all-in-one computers, work stations, super computers, mainframe computers, embedded computers, servers, home theater computers, and game consoles.
- [0036] Embodiments include methods, and systems and devices implementing such methods, for improving learning algorithm performance by implementing hardware accelerated machine learning and raw data analysis using a data vectorization unit for traversal of raw data, extracting key feature vectors, and generating feature vectors, and a two-dimensional array of vector units for performing matrix multiplication or vector dot products of machine learning algorithms using the feature vectors and weight (kernel) vectors.
- [0037] The data vectorization unit may include multiple feature buffers and an output buffer. Each feature buffer may include a key feature translator, a key feature queue, and a feature generator for pre-processing data prior to applying machine learning on the data. Each feature buffer may interface with multiple raw data source devices, including a raw data storage device or a sensor.
- [0038] Raw data received by a feature buffer may be provided to the key feature translator for extraction of key feature vectors from the raw data for use in creating feature vectors. The feature translator may read the raw data in a traversal order or as the raw data arrives. The key feature vectors may be extracted in multiple manners depending on what data is useful for the machine learning. The useful data may be extracted and serialized as key feature vectors from the raw data, and the remaining raw data may be discarded. The key feature vectors may include only enough of the useful data for the machine learning such that the key feature vectors may be used for generating feature vectors for the machine learning, for example by interpolation, without including duplicate useful data in the key feature vectors.
- [0039] The key feature vectors may be queued in a key feature queue from which the feature generator may receive the key feature vectors for generating the feature vectors. The key feature queue may be a first-in first-out queue or a circular queue. In an embodiment, a first key feature vector in the key feature queue may represent a first feature vector, and the feature generator may output the first feature vector.
- [0040] In an embodiment, the feature generator may construct a second feature vector from a combination of the data

from the first key feature vector and data from a second key feature vector, and output the second feature vector [0041]. An array of vector units, topologically mapped to an output matrix, may receive the feature vectors from and provide the output matrix to the data vectorization unit. Each vector unit may include a weight buffer, a process unit, and a partial output buffer. A set of vector units may be associated with a feature buffer, and the set of vector units may receive the feature vectors from the associated feature buffer. The vector units may also receive a weight vector, which may be provided from memory, and store the weight vector in the weight buffer. The process unit is arranged to implement a vector function (e.g., a sigmoid function, multiply-accumulate operation, etc.) using the received feature vector, the weight vector, and/or the feature vector altered by the weight factor. Partial outputs of the process unit may be stored in the partial output buffer until the complete output from processing the feature vector is output to the output buffers or back to the feature buffers of the data vectorization unit. The complete output from each vector unit may represent a portion of an output matrix.

[0042] The data received by the feature buffer may be streamed from the raw data source device to the feature buffer, even while the data continue to be collected by the raw data source device. The components of the data vectorization unit and the array of vector units may operate on their respective inputs concurrently. For each component of the data vectorization unit and the vector units, an input may trigger a respective operation.

[0043] The key feature translator may continually extract and output key feature vectors from the streaming data. The key feature queue may continually retain the key feature vectors and provide the key feature vectors to the feature generator. The feature generator may continually construct and output the feature vectors. The vector units may continually process the feature vectors and output portions of the output matrix until there is no streaming data, key feature vectors, or feature vectors remaining. In response to a lack of streaming data and no activity of an associated set of components in the data vectorization unit and the array of vector units, the data vectorization unit and/or array of vector units may enter or partially enter a low power idle state, powering down some components.

[0044] The data vectorization unit and the array of vector units in hardware may be arranged so that streaming data may be operated on to perform raw data analysis and machine learning in a just-in-time data-flow manner, where there is no need to wait for a full set of data from a data recording event. Thus, the various embodiments enable more efficient use of resources by eliminating multiple memory access operations for retrieving raw data and storing pre-processed data, and central processing unit (CPU) operations for pre-processing the raw data. The manner in which the key feature vectors are extracted and the feature vectors are generated further reduces resource usage by avoiding memory accesses and CPU operations for duplicate data.

[0045] FIG. 1 illustrates a system including a computing device 10 in communication with a remote computing device 50 suitable for use with the various embodiments. The computing device 10 may include a system-on-chip (SoC) 12 with a processor 14, a memory 16, a communication interface 18, and a storage memory interface 20. The computing device may further include a communication

component 22 such as a wired or wireless modem, a storage memory 24, an antenna 26 for establishing a wireless connection 32 to a wireless network 30, and/or the network interface 28 for connecting to a wired connection 44 to the Internet 40. The processor 14 may include any of a variety of hardware cores, for example a number of processor cores [0046]. The term "system-on-chip" (SoC) is used herein to refer to a set of interconnected electronic circuits typically, but not exclusively, including a hardware core, a memory, and a communication interface. A hardware core may include a variety of different types of processors, such as a general purpose processor, a central processing unit (CPU), a digital signal processor (DSP), a graphics processing unit (GPU), an accelerated processing unit (APU), an auxiliary processor, a single-core processor, and a multi-core processor. A hardware core may further embody other hardware and hardware combinations, such as a field programmable gate array (FPGA), an application-specific integrated circuit (ASIC), other programmable logic device, discrete gate logic, transistor logic, performance monitoring hardware, watchdog hardware, and time references. Integrated circuits may be configured such that the components of the integrated circuit reside on a single piece of semiconductor material, such as silicon. The SoC 12 may include one or more processors 14. The computing device 10 may include more than one SoC's 12, thereby increasing the number of processors 14 and processor cores. The computing device 10 may also include processors 14 that are not associated with an SoC 12. Individual processors 14 may be multi-core processors as described below with reference to FIG. 2. The processor 14 may each be configured for specific purposes that may be the same as or different from other processors 14 of the computing device 10. One or more of the processors 14 and processor cores of the same or different configurations may be grouped together. A group of processors 14 or processor cores may be referred to as a multi-processor cluster.

[0047] The memory 16 of the SoC 12 may be a volatile or non-volatile memory configured for storing data and processor-executable code for access by the processor 14. The computing device 10 and/or SoC 12 may include one or more memories 16 configured for various purposes. In an embodiment, one or more memories 16 may include volatile memories such as random access memory (RAM) or main memory, or cache memory. These memories 16 may be configured to temporarily hold a limited amount of data received from a data sensor or subsystem, data and/or processor-executable code instructions that are requested from non-volatile memory, loaded in the memories 16 from non-volatile memory in anticipation of future access based on a variety of factors, and/or intermediary processing data and/or processor-executable code instructions produced by the processor 14 and temporarily stored for future quick access without being stored in non-volatile memory.

[0048] The memory 16 may be configured to store data and processor-executable code, at least temporarily, that is loaded to the memory 16 from another memory device, such as another memory 16 or storage memory 24, for access by one or more of the processors 14. The data or processor-executable code loaded to the memory 16 may be loaded in response to execution of a function by the processor 14. Loading the data or processor-executable code to the memory 16 in response to execution of a function may result from a memory access request to the memory 16 that is

unsuccessful, or a miss, because the requested data or processor-executable code is not located in the memory 16. In response to a miss, a memory access request to another memory 16 or storage memory 24 may be made to load the requested data or processor-executable code from the other memory 16 or storage memory 24 to the memory device 16. Loading the data or processor-executable code to the memory 16 in response to execution of a function may result from a memory access request to another memory 16 or storage memory 24, and the data or processor-executable code may be loaded to the memory 16 for later access.

[0049] In an embodiment, the memory 16 may be configured to store raw data, at least temporarily, that is loaded to the memory 16 from a raw data source device, such as a sensor or subsystem. Raw data may stream from the raw data source device to the memory 16 and be stored by the memory until the raw data can be received and processed by a machine learning accelerator as discussed further herein with reference to FIGS. 3-19.

[0050] The communication interface 18, communication component 22, antenna 26, and/or network interface 28 may work in unison to enable the computing device 10 to communicate over a wireless network 30 via a wireless connection 32, and/or a wired network 34 with the remote computing device 30. The wireless network 30 may be implemented using a variety of wireless communication technologies, including, for example, radio frequency spectrum used for wireless communications, to provide the computing device 10 with a connection to the Internet 40 by which it may exchange data with the remote computing device 30.

[0051] The storage memory interface 20 and the storage memory 24 may work in unison to allow the computing device 10 to store data and processor-executable code on a non-volatile storage medium. The storage memory 24 may be configured much like an embodiment of the memory 16 in which the storage memory 24 may store the data or processor-executable code for access by one or more of the processors 14. The storage memory 24, being non-volatile, may retain the information even after the power of the computing device 10 has been shut off. When the power is turned back on and the computing device 10 reboots, the information stored on the storage memory 24 may be available to the computing device 10. The storage memory interface 20 may control access to the storage memory 24 and allow the processor 14 to read data from and write data to the storage memory 24.

[0052] Some or all of the components of the computing device 10 may be differently arranged and/or combined while still serving the necessary functions. Moreover, the computing device 10 may not be limited to one of each of the components, and multiple instances of each component may be included in various configurations of the computing device 10.

[0053] FIG. 2 illustrates a multi-core processor 14 suitable for implementing an embodiment. The multi-core processor 14 may have a plurality of homogeneous or heterogeneous processor cores 200, 201, 202, 203. The processor cores 200, 201, 202, 203 may be homogeneous in that, the processor cores 200, 201, 202, 203 of a single processor 14 may be configured for the same purpose and have the same or similar performance characteristics. For example, the processor 14 may be a general purpose processor, and the processor cores 200, 201, 202, 203 may be homogeneous

general purpose processor cores. Alternatively, the processor 14 may be a graphics processing unit or a digital signal processor, and the processor cores 200, 201, 202, 203 may be homogeneous graphics processor cores or digital signal processor cores, respectively. For ease of reference, the terms "processor" and "processor core" may be used interchangeably herein.

[0054] The processor cores 200, 201, 202, 203 may be heterogeneous in that, the processor cores 200, 201, 202, 203 of a single processor 14 may be configured for different purposes and/or have different performance characteristics. The heterogeneity of such heterogeneous processor cores may include different instruction set architecture, pipelines, operating frequencies, etc. An example of such heterogeneous processor cores may include what are known as "big.LITTLE" architectures in which slower, low-power processor cores may be coupled with more powerful and power-hungry processor cores. In similar embodiments, the SoC 12 may include a number of homogeneous or heterogeneous processor cores 14.

[0055] In the example illustrated in FIG. 2, the multi-core processor 14 includes four processor cores 200, 201, 202, 203 (i.e., processor core 0, processor core 1, processor core 2, and processor core 3). For ease of explanation, the examples herein may refer to the four processor cores 200, 201, 202, 203 illustrated in FIG. 2. However, the four processor cores 200, 201, 202, 203 illustrated in FIG. 2 and described herein are merely provided as an example and in no way are meant to limit the various embodiments to a four-core processor system. The computing device 10, the SoC 12, or the multi-core processor 14 may individually or in combination include fewer or more than the four processor cores 200, 201, 202, 203 illustrated and described herein.

[0056] FIG. 3 illustrates an example machine learning accelerator 300 suitable for implementing an embodiment. The machine learning accelerator 300, which is also referred to as an annulus herein, may include a data vectorization unit 302 and an array of vector units 304 (e.g., 304a-304n). The machine learning accelerator 300 may include or be connected to a raw data source device 310, a weight storage device 312, and a number of weight buffers 314 (e.g., 314a-314n). The machine learning accelerator 300 may be configured to accelerate the processing of raw data by vectorizing the raw data into feature vectors of the raw data and performing matrix multiplication or vector dot products of machine learning algorithms. The composition of the components of the machine learning accelerator 300 may differ depending on various factors, including the machine learning algorithm implemented, the size and/or complexity of the raw data, and the power and/or performance requirements of the computing device.

[0057] The data vectorization unit 302 may include a number of feature buffers 306 (e.g., 306a-306f) and at least one output buffer 308. The raw data source device 310 may provide raw data to the data vectorization unit 302. In an embodiment, the raw data may be streamed from the raw data source device 310 to the data vectorization unit 302. Streaming the raw data may include continually providing the raw data to the data vectorization unit 302 as the raw data is acquired or else in line thereafter by the raw data source device 310. For example, the raw data source device 310 may be a video capture device that may stream raw video data as it is captured by the video capture device. The raw data source device 310 may similarly be any device capable

of acquiring data relating to an input in real-time or near-real-time, such as at least one of an audio sensor, an electromagnetic radiation sensor, chemical sensor, temperature sensor, etc. In another example, the raw data source device 310 may be a fast memory, such as a cache memory, random access memory, or other solid state memory device, connected to a sensor and receiving the raw data from the sensor. The fast memory may provide the raw data to the data vectorization unit 302 as the raw data is acquired or close in time thereto. In an embodiment, the fast memory may store the raw data and provide it to the data vectorization unit 302 in a streaming manner as needed.

[0058] The data vectorization unit 302 may receive the raw data of the feature buffers 306. Various combinations of feature buffers 306 may be used to receive the raw data (e.g., feature buffer 306a, feature buffers 306a and 306c; feature buffers 306a-306c; or feature buffers 306a-306d). The feature buffers 306 may receive the raw data and extract feature vectors from the raw data discussed further herein with reference to FIGS. 5, 6, and 8-12G. Each feature buffer 306 may be activated or inactivated depending on whether there is raw data available for the feature buffer 306. The number of feature buffers 306 included in the data vectorization unit 302 may depend on various factors, including the machine learning algorithms implemented, the size and/or complexity of the raw data, and the power and/or performance requirements of the computing device.

[0059] The feature buffers 306 may output the feature vectors to the array of vector units 304. Each feature buffer 306 may be associated with a set of the array of vector units 304. In an embodiment, each feature buffer 306 may be associated with a row of the array of vector units 304 (e.g., feature buffer 306a may be associated with vector units 304a-304d; feature buffer 306b may be associated with vector units 304e-304h; feature buffer 306c may be associated with vector units 304i-304j; and feature buffer 306d may be associated with vector units 304k-304l). The array of vector unit 304 may be logically mapped to an output matrix representing the structure of the output data from the machine learning algorithms used to process the raw data. The feature vectors received from the feature buffers 306 may represent portions of the raw data matching locations in the raw data with locations in the output matrix for the processed data. Respective feature vectors may be received by the vector units 304 from their associated feature buffer 306. In the example in which a row of vector units 304 is associated with a particular feature buffer 306, each feature buffer in the row of vector units 304 may receive the same feature vector in a respective portion of the feature vector.

[0060] Weight factors may be used by the vector units 304 to modify the values of the feature vectors. In an embodiment, a weights storage device 312 may be any type of volatile or non-volatile storage device, and may store the weight factors for modifying the feature vectors. The weight factors may be retrieved from the weight storage device 312 and received by the weight buffers 314. The vector units 304 may be connected to or include a weight buffer 314 associated with the vector unit 304. In an example, a dedicated weight buffer 314 may be associated with a column of the array of vector units 304 (e.g., weight buffer 314a may be associated with vector units 304a, 304c, 304d, and 304g; weight buffer 314b may be associated with vector units 304b, 304f, 304h, and 304i; weight buffer 314c may be associated with vector units 304e, 304g, 304k, and 304l; and

weight buffer 314d may be associated with vector units 304d, 304g, 304i, and 304j). The weight factors received by each weight buffer 314 may be the same weight factors for all of the vector units 304 associated with a respective weight buffer 314, or the weight factors may vary for different vector units 304 associated with a respective weight buffer 314.

[0061] The vector units 304 may be configured to perform a vector function (e.g., a sigmoid function, multiply-accumulate operation, etc.) on the feature vectors, either using the feature vector as received or as modified by the weight factor. The vector function performed by the vector units 304 may vary depending on the type of data analysis and machine learning. Operating on the feature vectors by the vector units 304 allows the machine learning accelerator 300 to execute the machine learning using basic linear algebra subprograms. The resulting output of each vector unit 304 is a partial output of the output matrix for the array of vector units 304. Each vector unit 304 and weight buffer 314 may be activated or deactivated depending on whether there is raw data available for an associated feature buffer 306 or a feature vector for the vector unit 304. Activation/deactivation of the vector units 304 and weight buffers 314 may also depend on the size of the feature vectors. The number of vector units 304 and weight buffers 314 may depend on various factors, including the machine learning algorithms implemented, the size and/or complexity of the raw data, and the power and/or performance requirements of the computing device.

[0062] The output matrix may represent a matrix multiplication or vector dot product of the feature vectors and the weights. The partial outputs of the vector units 304 may be output to the output buffer 308 of the data vectorization unit 302. The output buffer 308 may temporarily store the partial output until the output matrix for a portion of the raw data is completed, and output the output matrix to a processor 14, subsystem, or memory 16, 24 of the computing device 10 (reference 11G-14), or may output the output matrix to the feature buffers 306 for further processing. The machine learning accelerator 300 may eventually produce output matrices in response to receiving the raw data.

[0063] FIG. 4 illustrates an example machine learning accelerator 400 (also referred to as an apparatus herein suitable for implementing an embodiment). The machine learning accelerator 400 may be implemented in a variety of configurations depending on various factors, including the machine learning algorithms implemented, the size and/or complexity of the raw data, the power and/or performance requirements of the computing device, and the processing requirements for the raw data. In an example illustrated in FIG. 4, the machine learning accelerator 400 may include similar components to the example illustrated in FIG. 3, including the data vectorization unit 302, the vector units 304 (e.g., 304a, 304b, 304c, 304d, 304e, 304f, 304g, and 304h), the machine learning accelerator 300 may also include or be connected to the raw data source device 310, the weight storage device 312, and the weight buffers 314 (e.g., 314a-314d). In an embodiment, the raw data may require multiple iterations machine learning processing before the output matrix may be completed. The example in FIG. 4 illustrates a two iteration machine learning process. In this example, the feature vectors produced by feature buffers 306a-306d are operated on by the vector units 304a, 304b, 304c, 304d. The partial output of the vector units 304a,

[304b, 304c, 304d] may be fed to the feature buffers 306, 316, rather than to the output buffer 308 as in the example illustrated in FIG. 3. The feature buffers 306, 316, 306d may produce further feature vectors from the partial outputs of the vector units 304a, 304b, 304c, 304d. The feature vectors produced from the partial outputs may be operated on by the vector units 304e, 304f, 304ac, and 304b, which may produce further partial outputs that are used to produce the output matrix in the output buffer 308.

[10664] FIG. 5 illustrates an example feature buffer 306 suitable for implementing an embodiment. The feature buffer 306 may include a key feature translator 500, a key feature queue 502 and a feature generator 504. As in the other examples described herein, the feature buffer 306 may be connected to the raw data source device 310, and receive raw data on a streaming or as-needed basis from the raw data source device 310. The key feature translator 500 may extract key feature vectors from the raw data for use in generating the feature vectors. The key feature vectors may include portions of raw data, or key features, that are sized based on feature vector requirements for implementing the machine learning. In other words, the size of a key feature vector may match the size of the feature vector used in the vector operations of the vector units. The portions of raw data, or key features, used to produce the key feature vectors may be determined by a set of parameters provided based on the machine learning implemented by the machine learning accelerator. In an embodiment, the key feature vector parameters may include a size parameter and a stride parameter. The size parameter may determine a size of a matrix of raw data, or key features, to use for producing the key feature vectors, and may depend on a type of machine learning, a granularity for processing the raw data, and/or a number and capability of the vector units of the machine learning accelerator. The stride parameter may determine a movement of the start of the raw data, or key features, for producing the key feature vectors. The stride parameter may be set such that the selections of raw data, or key features, for the key feature vectors do not overlap, or are mutually exclusive from each other. The key feature translator 500 may extract the key feature vectors from the raw data as it receives the raw data and output the key feature vectors to the key feature queue 502.

[10665] The key feature queue 502 may be configured to temporarily store the key feature vectors 506. The key feature queue 502 may be a first-in-first-out queue or a circular queue configured to store “n” key feature vectors 506. The key feature vectors 506 may be received by the key feature queue 502 as they are extracted from the raw data by the key feature translator 500. A key feature vector 506 (e.g., key feature vector 1) at the top of the key feature queue 502 may be output to the feature generator 504. In an embodiment, the key feature vector 506 output to the feature generator 504 may be discarded or overwritten so that a next key feature vector 506 (e.g., key feature vector 2) may be moved to the top of the key feature queue 502, the remaining key feature vectors 506 may be shifted up in the key feature queue 502, and a new key feature vector 506 may be written to the bottom of the key feature queue 502.

[10666] The feature generator 504 may receive a key feature vector 506 from the key feature queue 502 and generate a feature vector using the key feature vector 506, as discussed further herein with reference to FIGS. 6, 10, 11, and 12-12G. In an embodiment, the feature generator 504 may

leave the key feature vector 506 unaltered and use it as the feature vector. In an embodiment, the feature generator 504 may use portions of a first key feature vector 506 combined with portions of a second key feature vector 506 to generate the feature vector. The generated feature vectors may represent vectorized portions of the raw data. The feature vectors may be output to the vector units 304 associated with the feature buffer 306.

[10667] FIG. 6 illustrates an example feature generator 504 suitable for implementing an embodiment. The feature generator 504 may be connected between the feature queue 502 and the vector units 304 associated with the feature buffer having the feature generator 504. The feature generator 504 may receive key feature vectors from the feature queue 502 and output feature vectors to the vector units 304. The feature generator 504 may include a storage device for the received key feature vector, such as a current feature register 600, and an operating device for modifying the received key feature vectors, such as the feature shifter 602. The feature generator 504 may be configured to generate feature vectors based on various factors, including the machine learning algorithms implemented, the size and/or complexity of the raw data, the power and/or performance requirements of the computing device, the processing requirements for the raw data, the number and capability of the vector units of the machine learning accelerator, and the configuration of the key feature vectors, and the configuration of the key feature vectors.

[10668] A key feature vector received from the key feature queue 502 may be written to the current feature register 600. In an embodiment, the feature generator 504 may alternate between using the key feature vector as is to generate the feature vector and modifying the key feature vector to generate the feature vector. For feature vectors generated from unmodified key feature vectors, the feature generator 504 may output the generated feature vector to the connected vector units 304. For feature vectors generated from modified key feature vectors, the feature generator 504 may write the received key feature vector from the current feature register 600 to the feature shifter 602. The key feature vector written to the feature shifter 602 may be modified by combining the key feature vector with another key feature vector to generate a feature vector that is a combination of multiple key feature vectors. The generated feature vector may be written to the current feature register 600 and input to the connected vector units 304.

[10669] FIG. 7 illustrates an embodiment method 700 for implementing acceleration of machine learning and raw data analysis. The method 700 may be implemented in a computing device in software executing in a processor, in general purpose hardware, or dedicated hardware, such as a processor executing software within a machine learning accelerator that includes other individual components. In order to encompass the alternative configurations enabled in the various embodiments, the hardware implementing the method 700 is referred to herein as an “apparatus.”

[10670] In block 702, an apparatus (e.g., a machine learning accelerator) of a computing device may determine a size of a processing matrix for the streaming data. The size of the processing matrix for the streaming data may be used to activate and deactivate the feature buffers and vector units of the machine learning accelerator. The processing matrix may be implemented in a variety of configurations depending on various factors, including the machine learning

algorithms implemented, the size and/or complexity of the raw data, the power and/or performance requirements of the computing device, and the processing requirements for the raw data. The processing matrix is not required to be the same size as the output matrix. For example, the processing matrix may be smaller than the output matrix, because the activated vector units may output their partial outputs to the output matrix, and the output matrix may be assembled in the output buffer using multiple partial outputs from the vector units.

[0071] In block 704, the apparatus may activate or deactivate one or more sets (e.g., rows or columns) of vector units. In an embodiment, a feature buffer associated with deactivated vector units may also be deactivated when all of its associated vector units are deactivated. In an embodiment, a feature buffer associated with activated vector units may also be activated when even a single associated vector unit is activated. In block 706, the apparatus may receive the raw data, either on a streaming or as need basis. In an embodiment, the raw data may be received at the machine learning accelerator from the raw data source device. In block 708, the apparatus may process the raw data, discussed further herein with reference to FIGS. 5, 6, and 14-19 [0072]. FIG. 8 illustrates an embodiment method 800 for accelerating machine learning and raw data analysis. The method 800 may be executed as part of block 708 in the method 700. The method 800 may be implemented in a computing device in software executing in a processor, in general purpose hardware, or dedicated hardware, such as a processor executing software within a machine learning accelerator that includes other individual components. In order to encompass the alternative configurations enabled in the various embodiments, the hardware implementing the method 800 is referred to herein as an apparatus.

[0073] In block 802, an apparatus of the computing device may extract key features from the raw data received in a streaming or as needed manner. Which of the raw data may be used in the key feature vectors and how the raw data is used to generate the key feature vectors may be determined based on the size and stride parameters for generating the key feature vectors, as discussed further herein with reference to FIGS. 5, 9, 12A, and 12B.

[0074] In block 804, the apparatus may buffer the key feature vectors. In an embodiment, buffering the key feature vectors may include writing the key feature vectors to appropriate locations in the key feature buffers.

[0075] In block 806, the apparatus may generate feature vectors from the key feature vectors, as discussed further herein with reference to FIGS. 6, 10, 11, and 12C-12G. In block 808, the apparatus may generate a partial output of the processed raw data. In an embodiment, the feature vectors may be used in an operation to generate and output the partial output of the processed raw data as discussed further herein with reference to FIGS. 13-19. In block 810, the apparatus may output the partial output of the processed raw data. In an embodiment, the partial output may be output from the vector units to the output buffer.

[0076] Concurrently with various blocks of the method 800 (e.g., stemming from block 804 and executing with one or more of blocks 806-810), in determination block 812, the apparatus may determine whether it has or is receiving more raw data. In an embodiment, the raw data may be retained or received at the apparatus (e.g., a machine learning accelerator) from the raw data source device. The apparatus may

have or be receiving more raw data when the apparatus is retaining already received raw data, such as in a feature buffer before the key feature vectors are extracted, or when the apparatus is receiving additional raw data from the raw data source device in a streaming or as needed manner. In response to determining that the apparatus has or is receiving raw data (i.e., determination block 812 "Yes"), the apparatus may extract key feature vectors from the raw data in block 804.

[0077] In response to determining that the apparatus does not have or is not receiving raw data (i.e., determination block 812 "No"), or stemming from another block of the method 800 (e.g., block 810), the apparatus may determine whether it has any feature vectors remaining in determination block 812. In an embodiment, the feature vectors may be retained by the machine learning accelerator, for example in the vector units as the vector units operate using the feature vectors.

[0078] In response to determining that the apparatus has remaining feature vectors (i.e., determination block 812 "Yes"), the apparatus may generate a partial output of the processed raw data in block 808.

[0079] In response to determining that the apparatus does not have remaining feature vectors (i.e., determination block 812 "No"), the apparatus may determine whether it has any key feature vectors remaining in determination block 814. In an embodiment, the key feature vectors may be retained by the machine learning accelerator, for example in the key feature queue of the feature buffer.

[0080] In response to determining that the apparatus has remaining key feature vectors (i.e., determination block 814 "Yes"), the apparatus may generate feature vectors from the key feature vectors in block 806.

[0081] In response to determining that the apparatus does not have remaining key feature vectors (i.e., determination block 814 "No"), the apparatus may deactivate a set of vector units associated with a feature buffer lacking key feature vectors. In an embodiment, the feature buffer associated with the vector units to be deactivated and also lacking key feature vectors may also be deactivated.

[0082] FIG. 9 illustrates an embodiment method 900 for extracting a key feature vector from raw data. The method 900 may be executed as part of block 708 in the method 700 or as part of block 802 in the method 800. The method 900 may be implemented in a computing device in software executing in a processor, in general purpose hardware, or dedicated hardware, such as a processor executing software within a machine learning accelerator that includes other individual components. In order to encompass the alternative configurations enabled in the various embodiments, the hardware implementing the method 900 is referred to herein as an apparatus.

[0083] In optional block 902, the apparatus of the computing device may receive key feature vector parameters for raw data processing. In an embodiment, the key feature vector parameters may include a size parameter and a stride parameter. In an embodiment, the key feature vector parameters may be predetermined or determined based on a type of machine learning, a granularity for processing the raw data, and/or a number and capability of the vector units of the machine learning accelerator.

[0084] In block 904, the apparatus may identify key features of the raw data. The apparatus may apply the key feature vector parameters to a block of received raw data to

identify a key feature of the raw data. In an embodiment, the key features of the raw data may be defined by a two dimensional matrix of raw data values from the raw data, for example a two dimensional matrix starting at a beginning of the block of raw data. Each successive key feature of the raw data may be identified using the same size parameter, or the same two dimensional matrix, applied to a different location in the raw data. The location of each successive key feature may be determined based on the location of the previous key feature and the stride parameter. The stride parameter may indicate where to locate a successive key feature based on the location of the previous key feature by indicating a number of units from the previous location to apply the size parameter to determine the successive key feature. In an embodiment, the size and stride parameters may be defined such that successive key features of the raw data avoid including raw data from a previous key feature of the raw data. In an embodiment, the stride parameter may equal one of the dimensions of the size parameter.

[0085] In block 908, the apparatus may translate the key features to key feature vectors. The apparatus may be configured to translate the key features to key feature vectors in a variety of ways. In an embodiment, translating the key features to key feature vectors may include appending successive rows of the two dimensional matrix of raw data to a first or previous row of the two dimensional matrix, such that the translated key feature vector represents an array-like structure of the raw data of the two dimensional matrix. However, any translation of the key features to key feature vectors may be used, so long as the key feature vectors are usable to generate feature vectors that can be properly processed to produce the output matrix. The method 900 may return to the method 800 and buffer the key feature vectors in block 804.

[0086] FIG. 10 illustrates an embodiment method 1000 for generating a feature from a key feature vectors. The method 1000 may be executed as part of block 802, as part of block 806 in the method 800. The method 1000 may be implemented in a computing device in software executing in a processor, in general purpose hardware, or dedicated hardware, such as a processor executing software within a machine learning accelerator that includes other individual components. In order to encompass the alternative configurations enabled in the various embodiments, the hardware implementing the method 1000 is referred to herein as an apparatus.

[0087] In optional block 1002, the apparatus of the computing device may receive feature generation parameters for raw data processing, such as the size of the feature vector. In an embodiment, the parameters for raw data processing may depend on various factors, including the machine learning algorithms implemented, the size and/or complexity of the raw data, the power and/or performance requirements of the computing device, the processing requirements for the raw data, the number and capability of the vector units of the machine learning accelerator, and the configuration of the key feature vectors. In an embodiment, the size of the feature vector may equal the size of the key feature vector.

[0088] In block 1004, the apparatus may use the top key feature vector, for example from the top of the key feature queue, as a feature vector. In an embodiment, the generation of a feature vector may not require any manipulation of the key feature vector, and may use the key feature vector data as is to generate the feature vector.

[0089] In determination block 1006, the apparatus may determine whether multiple key feature vectors remain. In an embodiment, the key feature vectors may be retained by the apparatus in the key feature queue of the machine learning accelerator. Different locations in the key feature queue may be loaded with a key feature vector. As the key feature vectors are used, the location in the key feature queue may be emptied or nullified. Thus, under various circumstances the key feature queue may contain no key feature vectors, a single key feature vector, or multiple key feature vectors.

[0090] In response to determining that multiple key feature vectors do not remain (i.e., determination block 1006 “No”), the apparatus may discard or nullify the top key feature vector in block 1014. The method 1000 may return to the method 800 and generate a partial output of the processed raw data in block 808.

[0091] In response to determining that multiple key feature vectors do remain (i.e., determination block 1006 “Yes”), the apparatus may determine whether to combine the key feature vectors in determination block 1008. The determination whether to combine key vectors may depend on whether a key feature vector has or a combination of key feature vectors have already been used to generate a feature vector.

[0092] In an embodiment, feature vectors may be generated by using a single key feature vector, as in block 1004, or by combining multiple key feature vectors. Combining key feature vectors may allow the apparatus to generate feature vectors that are not created from the key feature vectors when they are used alone to generate the feature vector. In an embodiment, the extraction of key features and translation to key feature vectors may leave out combinations of raw data that may be needed to properly process the raw data to produce the output matrix. The combination of key feature vectors may allow the computing device to recombine those combinations of raw data without having to execute costly reads of the raw data to create each combination as a separate key feature vector. Therefore, depending on the extraction and translation of the key feature vectors, different combinations of key feature vectors may produce desired feature vectors.

[0093] In an embodiment, the apparatus may determine not to combine key feature vectors when the top key feature vector has not been used in generating a feature key, and to combine key feature vectors when the top key feature vector has been used in generating a feature key. In an embodiment, the apparatus may determine not to combine key feature vectors when the key feature vectors have been previously combined.

[0094] In response to determining not to combine the key feature vectors (i.e., determination block 1008 “No”), the apparatus may discard or nullify the top key feature vector in optional block 1010. In block 1012, the apparatus may assign the next key feature vector in the key feature queue as the top key feature vector. In an embodiment, rather than discarding or nullifying the top key feature vector in a circular key feature queue model, the apparatus may also assign the previous top key feature vector to another position in the key feature queue. In block 1004, the apparatus may use the top key feature as a feature vector.

[0095] In response to determining to combine the key feature vectors (i.e., determination block 1008 “Yes”), the apparatus may combine the key feature vectors to generate

a feature vector in block 1014. In an embodiment, apparatus may combine any of the key feature vectors, such as the top key feature vector and a next key feature vector. The combination of the key feature vectors may occur in various manners. For example, the combination of the key feature vectors may include the combination of successive key feature vectors such that the combination creates a data set of a key feature not identified by the apparatus such that the key feature would have included data from both of the successive key features. As discussed herein, combining the key features to create data sets of unidentified key features allows the computing device to avoid costly reads of the raw data to identify such key features.

[0069] In optional block 1010, the apparatus may discard the top key feature vector. In block 1012, the apparatus may assign the next key feature vector in the key feature queue as the top key feature vector. In block 1004, the apparatus may use the top key feature as a feature vector.

[0067] FIG. 11 illustrates an embodiment method 1100 for combining a top key feature vector and a next key feature vector as a feature. The method 1100 may be executed as part of block 1014 in the method 1000. The method 1100 may be implemented in a computing device in software executing in a processor, in general purpose hardware, or dedicated hardware, such as a processor executing software within a machine learning accelerator that includes other individual components. In order to encompass the alternative configurations enabled in the various embodiments, the hardware implementing the method 1100 is referred to herein as an apparatus.

[0068] In block 1102, the apparatus of the computing device may select at least two key feature vectors to generate a feature vector. In an embodiment, the key feature vectors may include at least the current key feature vector, which may be the top key feature vector, and a successive key feature vector in the key feature queue.

[0069] In block 1104, the apparatus may select key feature vector positions to shuffle to generate the feature vector. The key feature vector positions may be selected from each of the selected key feature vectors such that each position selected among the various selected key feature vectors represents a different location in the raw data that is not represented by another selected key feature position. The selected key feature positions may also represent an unidentified key feature of the raw data, for example a data set of the raw data with the same two dimensional characteristics as an identified key feature and spanning multiple identified key features.

[0110] In block 1106, the apparatus may write the selected key feature positions to the current key feature vector. In an embodiment, writing the selected key feature positions to the current key feature vector may be accomplished by writing the selected key feature positions in an order that would result from the translation of the unidentified key feature represented by the selected key feature positions to a key feature vector.

[0100] The method 1100 may return to the method 1000 and the apparatus may discard the top key feature vector in optional block 1010, or the apparatus may assign the next key feature vector in the key feature queue as the top key feature vector in block 1012.

[0102] FIGS. 12A-12C illustrate an example of a process flow for extracting a key feature vector from raw data and generating a feature vector from the key feature vector for

implementing an embodiment. This is only an example and not limiting in any manner, particularly with respect to the size, number, configuration, or content of the raw data, key features, key feature vectors, and feature vectors.

[0103] FIG. 12A illustrates an example raw data set 1200 from which key features may be identified, and key feature vectors and feature vectors may be generated as described further herein with reference to FIGS. 12B-12C. Each location in the raw data set 1200 may represent a separate unit of data. In different raw data sets 1200, the units of data may vary, for example the units may be a bit or a byte of data.

[0104] FIG. 12B illustrates the apparatus identifying the key features 1206, 1208 of various portions of the raw data set 1202, 1204 received by different feature buffers. For this example, the key feature vector parameters may be defined as a two-by-four matrix and a two null stride. Based on these key feature vector parameters, key feature 1206 (e.g., 1206a-1206c), 1208 (e.g., 1208a-1208c) may be identified to represent the entire raw data set 1200. Each key feature may be translated into a key feature vector 1210, 1212 (e.g., key feature 1206a may be translated into key feature vector 1210a; key feature 1206b may be translated into key feature vector 1210b; key feature 1206c may be translated into key feature vector 1210c; key feature 1208a may be translated into key feature vector 1212a; key feature 1208b may be translated into key feature vector 1212b; and key feature 1208c may be translated into key feature vector 1212c). The key feature vectors 1210, 1212 may be held in their respective key feature queues.

[0105] As illustrated in FIG. 12C, an apparatus of the apparatus may generate a feature vector 1214a, 1216a from the top key feature vector 1210a, 1212a from each key feature queue. In an embodiment, this particular generation of feature vectors 1214a, 1216a may include generating the feature vectors 1214a, 1216a without manipulating of the key feature vectors 1210a, 1212a. The generated feature vectors 1214a, 1216a may contain data corresponding to raw data of respective key features 1206a, 1208a.

[0106] FIG. 12D illustrates that the apparatus of the computing device may combine the top key feature vectors 1210a, 1212a with a next key feature vector 1210b, 1212b to generate another feature vector 1214b, 1216b. The data selected from the top key feature vectors 1210a, 1212a and the next key feature vectors 1210b, 1212b may correspond with previously unidentified key features 1206d, 1208d. In this example, the previously unidentified key features 1206d, 1208d may be such that they span previously identified key features 1206a, 1206b and 1208a, 1208b, respectively.

[0107] FIG. 12E illustrates when that the top key feature vectors 1210a, 1212a are no longer in the key feature queue, and previously next key feature vectors 1210b, 1212b have been reassigned as top key feature vectors 1210c, 1212c. Much like at FIG. 12C, the apparatus may generate a feature vector 1214c, 1216c from the top key feature vector 1210c, 1212c from each key feature queue, without manipulating the top key feature vector 1210c, 1212c such that they contain data corresponding to raw data of respective key features 1206b, 1208b.

[0108] Much like FIG. 12D, in the example illustrated in FIG. 12E, the apparatus of the computing device may combine the top key feature vectors 1210c, 1212c with a next key feature vector 1210d, 1212d to generate another

feature vector $1214d$, $1216d$, such that the data of each feature vector $1214d$, $1216d$ may correspond with previously unidentified key features $1206c$, $1206c$.

[0119] Much like in FIG. 12f, in the example illustrated in FIG. 12g, the top key feature vectors $1210e$, $1212b$ are no longer in the key feature queue, and previously next key feature vectors $1210e$, $1212c$ have been re-signed as top key feature vectors $1210e$, $1212c$. The apparatus of the apparatus may generate a feature vector $1214e$, $1216e$ from the top key feature vector $1210e$, $1212c$ such that they contain data corresponding to raw data of respective key features $1206e$, $1208e$.

[0120] FIG. 13g illustrates an example vector unit 304 suitable for implementing an embodiment. The vector unit 304 may be connected between an associated feature buffer 306, the weight storage device 312, and the output buffer 308. The vector unit 304 may receive feature vectors from the associated feature buffer 306 as they are generated and output to the vector unit 304. As described herein, the vector unit may be one of a number of vector units 304 associated with the feature buffer 306 and receiving the feature vector.

[0121] Portions of the received feature vectors may be provided to at least one process unit 1302, which may include an arithmetic logic unit (ALU) or other programmable logic device, for executing operations, such as basic linear algebra subprogram operation, using the portions of the feature vectors. The vector unit 304 may also receive a weight factor from the weight storage device 312.

[0122] The vector unit 304 may include at least one local weight vector register 1300 configured to temporarily store the received weight factor, and output the weight factor to the process unit 1302 for use in executing its operations using the received feature vector. In an embodiment, the weight factor may include a single value or a number of values, and may be configured a vector, such as a vector with a number of positions that may correspond to a number of process units 1302 in the vector unit 304. Each local weight vector register 1300 may be associated with a particular process unit 1302, and may output all or part of the weight factor to the associated process unit 1302.

[0123] The process units 1302 may execute an operation using the received feature vector and the received weight factor to generate a pre-partial output of the input matrix. The process units 1302 may output the pre-partial output to at least one partial output vector register 1304, which may be configured to temporarily store the received the pre-partial input, and combine multiple pre-partial outputs from the various process units 1302 into a partial output vector. The partial output vector registers 1304 may store the pre-partial outputs until receiving a pre-partial output from all of the process units 1302. The partial output vector registers 1304 may output the pre-partial outputs as a partial output vector to the output buffer 308.

[0124] FIG. 14g illustrates an embodiment method 1400 for generating a partial output of a processed raw data. The method 1400 may be executed as part of block, as part of block 808 in the method 800. The method 1400 may be implemented in a computing device in software executing in a processor, in general purpose hardware, or dedicated hardware, such as a processor executing software within a machine learning accelerator that includes other individual components. In order to encompass the alternative configura-

tions enabled in the various embodiments, the hardware implementing the method 1400 is referred to herein as an apparatus.

[0125] In block 1402, the apparatus of the computing device may receive the weight factor. As discussed herein, the weight factor may be a single weight value or a vector of weight values, and may be the same or different for each or a set of vector units. The weight factor received may depend on the type of machine learning accelerated by the machine learning accelerator.

[0126] In block 1404, the apparatus may store the received weight factor. The weight factor may be stored temporarily by the apparatus, for example in a weight buffer or weight vector register, at least until the apparatus is prepared to use the weight factor in generating the output matrix. In an embodiment, the weight factor may change for operations with different feature vectors, or the same or different raw data, and a new weight factor may be received and stored to be used in the operations. In an embodiment, the weight factors may be persistent for operations with different feature vectors of the same or different raw data, and the same weight factor may be retained and repeatedly used in various operations.

[0127] In block 1406, the apparatus may receive feature vectors. For example, the vector units may receive feature vectors from their associated feature buffers. Various vector units may receive different feature vectors depending on the feature buffer with which they are associated and the raw data received by the associated feature buffer. The apparatus may receive the feature vectors in a streaming or as-needed manner.

[0128] In block 1408, the apparatus may generate a pre-partial output using the weight factor and the feature vector. In an embodiment, the vector units may execute a variety of operations, including basic linear algebra subprogram operations, using the received weight factors and the feature vectors. The vector units may use any combination of the entire or part of the weight factor and the entire or part of the feature vector it receives in the operation to generate the pre-partial output.

[0129] In block 1410, the apparatus may store the pre-partial output. The pre-partial output may be only part of the partial output of the output matrix. In an embodiment, the partial output may include multiple pre-partial outputs generated from multiple vector units, such as vector units associated with the same feature buffer. In an embodiment, the partial output may include multiple pre-partial outputs generated from multiple process elements, such as process element belonging to the same vector unit. The apparatus may store each pre-partial output until there are sufficient pre-partial outputs stored to compose a partial output of the output matrix.

[0130] In block 1412, the apparatus may combine the pre-partial outputs to compose the partial output. The method 1400 may return to the method 800 and output the partial output of the processed raw data in block 810.

[0131] FIG. 15 illustrates an example vector unit 304 suitable for implementing an embodiment. The vector unit 304 may be connected between an associated feature buffer 306, the weight storage device 312, and the output buffer 308. In an embodiment, the feature buffer 306 may also be connected to a raw data source device, such as a random access memory 1500. The vector unit 304 may receive feature vectors from the associated feature buffer 306 as they

are generated and output to the vector unit 304. As described herein, the vector unit may be one of a number of vector units 304 associated with the feature buffer 306 and receiving the feature vector. The received feature vectors may be temporarily stored in at least one input register 1502. [0122] A kernels (or weight) first-in first-out (FIFO) register 1504 may receive raw data from the raw data source device. The kernels (or weight) first-in first-out register 1504 may provide at least one kernel (or weight) register 1506 with data from the received raw data in a first-in first-out manner. The kernels (or weight) register 1506 may act as a filter for the data from the raw data, limiting the data available for use based on the size of the kernels (or weight) register 1506, thereby generating kernels (or weights) for use in generating a pre-partial output. In an embodiment, the kernels (or weight) may include portions of the raw data.

[0123] The received feature vectors and the kernels (or weight) may be provided to a process unit 1302, which may include an arithmetic logic unit (ALU), a multiply-accumulate (MAC) unit, or other programmable logic device, for executing operations, such as basic linear algebra sub-program operation, using the feature vectors and the kernels (or weight). The process unit 1302 may execute its operation and output a pre-partial output to at least one partial output vector register 1304, which may be configured to temporarily store the received pre-partial output, and combine multiple pre-partial outputs from the various process units 1302 into a partial output vector.

[0124] The partial output vector register 1304 may store the pre-partial outputs until receiving a pre-partial output from all of the process units 1302. The partial output vector registers 1304 may output the pre-partial outputs as a partial output vector to the output buffer 308.

[0125] FIG. 16 illustrates an embodiment method 1600 for generating a partial output of a processed raw data. The method 1600 may be executed as part of block 1610 in method 800. The method 1600 may be implemented in a computing device in software executing in a processor, in general purpose hardware, or dedicated hardware, such as a processor executing software within a machine learning accelerator that includes other individual components. In order to encompass the alternative configurations enabled in the various embodiments, the hardware implementing the method 1600 is referred to herein as an apparatus.

[0126] In block 1602, the apparatus of the computing device may receive feature vectors and raw data. In an embodiment, the feature vectors may be received in the input registers of the vector units from the feature buffers with which the vector units are associated, and the raw data may be received in the kernels (or weight) first-in first-out register from the raw data source device. Different kernels (or weight) first-in first-out register for different vector units may receive the same or different portions of the raw data. The feature vectors and raw data may be received in a streaming or as needed manner.

[0127] In block 1604, the apparatus may store the received feature buffers. Temporary storage of the received feature buffers may be implemented to allow for completion of previous operation execution and filtering of the raw data.

[0128] In block 1606, the apparatus may filter the raw data. In an embodiment, filtering the raw data may include selecting a portion of the received raw data or filter location,

to apply to the operation with the feature vector. In embodiments where different kernels (or weight) first-in first-out register for different vector units may receive the same portions of the raw data using different filter locations may result in different filter values. In embodiments where different kernels (or weight) first-in first-out register for different vector units may receive different portions of the raw data using the same filter locations may result in different filter values.

[0129] In block 1608, the apparatus may generate a pre-partial output using the kernel (or weight) and the feature vector. In an embodiment, the vector units may execute a variety of operations, including basic linear algebra subprogram operations, using the filtered kernel (or weight) and the received feature vectors. The vector units may use any combination of the kernel (or weight) factor and the entire or part of the feature vector it receives in the operation to generate the pre-partial output.

[0130] In block 1610, the apparatus may store the pre-partial output. The pre-partial output may be only part of the partial output of the output matrix. In an embodiment, the partial output may include multiple pre-partial outputs generated from multiple vector units, such as vector units associated with the same feature buffer. In an embodiment, the partial output may include multiple pre-partial outputs generated from multiple process elements, such as process elements belonging to the same vector unit. The apparatus may store each pre-partial output until there are sufficient pre-partial outputs stored to compose a partial output of the output matrix.

[0131] In block 1612, the apparatus may combine the pre-partial outputs to compose the partial output. The method 1600 may return to the method 800 and output the partial output of the processed raw data in block 810.

[0132] FIGS. 17A-17D illustrate an example of a process flow for generating a kernel using filtered raw data. This is only an example and not limiting in any manner, particularly with respect to the size, number, configuration, or content of the raw data, feature vectors, and kernel (or weight factors).

[0133] FIG. 17A illustrates an example raw data set 1700 from which feature vectors may be generated and kernel (or weight factors) may be filtered, as described further herein with reference to FIGS. 17B-17D. Each location in the raw data set 1700 may represent a separate unit of data. In different raw data sets 1700, the units of data may vary, for example the units may be a bit or a byte of data. In this example, like shading may represent a different data channel from other shading. For example, the data channels may represent different pixel colors for raw image or video data. FIG. 17A illustrates an example filter queue 1702 having a set of filter locations for filtering data from the raw data set 1700.

[0134] FIG. 17B illustrates an application of a first filter location 1704a in the raw data set 1700 that may generate a first filtered portion 1706a for a particular vector unit. Similarly, in the continued examples shown in FIGS. 17C and 17D, the application of other filter locations 1704b, 1704c, to the raw data set 1700 may generate other filtered portions 1706b, 1706c for other vector units. The number of filter locations and the amount of data they extract from the raw data set in these examples is not limiting and the number of filter locations and the amount of data they extract may vary based upon various factors, including the machine learning algorithms implemented, the size and/or complexity of

the raw data, the power and/or performance requirements of the computing device, and the processing requirements for the raw data.

[0135] FIGS. 18A-18D illustrate an example of a process flow for generating a pre-partial output using a kernel and feature vector. This is only an example and not limiting in any manner, particularly with respect to the size, number, configuration, or content of the raw data, feature vectors, and kernels (or weight factors). FIGS. 18A-18D illustrates implementation of an operation using three vector units, such as multiply-accumulate (MAC) units 1808 (e.g., 1808a-1808c).

[0136] FIG. 18A shows the implementation of the operation using filtered data 1800a and a feature vector 1802a at a first time. The filter data 1800a may represent the data at various locations of the respective filter queues 1702 of the multiply-accumulate units 1808 (e.g., filter queue 1702a of multiply-accumulate unit 1808a; filter queue 1702b of multiply-accumulate unit 1808b; and filter queue 1702c of multiply-accumulate unit 1808c). In particular, the filter data 1800a may represent the data at the top of the respective filter queues 1702 for the first time (e.g., filter location 1 of 1804a of filter queue 1702a; filter location 18 of 1804c of filter queue 1702b; and filter location 146 of 1804c of filter queue 1702c). At the first time, the multiply-accumulate units 1808 may use the feature queue 1802a and the kernels (or weight factors) of the respective filters 1806 for each of the multiply-accumulate units 1808 (e.g., filter 1806a of multiply-accumulate unit 1808a; filter 1806b of multiply-accumulate unit 1808b; and filter 1806c of multiply-accumulate unit 1808c) to execute the operation.

[0137] Each filter 1808 may correspond to a particular filter location 1804 in the filter queue 1702 of the corresponding multiply-accumulate unit 1808 (e.g., filter location 10 of 1804a for the filter queue 1702a; and filter 1806a, filter location 146 of 1804c for the filter queue 1702b; and for filter 1806c, filter location 16 of 1804c for the filter queue 1702c; and for filter 1806c). The kernels (or weight factors) of the respective filters 1806 may correspond to the data at the particular filter location 1804 in the filter queue 1702 of the corresponding multiply-accumulate unit 1808. At the first time the operation may use data from the unshaded data channel.

[0138] Similarly, FIG. 18B illustrates the implementation of the operation using filtered data 1800b and a feature vector 1802b at a second time. At the second time the operation may use data from the stippled data channel. Each multiply-accumulate unit 1808 may have its respective filter 1806 with kernel (or weight factor) values that may correspond to the data at the particular filter location 1804 in the filter queue 1702 for the multiply-accumulate unit 1808 (e.g., multiply-accumulate unit 1808a may use the kernel (or weight factor) from filter 1806a corresponding to the data at filter location 1804a of filter queue 1702a; multiply-accumulate unit 1808b may use the kernel (or weight factor) from filter 1806b corresponding to the data at filter location 1804b of filter queue 1702b; and multiply-accumulate unit 1808c may use the kernel (or weight factor) from filter 1806c corresponding to the data at filter location 1804c of filter queue 1702c).

[0139] FIG. 18C illustrates the implementation of the operation using filtered data 1800c and a feature vector 1802c at a third time. At the third time the operation may use data from the more heavily stippled data channel. Each

multiply-accumulate unit 1808 may have its respective filter 1806 with kernel (or weight factor) values that may correspond to the data at the particular filter location 1804 in the filter queue 1702 for the multiply-accumulate unit 1808 (e.g., multiply-accumulate unit 1808a may use the kernel (or weight factor) from filter 1806a corresponding to the data at filter location 1804a of filter queue 1702a; multiply-accumulate unit 1808b may use the kernel (or weight factor) from filter 1806b corresponding to the data at filter location 1804b of filter queue 1702b; and multiply-accumulate unit 1808c may use the kernel (or weight factor) from filter 1806c corresponding to the data at filter location 1804c of filter queue 1702c).

[0140] FIG. 18D illustrates an example of a partial output 1810 of one of the multiply-accumulate units 1808 (e.g., 1808a) after executing the operation for the feature vector 1802 and the kernels (or weight factors) of the corresponding filters 1806 for all of the available channels of data. At each time, for each channel of data, the multiply-accumulate units 1808 may store the result of the executed operation and combine it with the other results to produce a partial output 1810, which may be output after the completion of the executions based on certain parameters, including a designated number of executions. In an embodiment, the partial output 1810 may be output to the partial output vector register associated with the multiply-accumulate units 1808.

[0141] FIG. 19 illustrates an example of a process flow for generating a feature vector using an arbiter to assign addresses to raw data. This is only an example and not limiting in any manner, particularly with respect to the size, number, configuration, or content of the raw data and feature vectors. In this example, the raw data set 1700 may be received by an arbiter 1900, for example via one or more list-in-list-out queues that may read the rows of the raw data set 1700. The arbiter 1900 may assign addresses from multiple feature vectors 1902 (e.g., 1902a-1902c), to each unit of data of the raw data set grouped by data channel. As such, the arbiter 1900 may be used instead of the feature buffers of the machine learning accelerator.

[0142] The various embodiments (including, but not limited to, embodiments discussed above with reference to FIGS. 1-19) may be implemented in a wide variety of computing systems, which may include an example mobile computing device suitable for use with the various embodiments illustrated in FIG. 20. The mobile computing device 2000 may include processor 2002 coupled to a touchscreen controller 2004 and an internal memory 2006. The processor 2002 may be one or more multicore integrated circuits designated for general or specific processing tasks. The internal memory 2006 may be volatile or non-volatile memory, and may also be static and/or encrypted memory, or may be a combination thereof. Examples of memory types that can be leveraged include but are not limited to DDR, LPDDR, QDDR, WDDR, RAM, SRAM, DRAM, PRAM, R-RAM, M-RAM, S-RAM, and embedded DRAM. The touchscreen controller 2004 and the processor 2002 may also be coupled to a touchscreen panel 2012, such as a resistive-sensing touchscreen, capacitive-sensing touchscreen, infrared-sensing touchscreen, etc. Additionally, the display of the computing device 2000 need not have touch screen capability.

[0143] The mobile computing device 2000 may have one or more radio signal transceivers 2008 (e.g., Peanut, Blu-

good, Zigbee, Wi-Fi, RF radio and antenna 2010, for sending and receiving communications, coupled to each other and/or to the processor 2002. The transceivers 2008 and antenna 2010 may be used with the above-mentioned circuitry to implement the various wireless transmission protocol stacks and interfaces. The mobile computing device 2000 may include a cellular network wireless system chip 2016 that enables communication via a cellular network and is coupled to the processor.

[0144] The mobile computing device 2000 may include a peripheral device connection interface 2018 coupled to the processor 2002. The peripheral device connection interface 2018 may be singularly configured to accept one type of connection, or may be configured to accept various types of physical and communication connections, common or proprietary, such as USB, FireWire, Thunderbolt, or PCIe. The peripheral device connection interface 2018 may also be coupled to a similarly configured peripheral device connection port (not shown).

[0145] The mobile computing device 2000 may also include speakers 2014 for providing audio outputs. The mobile computing device 2000 may also include a housing 2020, constructed of a plastic, metal, or a combination of materials, for containing all or some of the components discussed herein. The mobile computing device 2000 may include a power source 2022 coupled to the processor 2002, such as a disposable or rechargeable battery. The rechargeable battery may also be coupled to the peripheral device connection port to receive a charging current from a source external to the mobile computing device 2000. The mobile computing device 2000 may also include a physical button 2024 for receiving user inputs. The mobile computing device 2000 may also include a power button 2026 for turning the mobile computing device 2000 on and off.

[0146] The various embodiments discussed above, including, but not limited to, embodiments discussed above with reference to FIGS. 1-19, may be implemented in a wide variety of computing systems, which may include a variety of mobile computing devices, such as a laptop computer 2100 illustrated in FIG. 21. Many laptop computers include a touchpad/touch surface 2117 that serves as the computer's pointing device, and thus may receive drag, scroll, and flick gestures similar to those implemented on computing devices equipped with a touch screen display and described above. A laptop computer 2100 will typically include a processor 2111 coupled to volatile memory 2112 and a large capacity nonvolatile memory, such as a disk drive 2113 or flash memory. Additionally, the computer 2100 may have one or more antenna 2108 for sending and receiving electromagnetic radiation that may be connected to a wireless data link and/or cellular telephone transceiver 2116 coupled to the processor 2111. The computer 2100 may also include a floppy disc drive 2114 and a compact disc (CD) drive 2115 coupled to the processor 2111. In a notebook configuration, the computer housing includes the keyboard 2118 and the display 2119, all coupled to the processor 2111. Other configurations of the computing device may include a computer mouse or trackball coupled to the processor (e.g., via a USB input) as are well known, which may also be used in conjunction with the various embodiments.

[0147] The various embodiments (including, but not limited to, embodiments discussed above with reference to FIGS. 1-19) may be implemented in a wide variety of

computing systems, which may include any of a variety of commercially available servers for compressing data in server cache memory. An example server 2200 is illustrated in FIG. 22. Such a server 2200 typically includes one or more multi-core processor assemblies 2201 coupled to volatile memory 2202 and a large capacity nonvolatile memory, such as a disk drive 2204. As illustrated in FIG. 22, multi-core processor assemblies 2201 may be added to the server 2200 by inserting them into the slots of the assembly. The server 2200 may also include a floppy disc drive, compact disc (CD) or digital versatile disc (DVD) disc drive 2206 coupled to the processor 2201. The server 2200 may also include network access ports 2203 coupled to the multi-core processor assemblies 2201 for establishing network interface connections with a network 2205, such as a local area network coupled to other broadcast system computers and servers, the Internet, the public switched telephone network, and/or a cellular data network (e.g., CDMA, TDMA, GSM, PCS, 3G, 4G, 5G), or any other type of cellular data network.

[0148] Computer program code or "program code" for execution in a programmable processor for carrying out operations of the various embodiments may be written in a high level programming language such as C, C++, C#, Smalltalk, Java, JavaScript, Visual Basic, a Structured Query Language (e.g., Transact-SQL), Perl, or in various other programming languages. Program code or programs stored on a computer readable storage medium as used in this application may refer to machine language code (such as object code) whose format is understandable by a processor.

[0149] The foregoing method descriptions and the process-flow diagrams are provided merely as illustrative examples and are not intended to require or imply that the operations of the various embodiments must be performed in the order presented. As will be appreciated by one of skill in the art the order of operations in the foregoing embodiments may be performed in any order. Words such as "thereafter," "then," "next," etc. are not intended to limit the order of the operations; these words are simply used to guide the reader through the description of the methods. Further, any reference to claim elements in the singular, for example, using the articles "a," "an," or "the" is not to be construed as limiting the element to the singular.

[0150] The various illustrative logical blocks, modules, circuits, and algorithm operations described in connection with the various embodiments may be implemented as electronic hardware, computer software, or combinations of both. To clearly illustrate this interchangeability of hardware and software, various illustrative components, blocks, modules, circuits, and operations have been described above generally in terms of their functionality. Whether such functionality is implemented as hardware or software depends upon the particular application and design constraints imposed on the overall system. Skilled artisans may implement the described functionality in varying ways for each particular application, but such implementation decisions should not be interpreted as causing a departure from the scope of the claims.

[0151] The hardware used to implement the various illustrative logics, logical blocks, modules, and circuits described in connection with the embodiments disclosed herein may be implemented or performed with a general purpose processor, a digital signal processor (DSP), an application-specific integrated circuit (ASIC), a field pro-

grammable gate array (PGA) or other programmable logic device, discrete gate or transistor logic, discrete hardware components, or any combination thereof designed to perform the functions described herein. A general-purpose processor may be a microprocessor, but, in the alternative, the processor may be any conventional processor, controller, microcontroller, or state machine. A processor may also be implemented as a combination of computing devices, e.g., a combination of a DSP and a microprocessor, a plurality of microprocessors, one or more microprocessors in conjunction with a DSP core, or any other such configuration. Alternatively, some operations or methods may be performed by circuitry that is specific to a given function.

[0152] In one or more embodiments, the functions described may be implemented in hardware, software, firmware, or any combination thereof. If implemented in software, the functions may be stored as one or more instructions or code on a non-transitory computer-readable medium or a non-transitory processor-readable medium. The operations of a method or algorithm disclosed herein may be embodied in a processor-executable software module that may reside on a non-transitory computer-readable or processor-readable storage medium. Non-transitory computer-readable or processor-readable storage media may be any storage media that may be accessed by a computer or a processor. By way of example but not limitation, such non-transitory computer-readable or processor-readable media may include RAM, ROM, EEPROM, FLASH memory, CD-ROM or other optical disk storage, magnetic disk storage or other magnetic storage devices, or any other medium that may be used to store desired program code in the form of instructions or data structures and that may be accessed by a computer. Disk and disc, as used herein, includes compact disc (CD), laser disc, optical disc, digital versatile disc (DVD), floppy disk, and Blu-ray disc where disks usually reproduce data magnetically, while discs reproduce data optically with lasers. Combinations of the above are also included within the scope of non-transitory computer-readable and processor-readable media. Additionally, the operations of a method or algorithm may reside as one or more combinations of sets of codes and/or instructions on a non-transitory processor-readable medium and/or computer-readable medium, which may be incorporated into a computer program product.

[0153] The preceding description of the disclosed embodiments is provided to enable any person skilled in the art to make or use the claims. Various modifications to these embodiments will be readily apparent to those skilled in the art, and the generic principles defined herein may be applied to other embodiments without departing from the scope of the claims. Thus, the present disclosure is not intended to be limited to the embodiments shown herein but is to be accorded the widest scope consistent with the following claims and the principles and novel features disclosed herein.

What is claimed is:

1. A method of accelerating machine learning on a computing device, comprising:
receiving raw data from a raw data source device;
identifying key features as two dimensional matrices of the raw data such that the key features are mutually exclusive from each other;
translating the key features into key feature vectors;
- generating a feature vector from at least one of the key feature vectors;
- receiving a first partial output resulting from an execution of a basic linear algebra subprogram (BLAS) operation using the feature vector and a weight factor; and
combining the first partial output with a plurality of partial outputs to produce an output matrix.

2. The method of claim 1, wherein identifying key features as two dimensional matrices of the raw data such that the key features are mutually exclusive from each other comprises:
identifying a first key feature as a first two dimensional matrix of a designated size; and
identifying a second key feature as a second two dimensional matrix of the designated size a designated number of units from the first key feature.
3. The method of claim 1, wherein generating a feature vector from at least one of the key feature vectors comprises:
selecting a top key feature vector from a key feature vector queue; and
using the top key feature vector as the feature vector.
4. The method of claim 1, wherein generating a feature vector from at least one of the key feature vectors comprises:
selecting a top key feature vector from a key feature vector queue;
selecting a next key feature vector from the key feature vector queue;
selecting top key feature vector positions and next key feature vector positions; and
combining the selected top key feature vector position and the selected next key feature vector positions into the feature vector.
5. The method of claim 4, wherein:
selecting top key feature vector positions and next key feature vector positions comprises selecting the top key feature vector positions and the next key feature vector positions such that each is the selected top key feature vector position and the selected next key feature vector positions represent mutually exclusive locations from each other in the raw data and represent unidentified key feature of raw data that spans a plurality of the identified key features of the raw data; and
combining the selected top key feature vector position and the selected next key feature vector positions into the feature vector comprises combining the selected top key feature vector position and the selected next key feature vector positions into the feature vector such that the feature vector is configured like a key feature vector of the unidentified key feature.
6. The method of claim 1, further comprising:
activating a set of vector units upon receiving the raw data at a feature buffer associated with the set of vector units, wherein the set of vector units is mapped to the output matrix;
executing the BLAS operation by each vector unit of the set of vector units; and
computing at least one partial output by each vector unit.
7. The method of claim 6, further comprising:
determining whether any feature vectors return for use in an execution of the BLAS operation by the set of vector units; and

- deactivating the set of vector units in response to determining that no feature vectors remain for use in an execution of the BLAS operation by the set of vector units.
- 8.** The method of claim 1, wherein receiving raw data from a raw data source device comprises receiving streaming raw data from the raw data source device.
- 9.** An apparatus configured to accelerate machine learning on a computing device, comprising:
- a raw data source device; and
 - a vectorization unit communicatively connected to the raw data source device, and configured to perform operations comprising:
- receiving raw data from the raw data source device;
 - identifying key features as two dimensional matrices of the raw data such that the key features are mutually exclusive from each other;
 - translating the key features into key feature vectors;
 - generating a feature vector from at least one of the key feature vectors;
 - receiving a first partial output resulting from an execution of a basic linear algebra subprogram (BLAS) operation using the feature vector and a weight factor; and
 - combining the first partial output with a plurality of partial outputs to produce an output matrix.
- 10.** The apparatus of claim 9, wherein the vectorization unit is configured to perform operations such that identifying key features as two dimensional matrices of the raw data such that the key features are mutually exclusive from each other comprises:
- identifying a first key feature as a first two dimensional matrix of a designated size; and
 - identifying a second key feature as a second two dimensional matrix of the designated size a designated number of units from the first key feature.
- 11.** The apparatus of claim 9, wherein the vectorization unit is configured to perform operations such that generating a feature vector from at least one of the key feature vectors comprises:
- selecting a top key feature vector from a key feature vector queue; and
 - using the top key feature vector as the feature vector.
- 12.** The apparatus of claim 9, wherein the vectorization unit is configured to perform operations such that generating a feature vector from at least one of the key feature vectors comprises:
- selecting a top key feature vector from a key feature vector queue;
 - selecting a next key feature vector from the key feature vector queue;
 - selecting top key feature vector positions and next key feature vector positions; and
 - combining the selected top key feature vector position and the selected next key feature vector positions into the feature vector.
- 13.** The apparatus of claim 12, wherein the vectorization unit is configured to perform operations such that:
- selecting top key feature vector positions and next key feature vector positions comprises selecting the top key feature vector positions and the next key feature vector positions such that each of the selected top key feature vector position and the selected next key feature vector positions represent mutually exclusive locations from each other in the raw data and represent an unidentified key feature of raw data that spans a plurality of the identified key features of the raw data; and
 - combining the selected top key feature vector position and the selected next key feature vector positions into the feature vector comprises combining the selected top key feature vector position and the selected next key feature vector positions into the feature vector such that the feature vector is configured like a key feature vector of the unidentified key feature.
- 14.** The apparatus of claim 9, further comprising a set of vector units communicatively connected to the vectorization unit, wherein the set of vector units is mapped to the input matrix, and wherein:
- the vectorization unit comprises a feature buffer associated with the set of vector units, and the vectorization unit is configured to execute operations further comprising activating the set of vector units upon receiving raw data at the feature buffer associated with the set of vector units;
 - each vector unit of the set of vector units is configured to perform operations comprising:
 - executing the BLAS operation; and
 - outputting at least one partial output.
- 15.** The apparatus of claim 14, wherein the vectorization unit is configured to execute operations further comprising:
- determining whether any feature vectors remain for use in an execution of the BLAS operation by the set of vector units; and
 - deactivating the set of vector units in response to determining that no feature vectors remain for use in an execution of the BLAS operation by the set of vector units.
- 16.** The apparatus of claim 9, wherein the vectorization unit is configured to execute operations such that receiving raw data from a raw data source device comprises receiving streaming raw data from the raw data source device.
- 17.** An apparatus configured to accelerate machine learning on a computing device, comprising:
- means for receiving raw data from a raw data source device;
 - means for identifying key features as two dimensional matrices of the raw data such that the key features are mutually exclusive from each other;
 - means for translating the key features into key feature vectors;
 - means for generating a feature vector from at least one of the key feature vectors;
 - means for receiving a first partial output resulting from an execution of a basic linear algebra subprogram (BLAS) operation using the feature vector and a weight factor; and
 - means for combining the first partial output with a plurality of partial outputs to produce an output matrix.
- 18.** The apparatus of claim 17, wherein means for identifying key features as two dimensional matrices of the raw data such that the key features are mutually exclusive from each other comprises:
- means for identifying a first key feature as a first two dimensional matrix of a designated size; and
 - means for identifying a second key feature as a second two dimensional matrix of the designated size a designated number of units from the first key feature.

19. The apparatus of claim 17, wherein means for generating a feature vector from at least one of the key feature vectors comprises:
- means for selecting a top key feature vector from a key feature vector queue;
 - means for using the top key feature vector as the feature vector.
20. The apparatus of claim 17, wherein means for generating a feature vector from at least one of the key feature vectors comprises:
- means for selecting a top key feature vector from a key feature vector queue;
 - means for selecting a next key feature vector from the key feature vector queue;
 - means for selecting top key feature vector positions and next key feature vector positions; and
 - means for combining the selected top key feature vector position and the selected next key feature vector position into the feature vector.
21. The apparatus of claim 20, wherein:
- means for selecting top key feature vector positions and next key feature vector positions comprises means for selecting the top key feature vector positions and the next key feature vector positions such that each of the selected top key feature vector position and the selected next key feature vector positions represent mutually exclusive locations from each other in the raw data and represent an unidentified key feature of the raw data that spans a plurality of the identified key features of the raw data; and
 - means for combining the selected top key feature vector position and the selected next key feature vector position into the feature vector comprises means for combining the selected top key feature vector position and the selected next key feature vector position into the feature vector such that the feature vector is configured like a key feature vector of the unidentified key feature.
22. The apparatus of claim 17, further comprising:
- means for executing the BLAS operation;
 - means for outputting at least one partial output, wherein means for executing the BLAS operation and means for outputting at least one partial output are mapped to the input matrix;
 - means for activating means for executing the BLAS operation and means for outputting the at least one partial output upon receiving the raw data;
 - means for determining whether any feature vectors remain for use in an execution of the BLAS operation; and
 - means for deactivating means for executing the BLAS operation and means for outputting the at least one partial output in response to determining that no feature vectors remain for use in an execution of the BLAS operation.
23. The apparatus of claim 17, wherein means for receiving raw data from a raw data source device comprises means for receiving streaming raw data from the raw data source device.
24. A non-transitory processor-readable storage medium having stored thereon processor-executable instructions configured to cause a processor of a computing device to perform operations comprising:
- receiving raw data from a raw data source device;
 - identifying key features as two dimensional matrices of the raw data such that the key features are mutually exclusive from each other;
 - translating the key features into key feature vectors;
 - generating a feature vector from at least one of the key feature vectors;
 - receiving a first partial output resulting from an execution of a basic linear algebra subprogram (BLAS) operation using the feature vector and a weight factor; and
 - combining the first partial output with a plurality of partial outputs to produce an output matrix.
25. The non-transitory processor-readable storage medium of claim 24, wherein the stored processor-executable instructions are configured to cause the processor to perform operations such that identifying key features as two dimensional matrices of the raw data such that the key features are mutually exclusive from each other comprises:
- identifying a first key feature as a first two dimensional matrix of a designated size; and
 - identifying a second key feature as a second two dimensional matrix of the designated size a designated number of units from the first key feature.
26. The non-transitory processor-readable storage medium of claim 24, wherein the stored processor-executable instructions are configured to cause the processor to perform operations such that generating a feature vector from at least one of the key feature vectors comprises:
- selecting a top key feature vector from a key feature vector queue; and
 - using the top key feature vector as the feature vector.
27. The non-transitory processor-readable storage medium of claim 24, wherein the stored processor-executable instructions are configured to cause the processor to perform operations such that generating a feature vector from at least one of the key feature vectors comprises:
- selecting a top key feature vector from the key feature vector queue;
 - selecting top key feature vector positions and next key feature vector positions; and
 - combining the selected top key feature vector position and the selected next key feature vector positions into the feature vector.
28. The non-transitory processor-readable storage medium of claim 27, wherein the stored processor-executable instructions are configured to cause the processor to perform operations such that:
- selecting top key feature vector positions and next key feature vector positions comprises selecting the top key feature vector positions and the next key feature vector positions such that each of the selected top key feature vector position and the selected next key feature vector positions represent mutually exclusive locations from each other in the raw data and represent an unidentified key feature of raw data that spans a plurality of the identified key features of the raw data; and
 - combining the selected top key feature vector position and the selected next key feature vector positions into the feature vector comprises combining the selected top key feature vector position and the selected next key feature vector positions into the feature vector such that the feature vector is configured like a key feature vector of the unidentified key feature.

- 29.** The non-transitory processor-readable storage medium of claim 24, wherein the stored processor-executable instructions are configured to cause the processor to perform operations further comprising:
activating the processor upon receiving the raw data,
wherein the processor is mapped to the output matrix;
executing the BLAS operation;
outputting at least one partial output;
determining whether any feature vectors remain for use in an execution of the BLAS operation by the processor;
and
deactivating the processor in response to determining that no feature vectors remain for use in an execution of the BLAS operation by the processor.
- 30.** The non-transitory processor-readable storage medium of claim 24, wherein the stored processor-executable instructions are configured to cause the processor to perform operations such that receiving raw data from a raw data source device comprises receiving streaming raw data from the raw data source device.



(19) United States

(21) Patent Application Publication (20) Pub. No.: US 2017/0286182 A1
(22) Suarez Gracia et al. (23) Pub. Date: Oct. 5, 2017

(54) IDENTIFYING ENHANCED SYNCHRONIZATION OPERATION OUTCOMES TO IMPROVE RUNTIME OPERATIONS

(71) Applicant: QUALCOMM Incorporated, San Diego, CA (U.S.)

(72) Inventors: **Dario Suarez Gracia**, Intel (ES), Gheorghe Cascaval, Palo Alto, CA (U.S); Han Zhao, Santa Clara, CA (U.S); Tushar Kumar, San Jose, CA (U.S); Aravind Narayanan, Sunnyvale, CA (U.S); Arun Roman, Fremont, CA (U.S)

(21) Appl. No.: 150085,108

(22) Filed: Mar. 30, 2016

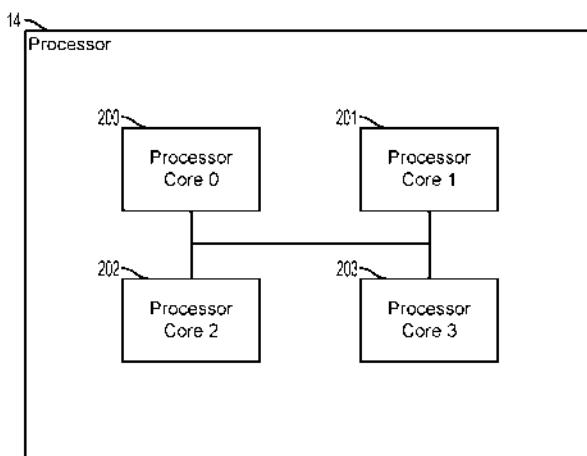
Publication Classification

(51) Int. Cl. G06F 9/52 (2006.01)

(52) U.S. Cl. CPC G06F 9/52 (2013.01)

(57) ABSTRACT

Embodiments include computing devices, systems, and methods identifying enhanced synchronization operation outcomes. A computing device may receive a first resource access request for a first resource of a computing device including a first requester identifier from a first computing element of the computing device. The computing device may also receive a second resource access request for the first resource including a second requester identifier from a second computing element of the computing device. The computing device may grant the first computing element access to the first resource based on the first resource access request, and return a response to the second computing element including the first requester identifier as a winner computing element identifier.



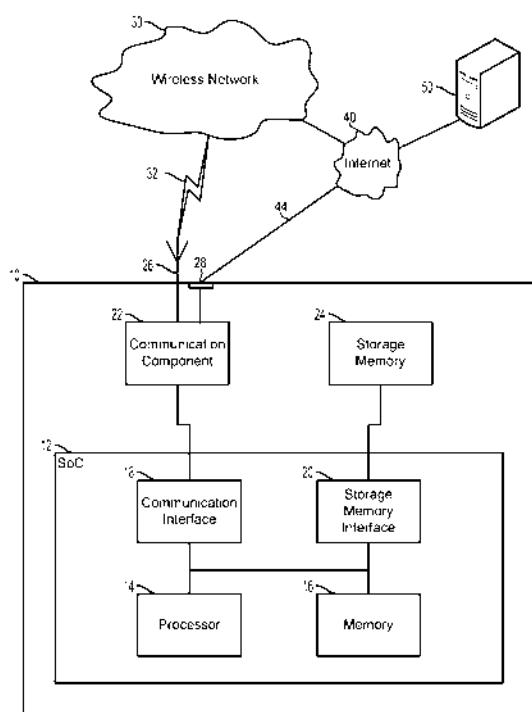


FIG. 1

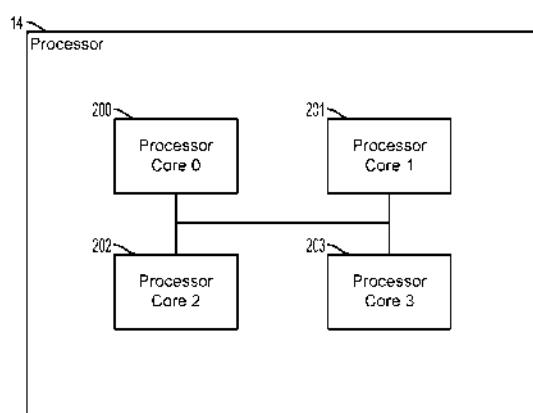


FIG. 2

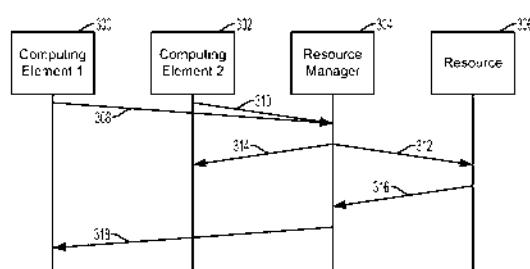


FIG. 3

402	404	406	408	410
Computing Element ID	Shared Resource 1 Computing Element IDs	Shared Resource 2 Computing Element IDs	...	Shared Resource N-1 Computing Element IDs

FIG. 4

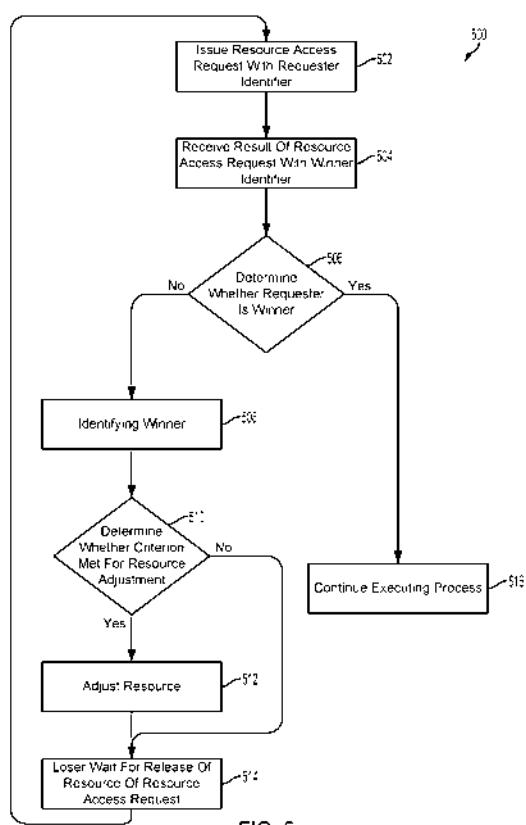


FIG. 5

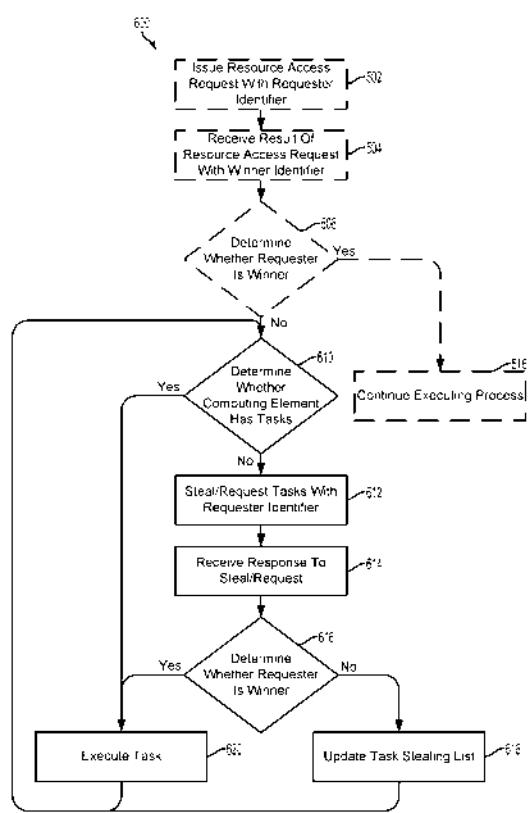


FIG. 6

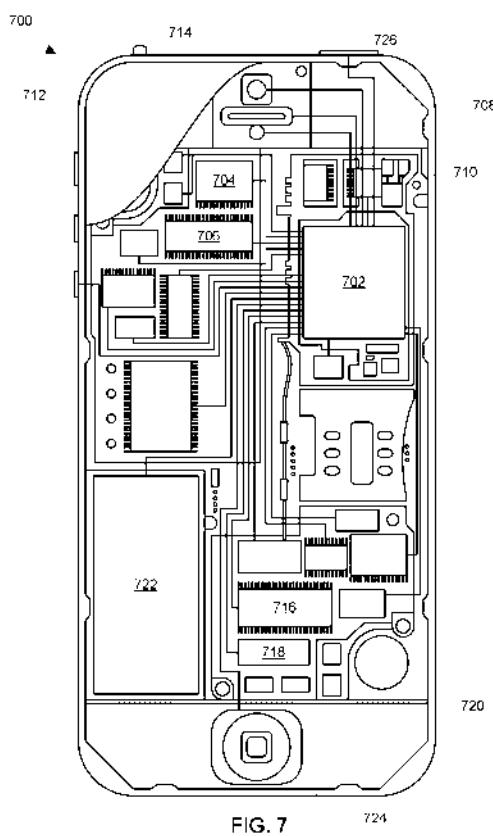


FIG. 7

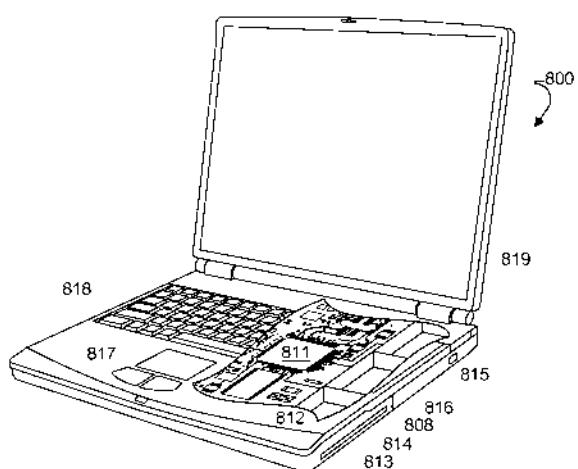


FIG. 8

IDENTIFYING ENHANCED SYNCHRONIZATION OPERATION OUTCOMES TO IMPROVE BUNDLE OPERATIONS

BACKGROUND

[0001] Ensuring correctness in parallel application execution requires hardware atomic synchronization instructions. Such instructions ensure that if multiple processor cores try to concurrently update the same variable, only one processor core will succeed. Some examples of atomic synchronization instructions supported by current hardware include load link/store conditional, compare and swap, fetch-and-increment, etc.

[0002] Synchronization instructions only return binary notifications of success/twin/failure class to the processor cores, consisting of information gap between the hardware and the software. Therefore, a processor core only receives a notification of whether its update was successful. However, arbiters or other resource synchronization and management components on an interconnection network between the processor cores and resources do not share other information related to a successful failed update. Therefore, information is lost between abstractly hardware and the abstraction software architecture.

[0003] Exclusive access to a resource by two or more processor cores that are executing concurrently may be obtained by executing atomic synchronization instructions in order to gain access to said resource. The processor core that executes the synchronization instruction successfully will have obtained exclusive access to the resource.

[0004] Exclusive access to a contended resource may also be granted to a processor core issuing a resource access request for the contended resource on a first come first serve basis. A resource manager can determine whether to grant or deny access to a resource access request issued by any of the processor cores, i.e., requester processor core, based on availability of the contended resource.

SUMMARY

[0005] The methods and apparatuses of various embodiments provide apparatus and methods for identifying enhanced synchronization operation outcomes at a computing device. Various embodiments may include receiving a plurality of resource access requests for a first resource of the computing device from a plurality of computing elements of the computing device granting the first computing element access to the first resource based on the first resource access request, and returning a response to the second computing element. The plurality of resource access requests may include a first resource access request from a first computing element of the plurality of computing elements and a second resource access request from a second computing element of the plurality of computing elements. The first resource access request may include a first requester identifier from the first computing element. The second resource access request may include a second requester identifier from the second computing element. The response may include the first requester identifier as a winner computing element identifier. The computing elements may include physical processor cores, or logical threads as defined herein.

[0006] Some embodiments may further include comparing the second requester identifier to the winner computing element identifier, and determining whether the second computing element is a winner computing element by determining whether the second requester identifier matches the winner computing element identifier.

[0007] Some embodiments may further include identifying the winner computing element from the winner computing element identifier and determining whether a criterion is met for adjusting a second resource of the computing device in response to determining that the second computing element is not the winner computing element. Such embodiments may further include adjusting the second resource by the second computing element in response to determining that the criterion is met for adjusting the second resource.

[0008] In some embodiments, determining whether a criterion is met for adjusting a second resource of the computing device may include determining by the second computing element, a likelihood of sharing the second resource by the first computing element and the second computing element based on one or more criteria. The criteria may include the first computing element and the second computing element having a shared operating system, shared dynamic voltage and frequency scaling, and a shared topology.

[0009] Some embodiments may further include receiving a third resource access request for the first resource, the third resource access request including a third requester identifier from a third computing element of the plurality of computing elements, and returning the response to the third computing element including the first requester identifier as the winner computing element identifier.

[0010] Some embodiments may further include determining whether the second computing element has a task to execute, and sending a signal to steal a task from the first computing element in response to determining that the second computing element does not have a task to execute, in which the signal includes the second requester identifier.

[0011] Some embodiments may further include receiving a response to the attempt to steal a task, the response including a task winner computing element identifier. Such embodiments may further include comparing the second requester identifier to the task winner computing element identifier and determining whether the second computing element is task winner computing element by determining whether the second requester identifier matches the task winner computing element identifier. Such embodiments may further include adjusting a task stealing list of the second computing element in response to determining that the second computing element is not the task winner computing element.

[0012] In some embodiments, adjusting the task stealing list of the second computing element may include rearranging items in the stealing list based at least in part on whether a computing element is executing a recursive task or a non-recursive task.

[0013] Various embodiments may include a computing device configured for identifying enhanced synchronization operation outcomes. The computing device may include a plurality of computing elements, including a first computing element and a second computing element, a first resource, and a resource manager communicatively connected to the plurality of computing elements and the resource, and con-

figured with resource manager executable instructions to perform operations of one or more of the embodiment methods summarized above.

[0014] Various embodiments may include a computing device configured for identifying enhanced synchronization operation outcomes having means for performing functions of one or more of the embodiment methods summarized above.

[0015] Various embodiments may include a non-transitory processor-readable storage medium having stored thereon processor-executable instructions configured to cause a processor of a computing device to perform operations of one or more of the embodiment methods summarized above.

BRUT DESCRIPTION OF THE DRAWINGS

[0016] The accompanying drawings, which are incorporated herein and constitute part of this specification, illustrate example embodiments of various embodiments, and together with the general description given above and the detailed description given below, serve to explain the features of the claims.

[0017] FIG. 1 is a component block diagram illustrating a computing device suitable for implementing an embodiment.

[0018] FIG. 2 is a component block diagram illustrating an example mobile processor suitable for implementing an embodiment.

[0019] FIG. 3 is a process and signaling diagram illustrating hardware support for identifying enhanced synchronization operation outcomes according to an embodiment.

[0020] FIG. 4 is a representational diagram illustrating an identifier register according to an embodiment.

[0021] FIG. 5 is a process flow diagram illustrating an embodiment method for adjusting resources at a physical level based on a winner.

[0022] FIG. 6 is a process flow diagram illustrating an embodiment method for adapting task stealing heuristics at a logical level based on a winner.

[0023] FIG. 7 is component block diagram illustrating an example mobile computing device suitable for use with the various embodiments.

[0024] FIG. 8 is component block diagram illustrating an example mobile computing device suitable for use with the various embodiments.

DETAILED DESCRIPTION

[0025] The various embodiments will be described in detail with reference to the accompanying drawings. Whenever possible, the same reference numbers will be used throughout the drawings to refer to the same or like parts. References made to particular examples and implementations are for illustrative purposes, and are not intended to limit the scope of the claims.

[0026] The terms "computing device" and "mobile computing device" are used interchangeably herein to refer to one of a variety of electronic devices that include a memory, and a programmable processor with any number of processor cores. A processor with more than one processor core may be referred to as a multi-core processor. Examples of computing devices include cellular telephones, smartphones, personal or mobile multimedia players, personal data assistants (PDAs), tablets, 12-in-1 computers, smartbooks, convertible laptops, tablets, 12-in-1 computers, smartbooks,

laptops, notebooks, palm top computers, wireless electronic mail receivers, multimedia internet enabled cellular telephones, mobile gaming consoles, and wireless gaming controllers. The term "computing device" may further refer to stationary computing devices including personal computers, desktop computers, all-in-one computers, workstations, supercomputers, mainframe computers, embedded computers, servers, logic integrated computers, and game consoles. The various embodiments may be particularly useful for mobile computing devices with limited memory and battery resources. However, the embodiments may be generally useful in any electronic device that implements a plurality of memory devices and a limited power budget in which reducing the power consumption of the processors can extend the battery operating time of the electronic device.

[0027] Embodiments may include methods, systems, and devices for sharing more iteration among atomic synchronization operations. The methods/apparatus may include identifying enhanced synchronization operation outcomes to a locking contention, such as by storing the information in a generally accessible register. Embodiments may include sending an identifier of each issuer of resource access requests requester identifier with a synchronization instruction of the request, and returning to other contenders for storing in a register, the identifier of the issuer that is granted access (winner requester identifier) to the contended resource. Embodiments may also include methods for using the winner requester identifier (or winner identifier) to adjust resource configurations at a hardware level, and/or to adjust workload balancing heuristics (e.g., work stealing heuristics) at a software level.

[0028] Computing elements within a computing device, such as physical processors and cores, or logical threads, may issue resource access requests that include a synchronization instruction and requester identifier of a computing element issuing an access request for a contended resource (or requester computing element). A resource manager, such as an arbiter, barrier, or controller, receiving an access request for a contended resource returns a winner identifier to the requester computing elements. The winner identifier identifies the computing element that won the contention and thus has sole ownership of the contended resource. The computing element having ownership of the contended resource may be referred to herein as the owner computing element, winner computing element, owner device, winner device, owner, or winner. The requester computing element that lost the contention to the contended resource may be referred to herein as a non-owner computing element, loser computing element, non-owner device, loser device, non-owner, or loser. The requester identifier may include an identifier for any hardware component or software element requesting access to the contended resource.

[0029] The requested computing element receiving the winner identifier of the winner computing element may determine that access to the contended resource is denied and may adjust hardware resource configurations and/or software resources based on a partitioning to the winner computing element, and shared and/or logically close resources. Adjusting resources may benefit overall performance as the loser computing element may be holding software resources that are needed for the winner computing element to make progress. Thus, a loser computing element is informed of the identity of the winner computing element, and with this information, actions may be taken to transfer

ownership of the resources held by the loser computing element. For example, the loser computing element may determine a likelihood of sharing hardware resources based on whether the winner device is managed by the same operating system, managed within the same dynamic voltage and frequency scaling domain, and/or within physical proximity of the loser computing element. Based on this information, resource configurations may be adjusted, including processing frequency, voltage scaling, number of memory operations, activity states, battery load use, etc.

[0030] How and which resource configurations are adjusted may depend on a level of resource sharing between the winner computing element and the loser computing element, and management policies of a computing device. An implementation may include reducing the frequency of the loser computing element and increasing the winner computing element's frequency to allow the winner computing element to execute faster. Control of the respective processing frequencies of the loser computing element and the winner computing element may be implemented by an operating system (OS) as preprogrammed or at runtime in response to notification by signals of the atomic operation outcome.

[0031] The requester computing element receiving the identifier of the winner computing element may also adjust workload balancing heuristics, such as work stealing heuristics, based on the winner of the contended resource. For example, a work stealing list may include logical elements used to execute an application.

[0032] Adjusting the work balancing heuristics may involve adjusting work stealing heuristics to take into account the behavior of a task. An example of a work stealing heuristic may be one in which a logical element can steal unfinished work items from others upon completion of its original assignment. For a recursive task, which is a task launched by another task, in response to identifying an winner logical element of the contended resource, the winner logical element may be identified as potentially spawning further tasks to execute the iterations of the recursive task. The stealing list can be reordered to indicate that the winner logical element is a first one of the logical elements to check for tasks to steal. For a non-recursive task, in response to identifying a winner logical element of the contended resource, the winner logical element may be identified as having finished all of its assigned tasks for executing the application and stealing tasks from other logical elements, signifying that the winner logical element does not have any tasks for stealing. The stealing list can be modified to remove the winner logical element from the stealing list or reordered to indicate that the winner logical element is the last one of the logical elements to check for tasks to be stolen.

[0033] The identifier of the requester computing element may be stored in a register and provided to the resource manager with the synchronization instruction. State information relating to the hardware component or software component associated with the identifier may also be stored in the register and provided to the resource manager. The resource manager may return the identifier if received for the winner of the contended resource. The resource manager may also track contention information, including a number of failed resource access requests since a successful resource access request for the contended resource and a number of

non-owners of the contended resource. The resource manager may also return the contention information.

[0034] FIG. 1 illustrates a system including a computing device 10 in communication with a remote computing device 50 suitable for use with the various embodiments. The computing device 10 may include a system-on-chip (SoC) 12 with a processor 14, a memory 16, a communication interface 18, and a storage memory interface 20. The computing device 10 may further include a communication component 22 such as a wired or wireless modem, a storage memory 24, an antenna 26 for establishing a wireless connection 32 to a wireless network 30, and/or the network interface 28 for connecting to a wired connection 44 to the Internet 40. The processor 14 may include one or a variety of hardware cores, for example a number of processor cores [0035]. The term "system-on-chip" (SoC) is used herein to refer to a set of interconnected electronic circuits typically, but not exclusively, including a hardware core, a memory, and a communication interface. A hardware core may include a variety of different types of processors, such as a general purpose processor, a central processing unit (CPU), a digital signal processor (DSP), a graphics processing unit (GPU), an accelerated processing unit (APU), an auxiliary processor, a single-core processor, and a multi-core processor. A hardware core may further embody other hardware and hardware combinations such as a field programmable gate array (FPGA), an application-specific integrated circuit (ASIC), other programmable logic device, discrete gate logic, transistor logic, performance monitoring hardware, watchdog hardware and time references. Integrated circuits may be configured such that the components of the integrated circuit reside on a single piece of semiconductor material, such as silicon. In various embodiments, various combinations of the components of the computing device 10 may be separate components not included on the SoC 12.

[0036] The SoC 12 may include one or more processors 14. The computing device 10 may include more than one SoC's 12, thereby increasing the number of processors 14 and processor cores. The computing device 10 may also include processors 14 that are not associated with an SoC 12. Individual processor cores may be multiple processor cores, as described below with reference to FIG. 2. The processors 14 may each be configured for specific purposes that may be the same as or different from other processors 14 on the computing device 10. One or more of the processors 14 and processor cores of the same or different configurations may be grouped together. A group of processors 14 or processor cores may be referred to as a multiprocessor cluster.

[0037] The memory 16 of the SoC 12 may be volatile or non-volatile memory configured for storing data and processor-executable code for access by the processor 14. The computing device 10 and/or SoC 12 may include one or more memories 16 configured for various purposes. In an embodiment, one or more memories 16 may include volatile memories such as random access memory (RAM) or main memory, or cache memory. These memories 16 may be configured to temporarily hold a limited amount of data received from a data sensor or subsystem and data and/or processor-executable code in instructions that are requested from non-volatile memory. These memories 16 may also be configured to temporarily hold data and/or processor executable code instructions loaded to the memories 16 from non-volatile memory in anticipation of future access based on a variety of factors. The memories 16 may also be

configured to temporarily hold intermediary processing data and/or processor-executable code instructions produced by the processor 14 and temporarily stored for future quick access.

[0038] The memory 16 may be configured to store data and processor-executable code, at least temporarily, that is loaded to the memory 16 from another memory device, such as another memory 16 or storage memory 24, in access by one or more of the processors 14. The data or processor-executable code loaded to the memory 16 may be loaded in response to execution of a function by the processor 14. Loading the data or processor-executable code to the memory 16 in response to execution of a function may result from a memory access request to the memory 16 that is unsuccessful, or a miss, because the requested data or processor-executable code is not located in the memory 16. In response to a miss, a memory access request to another memory 16 or storage memory 24 may be made to load the requested data or processor-executable code from the other memory 16 or storage memory 24 to the memory device 16. Loading the data or processor-executable code to the memory 16 in response to execution of a function may result from a memory access request to another memory 16 or storage memory 24, and the data or processor-executable code may be loaded to the memory 16 for later access.

[0039] The communication interface 18, communication component 22, antenna 26, and/or network interface 28 may work in concert to enable the computing device 10 to communicate over a wireless network 30 via a wireless connection 32, and/or a wired network 44 with the remote computing device 50. The wireless network 30 may be implemented using a variety of wireless communication technologies, including, for example, radio frequency spectrum used for wireless communications, to provide the computing device 10 with a connection to the Internet 40 by which it may exchange data with the remote computing device 50.

[0040] The storage memory interface 20 and the storage memory 24 may work in tandem to allow the computing device 10 to store data and processor-executable code onto a non-volatile storage medium. The storage memory 24 may be configured much like an embodiment of the memory 16 in which the storage memory 24 may store the data or processor-executable code for access by one or more of the processors 14. The storage memory 24, being non-volatile, may retain the information even after the power of the computing device 10 has been removed. When the power is re-established and the computing device 10 reboots, the information stored on the storage memory 24 may be available to the computing device 10. The storage memory interface 20 may control access to the storage memory 24 and allow the processor 14 to read data from and write data to the storage memory 24.

[0041] Some or all of the components of the computing device 10 may be differently arranged and/or combined while still serving the necessary functions. Moreover, the computing device 10 may not be limited to one of each of the components, and multiple instances of each component may be included in various configurations of the computing device 10.

[0042] FIG. 2 illustrates a multi-core processor 14 suitable for implementing an embodiment. The multi-core processor 14 may have a plurality of homogeneous or heterogeneous processor cores 200, 201, 202, 203. The processor cores 200,

201, 202, 203 may be homogeneous in implementations in which the cores of a single processor 14 are configured for the same purpose and have the same or similar performance characteristics. For example, the processor 14 may be a general purpose processor, and the processor cores 200, 201, 202, 203 may be homogeneous general purpose processor cores. Alternatively, the processor 14 may be a graphics processing unit or a digital signal processor, and the processor cores 200, 201, 202, 203 may be homogeneous graphics processor cores or digital signal processor cores, respectively. For ease of reference, the terms "processor" and "processor core" may be used interchangeably herein. [0043] The processor cores 200, 201, 202, 203 may be heterogeneous in that the processor cores 200, 201, 202, 203 of a single processor 14 may be configured for different purposes and/or have different performance characteristics. The heterogeneity of such heterogeneous processor cores may include different instruction set architecture, pipelines, operating frequencies, etc. An example of such heterogeneous processor cores may include what are known as "big.LITTLE" architectures in which slower, low-power processor cores may be coupled with more powerful and power-hungry processor cores. In some embodiments, the SoC 12 may include a number of homogeneous or heterogeneous processors 14.

[0044] In the example illustrated in FIG. 2, the multi-core processor 14 includes four processor cores 200, 201, 202, 203 (i.e., processor core 0, processor core 1, processor core 2, and processor core 3). For ease of explanation, the examples herein may refer to the four processor cores 200, 201, 202, 203 illustrated in FIG. 2. However, the four processor cores 200, 201, 202, 203 illustrated in FIG. 2 and described herein are merely provided as an example and in no way are meant to limit the various embodiments to a four-core processor system. The computing device 10, the SoC 12, or the multi-core processor 14 may individually or in combination include fewer or more than the four processor cores 200, 201, 202, 203 illustrated and described herein.

[0045] FIG. 3 illustrates a process and signaling for identifying enhanced synchronization operation outcomes according to an embodiment. The example illustrated in FIG. 3 is non-limiting, particularly with respect to the number and types of components implementing the process and signaling, and the number and order of the signals illustrated. This example includes computing elements (computing element 1300 and computing element 2302), a resource manager 304, and a resource 306. In various implementations, the computing elements 300, 302 may include any of the semi-conductor-based hardware implementations, such as processor 14, processor cores 200, 201, 202, 203, and other hardware cores, for example as described herein, and logical implementations, such as threads and processes. The resource manager 304 may also include hardware implementations, such as an offload, a buffer, a controller, a management unit, and an interface device. The resources 306 may include any hardware component or software elements accessible and usable by the computing elements 300, 302 to execute a task, such as a memory or storage location, an input-output port of various components, or a communication channel.

[0046] In the example illustrated in FIG. 3, both the computing element 1300 and the computing element 2302 may issue resource access requests to the same resource 306 or multiple resources. As a non-limiting example and for

case of explanation, the descriptions herein may refer to the computing element 1 300 and the computing element 2 302 each issuing a single resource access request 308, 310 for a single resource 306. The computing element 1 300 may issue a resource access request 308, and the computing element 2 302 may issue a resource access request 310. The resource access request 310 may include a targeted resource 306, designated for example by a virtual or physical address, an operation, and a requester identifier. A synchronization operation may be an operation that requires the requester computing element 300, 302 to have exclusive access to the resource 306 or at least exclusive access to modify the resource 306. Without exclusive access to or exclusive access to modify the resource 306, the operation may encounter errors when an unexpected value is retrieved from the resource 306 after modification of the resource 306 by the other of the computing elements 300, 302. The requester identifier may include a value uniquely identifying the computing element 300, 302 issuing the request to access the resource 306. The requester identifier may be stored in a component, such as a register, cache, or buffer associated with the computing element 300, 302, and may be retrieved from the component for inclusion in the resource access request 308, 310, respectively.

[0047] The resource manager 304 may receive the resource access requests 308, 310 and determine whether to allow access to the resource 306 by either or both of the computing elements 300, 302. In some implementations, the resource 306 may become contested and the resource manager 304 may deny access to the resource 306 by one of the computing elements 300, 302. The resource 306 may become a contested resource because multiple computing elements 300, 302 are concurrently accessing or attempting to access the resource 306.

[0048] The contention for the resource 306 may stem from the resource manager 304 denying access to one of the computing element 1 300 and the computing element 2 302 while the other accesses the resource 306. Not all contention attempts to access the resource 306 may be contentious. However, contention may occur when multiple computing elements 300, 302 attempt to access the resource 306 to modify the resource 306. Contention may also occur when one of the computing element's access of the resource 306 relies on a consistent state of the resource 306 while the other computing element 300, 302 modifies the state of the resource 306. Contention may also occur when access to the resource 306 by one of the computing elements 300, 302 is dependent on previous access to the resource 306 by the other of the computing elements 300, 302.

[0049] Regardless of the reason for contention of the resource 306, the resource manager 304 may allow access to the resource 306 by one of the computing elements 300, 302 and deny access to the resource 306 by the other computing elements 300, 302. Thus, the resource 306 may allow implementation of one of the resource access requests 308, 310 and prohibit the implementation of the other of the resource access requests 308, 310.

[0050] For the allowed one of the resource access requests 308, 310, the resource manager 304 may permit implementation of an operation 312 on the resource 306. As discussed above, the operation 312 may include an operation that may modify the resource 306 or may rely on a consistent state of the resource 306 during the operation 312.

[0051] The resource manager 304 may store the requester identifier from the permitted resource access requests 308, 310. The resource manager 304 may store the requester identifier as a winner identifier as differentiated from a loser identifier corresponding with the requester identifier of the prohibited resource access requests 308, 310. In some implementations, the winner identifier may be stored in a location accessible to the computing elements 300, 302, such as a register, so that the computing elements may check the winner identifier for use in utilizing resources as discussed further herein.

[0052] The winner identifier may be correlated with the resource 306 to allow for tracking the ownership of the resource 306, so that the resource manager 304 and other computing elements requesting access to the resource 306 may be informed that the resource 306 is owned and by which computing element. For example, the resource manager 304 may use the stored winner identifier and its correlation to the resource 306 to make further determination of whether to allow or prohibit access to other concurrent resource access requests.

[0053] In the example illustrated in FIG. 3, the resource manager 304 permits the resource access requests 308 issued by the computing element 1 300. As a result, the computing element 1 300 is the winner of the contention for the resource 306, and the requested identifier of the computing element 1 300 and the resource access requests 308 may be designated as the winner identifier.

[0054] For the prohibited one of the resource access requests 308, 310, the resource manager 304 may return a response 314 to the computing element 300, 302 having issued the prohibited one of the resource access requests 308, 310. The response 314 may indicate to the receiving computing element 300, 302 that its respective resource access request 308, 310 is denied. The response 314 may indicate denial of the resource access request 308, 310 by including a signal, such as a designated bit, that may indicate the denial by having a designated value. The response 314 may also alternatively include the winner identifier.

[0055] The winner identifier may be used as the signal indicating the denial of the prohibited resource access request 308, 310. In this regard, the receiving computing element 300, 302 may compare the winner identifier to its own requester identifier and determine that the resource access request 308, 310 is denied in response to the winner identifier different from its own requester identifier.

[0056] The resource manager 304 may include the signal indicating the denial of the prohibited resource access request 308, 310 and/or the winner identifier in the response 314. In the example illustrated in FIG. 3, the resource manager 304 prohibits the resource access request 310 issued by the computing element 2 302. As a result, the computing element 2 302 is the loser of the contention for the resource 306. The resource manager 304 sends the response 314, including the winner identifier (i.e., the requester identifier of the computing element 1 300) to the computing element 2 302. The computing element 2 302 may determine that it is the loser of the contention for the resource 306 and may wait for the resource 306 to become available, continue executing tasks that can be executed without access to the resource 306, and/or adjust physical and/or logical resources as described further herein.

[0057] In response to the prohibited resource access requests 308, 310, a response 316 may be generated either

to notify the requester computing element 300, 302 of the permitted resource access requests 308, 310 of completion of the requested access to the resource 306 or to provide data from the resource 306. The resource manager 304 may receive the response 316, note whether the requested access to the resource 306 is complete, and may direct the response 316 to the requester computing element 300, 302 of the permitted resource access requests 308, 310 as a response 318. In some implementations, the resource manager 304 may relinquish the resource 306 upon completion of the requested access to the resource 306. In doing so, the resource manager 304 may remove or invalidate the stored winner identifier and its correlation to the resource 306 [0058]. In some implementations, the resource access requests 308, 310 may further include state information for the respective requester computing elements 300, 302, 126; state information may include processing frequency, voltage scaling, number of memory operations, activity states, bandwidth use, temperature, current leakage, etc.

[0059] The resource manager 304 may store and correlate the state information of the requester computing elements 300, 302 from the permitted resource access requests 308, 310 as part of the response 314. The loser computing elements 300, 302 may use the state information of the winning computing elements 300, 302 in adjusting physical and/or logical resources as described further herein.

[0060] The resource manager 304 may also track contention information for the contested resource 306, including a number of failed resource access requests since a successful resource access request for the contested resource 306 and a number loser computing elements 300, 302 of the contested resource 306. The resource manager 304 may store and correlate the contention information for the contested resource 306 with the winner identifier. The resource manager 304 may include the contention information for the contested resource 306 as part of the response 314. The loser computing elements 300, 302 may use the contention information for the contested resource 306 in adjusting physical and/or logical resources as described further herein.

[0061] FIG. 4 illustrates an identifier register 400 according to an embodiment. Each computing element 300, 302 may include as a component or be associated with an identifier register 400. The identifier register 400 may include a location for storing the computing element identifier (ID) 402, which may be accessed for retrieving the computing element identifier for use as the requesting identifier in a resource access request.

[0062] The identifier register 400 may also include locations associated with shared resources 404-412. The shared resources may be any resource shared by the computing element 300, 302 associated with the identifier register 400 and other computing elements 300, 302 for executing tasks, as opposed to being exclusively accessed by computing element 300, 302 associated with the identifier register 400.

[0063] The locations associated with shared resources 404-412 may each be dedicated to a shared resource and store computing element identifiers for the computing elements 300, 302 that share the shared resource. For example, the identifier register 400 may include a location 404 for storing computing element identifiers that share shared resource 1, a location 406 for storing computing element

identifiers that share shared resource 2, a location 408 for storing computing element identifiers that share shared resource N-1, and a location 412 for storing computing element identifiers that share shared resource N. The identifier register 400 may include any number of locations 404-412 for storing computing element identifiers for at least up to "N" number of shared resources.

[0064] The identifier register 400 associated with a loser computing element 300, 302 and a winner computing element 300, 302 may be accessed by the loser computing element 300, 302 to identify resources 306 shared between the winner and loser computing elements 300, 302. The resources 306 shared between the winner and loser computing elements 300, 302 may be adjusted to improve the execution of a local portion of a process by the winner computing element 300, as described further herein.

[0065] FIG. 5 illustrates a method 500 for adjusting resources 306 at a physical level based on a winner according to various embodiments. The method 500 may be executed in a computing device using software executing on general purpose hardware, such as the processor, and/or dedicated hardware implementing the computing elements and/or the resource manager.

[0066] In block 502, the computing device may issue a resource access request including a requester identifier. As discussed before, the requester identifier may be the computing element identifier of the computing element issuing the resource access request. Further, the resource access request may include a synchronization operation, and/or a physical or virtual address of the target resource of the resource access request. In some implementations, the resource access request may also include state information of the requester computing element, such as a processing frequency, a voltage scaling, a number of memory operations, activity states, a bandwidth use, a temperature, a current leakage, etc.

[0067] In block 504, the computing device may receive a result of the resource access request in a response including a winner identifier indicating the computing element granted access to the target resource by the resource manager in a resource contention. In some implementations, the response may include some or all of the state information of the winner computing element provided in the winner resource access request.

[0068] In determination block 506, the computing device may determine whether the requester computing element is the winner computing element. The computing device may retrieve the computing element identifier for the requester computing element from its associated identifier register, and compare the computing element identifier to the winner identifier. A comparison resulting in a match between the computing element identifier and the winner identifier may indicate that the requester computing element is the winner computing element. A comparison resulting in a mismatch between the computing element identifier and the winner identifier may indicate that the requester computing element is the loser computing element.

[0069] In response to determining that the requester computing element is the winner computing element (i.e., determination block 506 = "Yes"), the computing device may continue to execute a process being executed by the winner computing element in block 516. The winner computing element may be provided access to resources, such as the contested resource, necessary for executing a process. The

winner computing element may leverage the access to the contested resource in order complete a portion of an operation requiring use of the resource.

[0070] The winner computing element may continue to maintain ownership of the contested resource until the contested resource is no longer needed by the winner computing element to execute the operation, upon which the winner computing element may relinquish ownership of the contested resource. In some implementations, the winner computing element may be forced to relinquish ownership of the contested resource based on various factors, including time, number of loser computing elements for the contested resource, number of detailed access requests for the contested resource, use of the contested resource, etc., to avoid degradation or the performance of the computing device. In relinquishing ownership of the contested resource, the winner computing element may send a notification signal to the resource manager, the loser computing element, other computing elements, and/or register accessible by multiple computing elements that the contested resource is available.

[0071] In response to determining that the requester computing element is not the winner computing element (i.e., determination block 500 “No”), the computing device may identify the winner computing element. The computing device may identify the winner computing element as the computing element associated with the winner identifier included with its response to the resource access request received in block 504. The computing device may use the winner identifier to determine a relationship between the winner computing element and the loser computing element for adjusting physical and/or logical resources as described herein.

[0072] The computing device may determine whether a criterion is met for adjusting physical and/or logical resources of the computing device in determination block 510. In computing devices with multiple operating systems, dynamic voltage and frequency scaling domain, and topologies, the loser computing element may tune local and/or shared resources. The resources for tuning may be shared between the loser computing element and the winner computing element. In making this determination, the computing device may determine whether a criterion is met for adjusting physical and/or logical resources of the computing device when any of the conditions described herein are met and adjusting the resources is likely to improve the performance of the computing device. In various implementations, the criterion may depend, at least in part, on a relationship between the loser computing element and the winner computing element. The loser computing element may use the information of the winner computing device identified in block 508 to make the determination in determination block 510.

[0073] In response to determining that a criterion is not met for adjusting physical and/or logical resources of the computing device (i.e., determination block 510 “No”), the computing device may adjust physical and/or logical resources of the computing device in block 512. Adjusting the physical and/or logical resources of the computing device may be implemented as described herein and in any manner that may benefit the performance of the computing

device. For example, the loser computing element may adjust hardware resource configurations and/or software resources based on a relationship to the winner computing element, shared resources, and/or topologically close resources.

[0074] Adjusting resources in block 512 may benefit overall performance of the computing device, as the loser computing element may be holding resources that are needed for the winner computing element to make progress in executing the process. For example, the loser computing element may determine a likelihood of sharing hardware resources based on whether the winner computing element is managed by the same operating system, managed within the same dynamic voltage and frequency scaling domain, and is in close physical proximity of the winner computing element. Based on this information, resource configurations may be adjusted, including processing frequency, voltage scaling, number of memory operations, activity states, bandwidth use in flight misses, etc.

[0075] How and which resource configurations are adjusted in block 512 may depend on a level of resource sharing between the winner computing element and the loser computing element, and management policies of a computing device. Some implementations may include the loser computing element reducing its processing frequency and increasing the winner computing element's frequency to reduce the time needed to execute a critical section of an application including the atomic operations of the winner computing element using the contested resource. In another example, the loser computing element may adjust its cache bandwidth use and/or in-flight misses to reduce the number of outstanding miss requests for a period of time, thereby also reducing the number of slower storage lookups necessary and allowing greater resources to be used by the winner computing element.

[0076] Following or in parallel with adjusting resources, or in response to determining that no criterion is met for adjusting physical and/or logical resources of the computing device (i.e., determination block 510 “No”), the computing device may wait for the winner computing element to release ownership of the contested resource in block 514. As discussed herein, the loser computing element may be notified of the release ownership of the contested resource by the winner computing element in multiple ways. In some implementations, the loser computing element may receive a signal from the winner computing element or the resource manager indicating the release ownership of the contested resource by the winner computing element. In some implementations, the loser computing element may check an accessible register for an indication of the release of ownership of the contested resource by the winner computing element. Upon being notified of the release of ownership of the contested resource by the winner computing element, the computing device may issue a resource access request including a requester identifier in block 502.

[0077] FIG. 6 illustrates an embodiment method 600 for adjusting task-stealing heuristics at a logical level based on a winner according to various embodiments. The method 600 may be executed in a computing device using software executing on general purpose hardware, such as the processor, and/or dedicated hardware implementing the computing elements and/or the resource manager.

[0078] Blocks 502-506 and 516 may be implemented in a manner similar to that of like numbered blocks in method

500 is described with reference to FIG. 5. In some implementations of the method **600**, blocks **502-506** and **516** may be optionally implemented.

[0079] In response to determining that the requestor computing element is not the winner computing element or is the loser computing element (i.e., determination block **506** “No”), the computing device may determine whether the loser computing element has a task that is executable without access to the contested resource in determination block **610**.

[0080] In response to determining that the loser computing element does not have a task that is executable without access to the contested resource (i.e., determination block **610** “No”), the computing device may attempt to steal or request tasks in block **612**. If the resource access request signal to steal or request tasks may include the computing element identifier (requester identifier) of the computing element sending the signal to steal or request work, i.e., the loser computing element. The loser computing element may send a general signal to a resource manager implementing a scheduler, or check competing elements in its stealing list for competing elements likely to have available tasks and signal specifying the competing elements. In some implementations, the stealing list may contain competing elements executing the same application as the loser computing element. In the resource access request, the resource manager may determine a winner and a loser from multiple competing elements attempting to steal or request tasks, and return to or make available to the computing elements the winner identifier.

[0081] In block **614**, the computing device may receive a response to the signal to steal or request tasks. The response may include the winner identifier, and/or winner computing elements, a task assignment, to execute.

[0082] In determination block **616**, the computing device may determine whether the loser computing element is a winner computing element for the task stealing or request. If determining whether the computing element is a winner computing element in block **506**, the computing device may compare the winner identifier with the computing element identifier of the owner of the signal to steal or request tasks in determination block **616**. A comparison resulting in a match between the computing element identifier and the winner identifier may indicate that the requestor computing element is the task winner computing element. A comparison resulting in a mismatch between the computing element identifier and the winner identifier may indicate that the requestor computing element is the task loser computing element.

[0083] In response to determining that the loser computing element is the task winner computing element (i.e., determination block **616** “Yes”), the computing device may execute the stolen or received tasks.

[0084] In response to determining that the loser computing element is not the task winner computing element or is the task loser computing element (i.e., determination block **616** “No”), the computing device may update or adjust the task stealing list of the task loser computing element in block **618**.

[0085] Adjusting the stealing list in block **618** may take into account behavior of the application. For a task winner computing element executing a non-recursive task, the task winner computing element may be identified as having finished all of its assigned tasks for executing the application

and stealing tasks from other competing elements, signifying that the task winner computing element does not have any tasks for stealing. For example, in an application with non-recursive tasks, once a competing element finishes initially assigned tasks, the competing element will commence to look for other tasks by stealing or requesting other tasks. Thus, if a computing element is contending for tasks, then it has finished its initially assigned tasks and does not have any tasks to steal or give. The stealing list can be modified to remove the task winner computing element from the stealing list or accelerated to indicate that the task winner computing element is the last one of the competing elements to check for tasks.

[0086] For a task winner computing element executing a recursive task, the task winner computing element may be identified as potentially spawning further tasks to execute the iterations of the recursive task. For example, in an application with recursive tasks, once a competing element finishes initially assigned tasks, the competing element will commence to look for other tasks by stealing or requesting other tasks. Thus, if a computing element is contending for tasks, then it has finished its initially assigned tasks but may generate more tasks to steal or give it assigned further recursive tasks. The stealing list can be reordered to indicate that the task winner computing element is the first one of the competing elements to check for tasks.

[0087] The various embodiments including, but not limited to, embodiments discussed above with reference to FIGS. 1-6 may be implemented in a wide variety of computing systems, which may include an example mobile computing device suitable for use with the various embodiments illustrated in FIG. 7. The mobile computing device **700** may include a processor **702** coupled to a touchscreen controller **704** and an internal memory **706**. The processor **702** may be one or more multi-core integrated circuits designated for general or specific processing tasks. The internal memory **706** may be volatile or non-volatile memory, and may also be secure and/or encrypted memory, or unsecure and/or unencrypted memory, or any combination thereof. Examples of memory types that can be leveraged include but are not limited to DDR, LPDDR, GDDR, WiDDR, R-RAM, SRAM, DRAM, PARAM, R-SRAM, MR-RAM, ST-RAM, and embedded dynamic random access memory (eDRAM). The touchscreen controller **704** and the processor **702** may also be coupled to a touchsensor panel **712**, such as a resistive-sensing touchscreen, capacitive-sensing touchscreen, infrared sensing touchscreen, etc. Additionally, the display of the computing device **700** need not have touch screen capability.

[0088] The mobile computing device **700** may have one or more radio signal transceivers **708** (e.g., Picard, Bluetooth, Zigbee, Wi-Fi, BL radio and antenna **710**, for sending and receiving communications, coupled to each other and to the processor **702**). The transceivers **708** and antenna **710** may be used with the above-mentioned circuitry to implement the various wireless transmission protocol stacks and interfaces. The mobile computing device **700** may include a cellular network wireless modem chip **716** that enables communication via a cellular network and is coupled to the processor.

[0089] The mobile computing device **700** may include a peripheral device connection interface **718** coupled to the processor **702**. The peripheral device connection interface **718** may be singularly configured to accept one type of

connection, or may be configured to accept various types of physical and communication connectors, common or proprietary, such as USB, FireWire, Thunderbolt, or PCIe. The peripheral device connection interface 718 may also be coupled to a similarly configured peripheral device connector port (not shown).

[0090] The mobile computing device 700 may also include speakers 714 for providing audio outputs. The mobile computing device 700 may also include a housing 720, constructed of a plastic, metal, or a combination of materials, for containing all or some of the components discussed herein. The mobile computing device 700 may include a power source 722 coupled to the processor 702, such as a disposable or rechargeable battery. The rechargeable battery may also be coupled to the peripheral device connection port to receive a charging current from a source external to the mobile computing device 700. The mobile computing device 700 may also include a physical, built-in 724 for receiving user inputs. The mobile computing device 700 may also include a power button 726 for turning the mobile computing device 700 on and off.

[0091] The various embodiments, including, but not limited to, embodiments discussed above with reference to FIGS. 1-6, may be implemented in a wide variety of computing systems, which may include a variety of mobile computing devices, such as a laptop computer 800 illustrated in FIG. 8. Many laptop computers include a touch surface 817 that serves as the computer's pointing device, and thus may receive drag, scroll, and click gestures similar to those implemented on computing devices equipped with a touch screen display and described above. A laptop computer 800 will typically include a processor 811 coupled to volatile memory 812, and a large capacity, non-volatile memory, such as a disk drive 813, or eFlash memory. Additionally, the computer 800 may have one or more antenna 808 for sending and receiving electromagnetic radiation that may be connected to a wireless data link, and a cellular telephone transceiver 816 coupled to the processor 811. The computer 800 may also include a floppy disc drive 814 and a compact disc (CD) drive 815 coupled to the processor 811. In a notebook configuration, the computer housing includes the touchpad 817, the keyboard 818, and the display 819 all coupled to the processor 811. Other configurations of the computing device may include a computer mouse or trackball coupled to the processor via, e.g., a universal serial bus (USB) input, as is well known, which may also be used in conjunction with the various embodiments.

[0092] Computer program code, or "program code," for execution on a programmable processor for carrying out operations of the various embodiments may be written in a high level programming language such as C, C++, C#, Smalltalk, Java, JavaScript, Visual Basic, a Structured Query Language (e.g., Transact-SQL), Perl, or in various other programming languages. Program code or programs stored on a computer readable storage medium as used in this application may refer to machine language code, such as object code whose format is understandable by a processor.

[0093] The foregoing method descriptions and the process flow diagrams are provided merely as illustrative examples and are not intended to require or imply that the operations of the various embodiments must be performed in the order presented. As will be appreciated by one skilled in the art, the order of operations in the foregoing embodiments may be

performed in any order. Words such as "herein," "therein," etc. are not intended to limit the order of the operations; these words are simply used to guide the reader through the description of the methods. Further, any reference to claim elements in the singular, for example, using the articles "a," "an," or "the," is not to be construed as limiting the element to the singular.

[0094] The various illustrative logical blocks, modules, circuits, and algorithm operations described in connection with the various embodiments may be implemented as electronic hardware, computer software, or combinations of both. To clearly illustrate this interchangeability of hardware and software, various illustrative components, blocks, modules, etc., circuit, and operations have been described above generally in terms of their functionality. Whether such functionality is implemented as hardware or software depends upon the particular application and design constraints imposed on the overall system. Skilled artisans may implement the described functionality in varying ways for each particular application, but such implementation decisions should not be interpreted as causing a departure from the scope of the claims.

[0095] The logic blocks used to implement the various illustrative logics, logical blocks, modules, and circuits described in connection with the embodiments disclosed herein may be implemented, or performed with, a general purpose processor, a digital signal processor (DSP), an application specific integrated circuit (ASIC), a field programmable gate array (FPGA) or other programmable logic device, discrete gate or transistor logic, discrete hardware components, or any combination thereof, designed to perform the functions described herein. A general-purpose processor may be a microprocessor, but, in the alternative, the processor may be any conventional processor, controller, microcontroller, or state machine. A processor may also be implemented as a combination of computing devices, e.g., a combination of a DSP and a microprocessor, a plurality of microprocessors, one or more microprocessors in conjunction with a DSP core, or any other such configuration. Alternatively, some operations or methods may be performed by circuitry that is specific to a given function.

[0096] In one or more embodiments, the functions described may be implemented in hardware, software, firmware, or any combination thereof. If implemented in software, the functions may be stored as one or more instructions in a code on a non-transitory computer-readable medium or a non-transitory processor-readable medium. The operations of a method or algorithm disclosed herein may be embodied in a processor-executable software module that may reside on a non-transitory computer-readable or processor-readable storage medium. Non-transitory computer-readable or processor-readable storage media may be any storage media that may be accessed by a computer or a processor. By way of example but not limitation, such non-transitory computer-readable or processor-readable media may include RAM, ROM, EPROM, EEPROM, FLASH memory, CD-ROM, or other optical disk storage, magnetic disk storage or other magnetic storage devices, or any other medium that may be used to store desired program code in the form of instructions or data structures and that may be accessed by a computer. Disk and disc, as used herein, includes compact disc (CD), laser disc, optical disc, digital versatile disc (DVD), floppy disk, and Blu-ray disc, where disk usually reproduce data magnetically, while discs

reproduce data optically with lasers. Combinations of the above are also included within the scope of non-transitory computer-readable and processor-readable media. Additionally, the operations of a method or algorithm may reside as states of any combination of selected codes and/or instructions in a non-transitory processor-readable medium and/or computer-readable medium, which may be incorporated into a computer program product.

[0097] The preceding description of the disclosed embodiments is provided to enable any person skilled in the art to make or use the claims. Various modifications to these embodiments will be readily apparent to those skilled in the art, and the generic principles defined herein may be applied to other embodiments without departing from the scope of the claims. Thus, the present disclosure is not intended to be limited to the embodiments shown herein but is to be accorded the widest scope consistent with the following claims and the principles and novel features disclosed herein.

What is claimed is:

1. A method of identifying enhanced synchronization operation outcomes in a computing device, comprising:
receiving a plurality of resource access requests for a first resource of the computing device from a plurality of computing elements of the computing device including a first resource access request having a first requester identifier from a first computing element of the plurality of computing elements and a second resource access request having a second requester identifier from a second computing element of the plurality of computing elements;
granting the first computing element access to the first resource based on the first resource access request; and
returning a response to the second computing element including the first requester identifier as a winner computing element identifier;
2. The method of claim 1, further comprising:
comparing the second requester identifier to the winner computing element identifier; and
determining whether the second computing element is a winner computing element by the second requester identifier matching the winner computing element identifier;
3. The method of claim 2, further comprising:
identifying the winner computing element from the winner computing element identifier;
determining whether a criterion is met for adjusting a second resource of the computing device in response to determining that the second computing element is not the winner computing element; and
adjusting the second resource by the second computing element in response to determining that the criterion is met for adjusting the second resource;
4. The method of claim 3, wherein determining whether a criterion is met for adjusting a second resource of the computing device comprises, determining, by the second computing element, a likelihood of sharing the second resource by the first computing element and the second computing element based on one or more of a shared operating system, shared dynamic voltage and frequency scaling, and a shared topology;
5. The method of claim 1, further comprising:
receiving a third resource access request for the first resource including a third requester identifier from a third computing element of the plurality of computing elements; and
returning the response to the third computing element including the first requester identifier as the winner computing element identifier;
6. The method of claim 1, further comprising:
determining whether the second computing element has a task to execute; and
sending a signal to steal a task from the first computing element in response to determining that the second computing element does not have a task to execute, wherein the signal includes the second requester identifier;
7. The method of claim 6, further comprising:
receiving a response to the signal to steal a task including a task winner computing element identifier;
comparing the second requester identifier to the task winner computing element identifier;
determining whether the second computing element is a task winner computing element by the second requester identifier matching the task winner computing element identifier; and
adjusting a task stealing list of the second computing element in response to determining that the second computing element is not the task winner computing element;
8. The method of claim 7, wherein adjusting the task stealing list of the second computing element comprises:
removing from the stealing list based at least in part on whether a computing element is executing a recursive task or a non-recursive task;
9. A computing device configured for identifying enhanced synchronization operation outcomes, comprising:
a plurality of computing elements, including a first computing element and a second computing element;
a first resource; and
a resource manager communicatively connected to the plurality of computing elements and the resource, wherein the resource manager is configured with executable instructions to perform operations comprising:
receiving a plurality of resource access requests for the first resource including a first resource access request having a first requester identifier from the first computing element and a second resource access request having a second requester identifier from the second computing element;
granting the first computing element access to the first resource based on the first resource access request; and
returning a response to the second computing element including the first requester identifier as a winner computing element identifier;
10. The computing device of claim 9, wherein the second computing element is configured with executable instructions to perform operations comprising:
comparing the second requester identifier to the winner computing element identifier; and

- determining whether the second computing element is a winner computing element by the second requester identifier matching the winner computing element identifier;
11. The computing device of claim 10, further comprising a second resource communicatively connected to the second computing element, wherein the second computing element is configured with executable instructions to perform operations further comprising:
- identifying the winner computing element from the winner computing element identifier;
 - determining whether a criterion is met for adjusting the second resource in response to determining that the second computing element is not the winner computing element; and
 - adjusting the second resource in response to determining that the criterion is met, for adjusting the second resource;
12. The computing device of claim 11, wherein the second computing element is configured with executable instructions to perform operations such that determining whether a criterion is met for adjusting the second resource comprises determining a likelihood of sharing the second resource by the first computing element and the second computing element based on one or more of a shared operating system, shared dynamic voltage and frequency scaling, and a shared topology;
13. The computing device of claim 9, wherein the plurality of computing elements further comprises a third computing element, and wherein the resource manager is configured with executable instructions to perform operations further comprising:
- receiving a third resource access request for the first resource including a third requester identifier from the third computing element; and
 - returning the response to the third computing element including the first requester identifier as the winner computing element identifier;
14. The computing device of claim 9, wherein the second computing element is configured with executable instructions to perform operations comprising:
- determining whether the second computing element has a task to execute; and
 - sending a signal to steal a task from the first computing element in response to determining that the second computing element does not have a task to execute, wherein the signal includes the second requester identifier;
15. The computing device of claim 14, wherein the second computing element is configured with executable instructions to perform operations further comprising:
- receiving a response to the signal to steal a task including a task winner computing element identifier;
 - comparing the second requester identifier to the task winner computing element identifier;
 - determining whether the second computing element is a task winner computing element by the second requester identifier matching the task winner computing element identifier; and
 - adjusting a task stealing list of the second computing element in response to determining that the second computing element is not the task winner computing element;
16. The computing device of claim 15, wherein the second computing element is configured with computing element-executable instructions to perform operations such that adjusting the task stealing list of the second computing element comprises rearranging items in the stealing list based at least in part on whether a computing element is executing a recursive task or a non-recursive task;
17. A computing device configured for identifying enhanced synchronization operation outcomes, comprising:
- means for receiving a plurality of resource access requests for a first resource of the computing device from a plurality of computing elements of the computing device including a first resource access request having a first requester identifier from a first computing element of the plurality of computing elements and a second resource access request having a second requester identifier from a second computing element of the plurality of computing elements;
 - means for granting the first computing element access to the first resource based on the first resource access request; and
 - means for returning a response to the second computing element including the first requester identifier as a winner computing element identifier;
18. The computing device of claim 17, further comprising:
- means for comparing the second requester identifier to the winner computing element identifier; and
 - means for determining whether the second computing element is a winner computing element by the second requester identifier matching the winner computing element identifier;
19. The computing device of claim 18, further comprising:
- means for identifying the winner computing element from the winner computing element identifier;
 - means for determining whether a criterion is met for adjusting a second resource of the computing device in response to determining that the second computing element is not the winner computing element; and
 - means for adjusting the second resource in response to determining that the criterion is met for adjusting the second resource;
20. The computing device of claim 19, wherein means for determining whether a criterion is met for adjusting a second resource of the computing device comprises means for determining a likelihood of sharing the second resource by the first computing element and the second computing element based on one or more of a shared operating system, shared dynamic voltage and frequency scaling, and a shared topology;
21. The computing device of claim 15, further comprising:
- means for receiving a third resource access request for the first resource including a third requester identifier from a third computing element of the plurality of computing elements; and
 - means for returning the response to the third computing element including the first requester identifier as the winner computing element identifier;
22. The computing device of claim 17, further comprising:
- means for determining whether the second computing element has a task to execute; and

- means for sending a signal to steal a task from the first computing element in response to determining that the second computing element does not have a task to execute, wherein the signal includes the second requester identifier.
- 23.** The computing device of claim 22, further comprising:
- means for receiving a response to the attempt to steal a task including a task winner computing element identifier;
 - means for comparing the second requester identifier to the task winner computing element identifier;
 - means for determining whether the second computing element is a task winner computing element by the second requester identifier matching the task winner computing element identifier; and
 - means for adjusting a task stealing list of the second computing element in response to determining that the second computing element is not the task winner computing element;
- 24.** The computing device of claim 23, wherein means for adjusting the task stealing list of the second computing element comprises means for restraining items in the stealing list based at least in part on whether a computing element is executing a recursive task or a non-recursive task.
- 25.** A non-transitory processor-readable storage medium having stored processor-executable instructions configured to cause a processor of a computing device to perform operations comprising:
- receiving a plurality of resource access requests for a first resource of the computing device from a plurality of computing elements of the computing device including a first resource access request having a first requester identifier from a first computing element of the plurality of computing elements and a second resource access request having a second requester identifier from a second computing element of the plurality of computing elements;
 - granting the first computing element access to the first resource based on the first resource access request; and
 - returning a response to the second computing element including the first requester identifier as a winner computing element identifier;
- 26.** The non-transitory processor-readable storage medium of claim 25, wherein the stored processor-executable instructions are configured to cause the processor to perform operations further comprising:
- comparing the second requester identifier to the winner computing element identifier; and
 - determining whether the second computing element is a winner computing element by the second requester identifier matching the winner computing element identifier;
- 27.** The non-transitory processor-readable storage medium of claim 26, wherein the stored processor-executable instructions are configured to cause the processor to perform operations further comprising:
- identifying the winner computing element from the winner computing element identifier;
 - determining whether a criterion is met for adjusting a second resource of the computing device in response to
 - determining that the second computing element is not the winner computing element; and
 - adjusting the second resource in response to determining that the criterion is met for adjusting the second resource;
- 28.** The non-transitory processor-readable storage medium of claim 27, wherein the stored processor-executable instructions are configured to cause the processor to perform operations such that determining whether a criterion is met for adjusting a second resource of the computing device comprises determining a likelihood of sharing the second resource by the first computing element and the second computing element based on one or more of a shared operating system, shared dynamic voltage and frequency scaling, and a shared topology.
- 29.** The non-transitory processor-readable storage medium of claim 28, wherein the stored processor-executable instructions are configured to cause the processor to perform operations further comprising:
- receiving a third resource access request for the first resource including a third requester identifier from a third computing element of the plurality of computing elements; and
 - returning the response to the third computing element including the first requester identifier as the winner computing element identifier;
- 30.** The non-transitory processor-readable storage medium of claim 28, wherein the stored processor-executable instructions are configured to cause the processor to perform operations further comprising:
- determining whether the second computing element has a task to execute; and
 - sending a signal to steal a task from the first computing element in response to determining that the second computing element does not have a task to execute, wherein the signal includes the second requester identifier;
- 31.** The non-transitory processor-readable storage medium of claim 30, wherein the stored processor-executable instructions are configured to cause the processor to perform operations further comprising:
- receiving a response to the signal to steal a task including a task winner computing element identifier;
 - comparing the second requester identifier to the task winner computing element identifier;
 - determining whether the second computing element is a task winner computing element by the second requester identifier matching the task winner computing element identifier; and
 - adjusting a task stealing list of the second computing element in response to determining that the second computing element is not the task winner computing element;
- 32.** The non-transitory processor-readable storage medium of claim 31, wherein the stored processor-executable instructions are configured to cause the processor to perform operations such that adjusting the task stealing list of the second computing element comprises restraining items in the stealing list based at least in part on whether a computing element is executing a recursive task or a non-recursive task.

* * * *

6 Actividades científicas y tecnológicas

6.1 Publicaciones, documentos científicos y técnicos

Light NUCA: a proposal for bridging the inter-cache latency gap

Darío Suárez†, Teresa Monreal†, Fernando Vallejo‡, Ramón Beivide‡, and Víctor Viñals†

†GazD-DHS-I3A
Universidad de Zaragoza, Spain

‡Computer Architecture Group
Universidad de Cantabria, Spain

HiPEAC Network of Excellence

Email: {dario, tmonreal, victor}@unizar.es Email: {fernando.vallejo, ramon.beivide}@unican.es

Abstract—To deal with the “memory wall” problem, microprocessors include large secondary on-chip caches. But as these caches enlarge, they originate a new latency gap between them and fast L1 caches (inter-cache latency gap). Recently, Non-Uniform Cache Architectures (NUCAs) have been proposed to sustain the size growth trend of secondary caches that is threatened by wire-delay problems. NUCAs are size-oriented, and they were not conceived to close the inter-cache latency gap. To tackle this problem, we propose Light NUCA (L-NUCAs) leveraging on-chip wire density to interconnect small tiles through specialized networks, which convey packets with distributed and dynamic routing. Our design reduces the tile delay (cache access plus one-hop routing) to a single processor cycle and places cache lines at a finer granularity than conventional caches, reducing cache latency. Our evaluations show that in general, an L-NUCA improves simultaneously performance, energy, and area when integrated into both conventional or D-NUCA hierarchies.

I. INTRODUCTION

As technology scales, the latency gap between the processor and main memory widens forcing a size and latency increment in secondary caches. So, at the same time these large caches reduce their gap with respect to main memory, they widen an inter-cache latency gap between them and fast L1 caches. To bridge this gap, two main approaches can be adopted; either reducing the latency of secondary caches or increasing the size of first level caches without compromising their latency and bandwidth.

Within the first approach, Kim *et al.* proposed Non-Uniform Cache Architectures [1]. A NUCA connects individually accessible cache banks in a 2D-mesh. NUCA pioneers inter-bank block migration techniques, but has solely focused on large secondary caches. In respect of the second approach, Balasubramonian *et al.* provided evidence of the latency/size trade-off between L1 and L2 caches. They proposed a reconfigurable cache able to dynamically adjust its size to the working set [2]. However, this scheme only supports single-ported cells, and it may not be able to provide the high bandwidth that superscalar processors require.

The present work tries to close the latency gap between secondary on-chip caches and fast L1 caches by enlarging the cache accessible by the processor at low latencies without degrading bandwidth. Our proposal is based on a light dynamic NUCA formed by small cache banks that benefits from the fine granularity and working set adaptability of the Balasubramonian approach and from the non-uniform access time and

block-migration techniques of NUCAs. To make feasible this idea, we have to fight against the reasons that, up to now, have prevented NUCAs to be used with small cache banks.

To maximize performance, NUCA banks tend to be large [3]. Since NUCA network mechanisms have been designed for multi-megabyte caches, they focus on density rather than latency and bandwidth, e.g., the 2D-mesh network with wormhole routing requires at least one routing cycle before and after accessing any bank. Besides, NUCAs employ a single-injection point and shared banks with multiple cycle initiation rate that can stall the network in miss bursts.

Since NUCA latency and bandwidth mostly depend on its networking, L-NUCA focuses on improving topologies, routing and packet delivery. In an L-NUCA cache, the L1 cache is surrounded by a set of small tiles connected by three on-chip networks specially tuned for different cache operations. L-NUCA tiles manage a cache access and one hop routing within a single cycle. This allows to place blocks at latencies inversely proportional to their temporal locality at a finer granularity than conventional or NUCA hierarchies.

The rest of this paper is organized as follows. Section II presents the main features of L-NUCAs. Section III details the topologies, routing algorithms and flow control policies. Sections IV and V describe the experimental methodology and evaluate the results. Section VI comments on related work, and Section VII concludes the paper.

II. L-NUCA OVERVIEW AND ITS INTEGRATION IN THE CACHE HIERARCHY

We have studied L-NUCA in two representative environments: a conventional 3-level hierarchy in which an L-NUCA replaces the L2 cache (Figs. 1(a) and 1(b)) and a D-NUCA hierarchy in which an L-NUCA is placed between the L1 and the D-NUCA (Figs. 1(c) and 1(d)).

The processor interfaces the L-NUCA through the *root-tile*, which is a conventional L1 cache extended with the flow control logic required for sending and receiving blocks. The rest of tiles surround the r-tile and are interconnected only through local links in order to minimize wire delay. To simplify block migration, all the tiles share the same block size. Block search is efficiently performed by grouping tiles into growing size levels. For example, the L-NUCA of Fig. 1(b) has 4 levels, named *Lei*. The r-tile forms the first

6.8% for Integer and Floating Point, respectively. Besides, gains are consistent across benchmarks and in 54% of them IPC improves more than 10%.

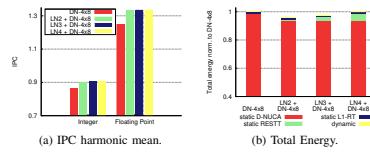


Fig. 5. Average IPC and Total Energy for L1 + D-NUCA and L-NUCA + D-NUCA configurations.

Regarding energy, Fig. 5(b) shows the total energy consumption relative to $DN\text{-}4x8$. Once again, the execution time reduction helps to decrease the energy consumption from 4.25% ($LN2 + DN\text{-}4x8$) to 0.2% ($LN4 + DN\text{-}4x8$). Interestingly, $LN2 + DN\text{-}4x8$ saves 19.8% of dynamic energy with regards to $DN\text{-}4x8$ because the added activity in the L-NUCA (8KB tiles, simple networking) requires less energy than the dynamic activity removed in the D-NUCA (256KB banks, virtual channel routing).

Summarizing, with a negligible 1.2% area increment, adding an $LN2\text{-}72KB$ to a D-NUCA hierarchy improves IPC by 4.2% and 6.8% for Integer and Floating Point benchmarks, respectively and saves 4.25% in total energy consumption.

VI. RELATED WORK

In addition to the related work presented in Section I, multiple authors have studied the on-die cache latency gap. For example, Beckmann and Wood proposed the use of transmission lines to reduce the wire delay [18]. Chishti *et al.* proposed NuRAPID decoupling the placement between tag and data [19]. Regarding NUCA improvements, Jin *et al.* proposed a novel router, a replacement algorithm, and a heterogeneous halo topology [7]. Muralimanohar and Balasubramonian introduced heterogeneity in the wires and in the topology with a mixed point-to-point bus network [3]. All these works focus on multi-megabyte caches, while L-NUCAs focus on size-reduced caches.

VII. CONCLUSIONS

The inclusion of large secondary on-chip caches for closing the latency gap between the processor and main-memory causes another latency gap between those large caches and the fast and small first level caches. This work tackles this problem by extending the cache size reachable by the processor at very low latencies. This objective is achieved by adapting NUCA caches to size-reduced caches.

One of the main novelties of L-NUCAs is their interconnection system. Three different networks have been conceived for the basic cache operations: search, transport, and replacement. All of them are based on short and scalable local links,

Besides, it supports fast lookup and block delivery in a fully-associate structure maximizing hit ratios. Routing is implicit in all the networks minimizing cost and increasing message delivery. With this interconnection fabric, L-NUCA performs in parallel a cache access and one-hop routing in a single cycle.

Our detailed simulations show that, in general, L-NUCA improves simultaneously performance, energy, and area when integrated into both conventional or D-NUCA hierarchies.

ACKNOWLEDGMENTS

The authors would like to thank Brian Greskamp, the members of the gaZ research group, Noemí López, and the anonymous reviewers for their suggestions on this paper. This work was supported in part by the Gobierno de Aragón grant “gaZ: Grupo Consolidado de Investigación”, the Spanish Ministry of Education and Science under contracts TIN2007-66423, TIN2007-68023-C02-01, and Consolider CSD2007-00050, and the European Union Network of Excellence HiPEAC-2 (FP7/ICT 217068).

REFERENCES

- [1] C. Kim, D. Burger, and S. W. Keckler, “An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches,” in *ASPLOS-X*, 2002.
- [2] R. Balasubramonian, D. Albonesi, A. Buyukosmoglu, and S. Dwarkadas, “Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures,” in *MICRO*, 2000.
- [3] N. Muralimanohar and R. Balasubramonian, “Interconnect design considerations for large NUCA caches,” in *ISCA*, 2007.
- [4] L. A. Barroso, K. Ghazanfloo, R. McNamara, A. Nowatzky, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Vargheseemph, “Pinwheel: a scalable architecture based on single-chip multiprocessors,” in *ISCA*, 2000.
- [5] W. J. Dally and B. P. Towles, *Principles and Practices of Interconnection Networks*, Morgan Kaufmann, 2004.
- [6] D. W. Plass and Y. H. Chan, “IBM POWER6 SRAM arrays,” *IBM J. of Research and Development*, vol. 51, no. 6, pp. 747–756, 2007.
- [7] Y. Jin, E. J. Kim, and K. H. Yun, “A domain-specific on-chip network design for large scale cache systems,” in *HPCA*, 2007.
- [8] S. Thazipoor, N. Muralimanohar, and N. P. Jouppi, “Cacti 5.0,” HP Labs, Tech. Rep. HPL-2007-167, October 2007.
- [9] A. Kumar, P. Kundu, A. P. Singh, L.-S. Pehy, and N. K. Jha, “A 4.6Tbit/s 3.6GHz single-cycle NcC router with a novel switch allocator in 65nm CMOS,” in *ICCD*, 2007.
- [10] L.-S. Peh and W. J. Dally, “A delay model and speculative architecture for pipelined routers,” in *HPCA*, 2001.
- [11] H. Wang, L.-S. Peh, and S. Malik, “Power-driven design of router microarchitectures in on-chip networks,” in *MICRO*, 2003.
- [12] S. Goelman, A. Mendelson, A. Naveh, and E. Rotem, “Introduction to Intel Core Duo processor architecture,” *Intel Tech. J.*, vol. 10, 2006.
- [13] A. Phansalkar, A. Joshi, and L. K. John, “Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite,” in *ISCA*, 2007.
- [14] G. Hamerly, E. Perelman, J. Lau, and B. Calder, “SimPoint 3.0: Faster and more flexible program phase analysis,” *J. of Instr-Level Parallelism*, vol. 7, 2005.
- [15] Intel Corporation, “Intel® Core™2 Duo Processor E8600, <http://ark.intel.com/cpu.aspx?groupID=35605>”
- [16] H.-S. Wang, X. Zhu, L.-S. Peh, and S. Malik, “Orion: a power-performance simulator for interconnection networks,” in *MICRO*, 2002.
- [17] J. Wu, D. Weiss, C. Morganti, and M. Driessen, “The Asynchronous 24 MB On-Chip Level 3 Cache for a Dual-Core Itanium Architecture Processor,” in *ISSCC*, 2005.
- [18] B. Beckmann and D. Wood, “TLC: Transmission line caches,” in *MICRO*, 2003.
- [19] Z. Chishti, M. D. Powell, and T. N. Vijaykumar, “Distance associativity for high-performance energy-efficient non-uniform cache architectures,” in *MICRO’03*, 2003.

SigRace: Signature-Based Data Race Detection

Abdullah Muzahid
University of Illinois at Urbana-Champaign, USA
muzahid2@illinois.edu

Darío Suárez
Universidad de Zaragoza
Zaragoza, Spain
dario@unizar.es

Josep Torrellas
University of Illinois at Urbana-Champaign, USA
torrella@illinois.edu

Shanxiang Qi
University of Illinois at Urbana-Champaign, USA
sqi2@illinois.edu

ABSTRACT

Detecting data races in parallel programs is important for both software development and production-run diagnosis. Recently, there have been several proposals for hardware-assisted data race detection. Such proposals typically modify the L1 cache and cache coherence protocol messages, and largely lose their capability when lines get displaced or invalidated from the cache. To avoid these shortcomings, this paper proposes a novel approach to hardware-assisted data race detection. The approach, called *SigRace*, relies on hardware address signatures. As a processor runs, the addresses of the data it accesses are automatically encoded in signatures. At certain times, the signatures are automatically passed to a hardware module that intersects them with those of other processors. If the intersection is not null, a data race may have occurred.

This paper presents the architecture of *SigRace*, an implementation, and its software interface. With *SigRace*, caches and coherence protocol messages are unmodified. Moreover, cache lines can be displaced and invalidated with no effect. Our experiments show that *SigRace* is significantly more effective than a state-of-the-art conventional hardware-assisted race detector. *SigRace* finds on average 29% more static races and 107% more dynamic races. Moreover, if we inject data races, *SigRace* finds 150% more static races than the conventional scheme.

Categories and Subject Descriptors

B [Hardware]: B.3 Memory Structures.B.3.2 Design Styles. Subjects: Shared memory; B.3.4 [Reliability, Testing, and Fault-Tolerance]: Error checking.

General Terms

Design, Measurement, Reliability.

Keywords

SigRace, Signature, Timestamp, Data Race, Concurrency Defect, Happened-Before.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISCA'09, June 20–24, 2009, Austin, Texas, USA.
Copyright 2009 ACM 978-1-60558-526-0/09/06 ...\$5.00.

1. INTRODUCTION

With the widespread use of multicore hardware, parallel programming is likely to become more prevalent. At the same time, concurrency bugs are likely to take on a higher profile and become a very costly problem. Consequently, it is crucial to continue developing more effective techniques to detect and fix them.

An important type of concurrency bug is a data race. A data race occurs when two threads access the same variable without an intervening synchronization and at least one of the accesses is a write. The erroneous program behavior caused by the race may only appear under certain access interleavings, making debugging data races notoriously hard.

For this reason, data race detection has been the subject of much work (e.g., [5, 8, 12, 14, 15, 16, 17, 18, 19, 22, 24, 26, 27, 29, 30]), including the development of commercial software tools for race debugging (e.g., [8, 26]) and even the proposal of special hardware structures in the machine (e.g., [12, 18, 19, 30]). In general, there are two approaches to finding data races, namely the lock-set approach, as in Eraser [24], and the happened-before one, as in Thread Checker [8]. The lockset approach is based on the idea that all accesses to a given shared variable should be protected by a common set of locks. Consequently, it tracks the set of locks held while accessing each variable. It reports a violation when the currently-held set of locks (lockset) at two different accesses to the same variable have a null intersection.

The happened-before approach relies on epochs. An epoch is a thread's execution between two consecutive synchronization operations. Each processor has a logical clock, which identifies the epoch that the processor is currently executing. In addition, each variable has a timestamp, which records at which epoch the processor accessed it. When another processor accesses the variable, it compares the variable's timestamp to its own clock, to determine the relationship between the two corresponding epochs: either one logically happened before the other, or the two logically overlap. In the latter case, we have a race.

Race detectors that use these algorithms in software typically induce about 10–50x slowdowns on programs [8, 14, 22, 24]. Such slowdowns can distort the timing of races identified in production runs, and make them hard to find. For this reason, there have been several recent proposals for race detectors with hardware assists [12, 18, 19, 30]. Such schemes should be effective at debugging races in production runs. However, they detect races by augmenting the cache state and the coherence protocol. Specifically, they tag each cache line with a timestamp [12, 18, 19] or a lock-set [30], perform additional operations on local/external access to

SigRace had an average instruction overhead due to re-execution of 22%, a bandwidth overhead of 63 bytes per thousand committed instructions, and an SRAM memory overhead of \approx 9KB per processor.

We are continuing our work in two main directions. The first one involves eliminating or minimizing the need to perform checkpointing — possibly at the cost of more re-execution. The second one involves improving the scalability of the happened-before clocks and RDM design.

7. ACKNOWLEDGMENTS

We thank the anonymous reviewers and the I-ACOMA group members for their comments. This work was supported in part by the National Science Foundation under grants CNS-0720593 and CCR-0325603; Intel and Microsoft under the Universal Parallel Computing Research Center; and gifts from IBM and Sun Microsystems. Suárez was supported by the Gobierno de Aragón under grant “gaZ: Grupo Consolidado de Investigación”, Spanish Ministry of Education and Science under contracts TIN2007-66423 and Consolider CSD2007-00050; and European Union Network of Excellence HiPEAC-2 (FP7/ICT 217068).

8. REFERENCES

- [1] C. Biernia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *International Conference on Parallel Architectures and Compilation Techniques*, October 2008.
- [2] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Comm. of the ACM*, 13(7):422–426, 1970.
- [3] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk enforcement of sequential consistency. In *International Symposium on Computer Architecture*, June 2007.
- [4] L. Ceze, J. Tuck, J. Torrellas, and C. Cascalav. Bulk disambiguation of speculative threads in multiprocessors. In *International Symposium on Computer Architecture*, June 2006.
- [5] J.-D. Choi, K. Lee, A. Loginov, R. O’Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Programming Language Design and Implementation*, June 2002.
- [6] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55, 1998.
- [7] C. Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24(8):28–33, 1991.
- [8] Intel Corporation. Intel Thread Checker. <http://www.intel.com>, 2008.
- [9] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. of the ACM*, 21(7):558–565, 1978.
- [10] C.-K. Luk et al. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation*, June 2005.
- [11] E. Lusk, J. Boyle, R. Butler, T. Disz, B. Glickfield, R. Overbeek, J. Patterson, and R. Stevens. *Portable programs for parallel processors*. Holt, Rinehart & Winston, 1988.
- [12] S. L. Min and J.-D. Choi. An efficient cache-based access anomaly detection scheme. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.
- [13] C. C. Minh et al. An effective hybrid transactional memory system with strong isolation guarantees. In *International Symposium on Computer Architecture*, June 2007.
- [14] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. In *Programming Language Design and Implementation*, June 2007.
- [15] R. H. B. Netzer and B. P. Miller. Detecting data races in parallel program executions. In *In Workshop on Advances in Languages and Compilers for Parallel Computing*, 1990.
- [16] R. H. B. Netzer and B. P. Miller. Improving the accuracy of data race detection. In *Principles and Practice of Parallel Programming*, April 1991.
- [17] R. O’Calahan and J.-D. Choi. Hybrid dynamic data race detection. In *Principles and Practice of Parallel Programming*, June 2003.
- [18] M. Prvulovic. CORD: Cost-effective (and nearly overhead-free) order-recording and data race detection. In *International Symposium on High-Performance Computer Architecture*, February 2006.
- [19] M. Prvulovic and J. Torrellas. ReEnact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *International Symposium on Computer Architecture*, June 2003.
- [20] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *International Symposium on Computer Architecture*, May 2002.
- [21] F. Qin, J. Tucek, Y. Zhou, and J. Sundaresan. Rx: Treating bugs as allergies—a safe method to survive software failures. *ACM Transactions on Computer Systems*, 25(3):7, 2007.
- [22] M. Ronse and K. De Bosscher. RecPlay: A fully integrated practical record/replay system. *ACM Transactions on Computer Systems*, 17(2):133–152, 1999.
- [23] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam. Implementing signatures for transactional memory. In *International Symposium on Microarchitecture*, December 2007.
- [24] S. Savage et al. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.
- [25] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *USENIX Annual Technical Conference*, June 2004.
- [26] Sun Microsystems. Sun Studio Thread Analyzer. <http://developers.sun.com/sunstudio>, 2007.
- [27] C. von Praun and T. R. Gross. Object race detection. In *Object-Oriented Programming, Systems, Languages, and Applications*, October 2001.
- [28] L. Yen et al. LogTM-SE: Decoupling hardware transactional memory from caches. In *International Symposium on High Performance Computer Architecture*, February 2007.
- [29] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *Symposium on Operating Systems Principles*, October 2005.
- [30] P. Zhou, R. Teodorescu, and Y. Zhou. HARD: Hardware-assisted lockset-based race detection. In *International Symposium on High Performance Computer Architecture*, February 2007.

LP-NUCA: Networks-in-Cache for High-Performance Low-Power Embedded Processors

Darío Suárez Gracia, *Student Member, IEEE*, Giorgos Dimitrakopoulos, *Member, IEEE*, Teresa Monreal Arnal, Manolis G. H. Katevenis, and Víctor Viñals Yúfera, *Member, IEEE*

Abstract—High-end embedded processors demand complex on-chip cache hierarchies satisfying several contradicting design requirements such as high-performance operation and low energy consumption. This paper introduces light-power (LP) nonuniform cache architecture (NUCA), a tiled-cache addressing both goals. LP-NUCA places a group of small and low-latency tiles between the L1 and the last level cache (LLC) that adapt better to the application working sets and keep most recently evicted blocks close to L1. LP-NUCA is built around three specialized “networks-in-cache,” each aimed at a separate cache operation. To prove the design feasibility, we have fully implemented LP-NUCA in a 90-nm technology. From the VLSI implementation, we observe that the proposed networks-in-cache incur minimal area, latency, and power overhead. To further reduce the energy consumption, LP-NUCA employs two network-wide techniques (miss wave stopping and sectoring) that together reduce the dynamic cache energy by 35% without degrading performance. Our evaluations also show that LP-NUCA improves performance with respect to cache hierarchies similar to those found in high-end embedded processors. Similar results have been obtained after scaling to a 32-nm technology.

Index Terms—Cache organization, interconnection networks, low-power design, network-on-chip, nonuniform cache architecture (NUCA), VLSI.

I. INTRODUCTION

THE complexity and variety of embedded applications are constantly increasing, thus demanding systems with high computing capacities and low power consumption. In order to

Manuscript received November 22, 2010; revised March 14, 2011; accepted May 05, 2011. Date of publication July 07, 2011; date of current version June 14, 2012. This work was supported in part by the HiPEAC European Network of Excellence through a collaboration grant and postdoctoral fellowship, by the Spanish Government and European ERDF under Grant TIN2007-66423 and Grant TIN2010-21291, the Aragón Government and European ESF through the gaZ: T48 Research Group, the Spanish Government under Consolider CSD2007-00050, and HiPEAC-2 NoE under Grant European FP7/ICT 217068. D. S. Gracia and V. V. Yúfera are with the Computer Architecture Group (gaZ), Departamento de Informática e Ingeniería de Sistemas, Instituto de Investigación en Ingeniería de Aragón, Universidad de Zaragoza, E-50018 Zaragoza, Spain (e-mail: dario@unizar.es; vici@unizar.es).

G. Dimitrakopoulos is with the Informatics and Communications Engineering Department, University of West Macedonia, GR-50100 Kozani, Greece (e-mail: gdimitrak@uowm.gr).

T. M. Arnal is with the Department of Computer Architecture, Universitat Politècnica de Catalunya (UPC), E-08034 Catalonia, Spain, and also with the Computer Architecture Group (gaZ), Universidad de Zaragoza, E-50018 Zaragoza, Spain (e-mail: teresa@ac.upc.edu).

M. G. H. Katevenis is with the Foundation for Research and Technology—Hellas, Institute of Computer Science (FORTH-ICS)—Computer Architecture and VLSI Systems (CARV) Laboratory, GR-70013 Heraklion, Crete, and also with the Department of Computer Science, University of Crete, GR-71409 Heraklion, Greece (e-mail: kateven@ics.forth.gr).

Color versions of one or more of the figures in this paper are available online at <http://ieeexplore.ieee.org>.

Digital Object Identifier 10.1109/TVLSI.2011.2158249

TABLE I
REPRESENTATIVE PROCESSOR SAMPLES OF CURRENT HIGH-END SOCS
(40–45 nm). SHADED CELLS SHOW THE LAST LEVEL CACHE (LLC)

Model, Brand	# threads	L1 / CPU	CPU (KB)	L2 / CPU	L3 (MB)
P5010, Freescale [6]	1	32	32	512KB	1
476fP, IBM-LSI [3]	1	32	32	512KB	2 shared + 4 eDRAM
LC3528, Intel [1]	2	32	32	256KB	4 shared
Cortex-A9, ARM [2]	1	32	32	up to 8MB	—
MIPS32-1004K, MIPS [4]	2	32	32	up to 1MB	—
XLP832, NetLogic [5]	4	64	32	512KB	8 shared

execute these applications and reduce costs embedded systems are integrated into system-on-chips (SoCs). Many manufacturers offer SoCs meeting these stringent requirements. To reach the performance goal, SoCs rely on concepts and techniques previously proposed for the high-performance computing segment, but tuned for minimizing energy consumption.

A review of current advanced embedded SoCs (40–45 nm) clearly shows how their CPUs follow close behind their high-performance siblings. Among other advanced features (see Table I), embedded processors profit from out-of-order wide issue or multithreaded execution [1]–[6].

These techniques such as wide issue and multithreading put pressure on the cache hierarchy, so its design also inherits concepts and techniques from the high-performance segment. For example, we can see multiported L1 data caches optimized for speed [5] and relatively big LLCs optimized for size.

An appealing proposal that has received much attention for improving the performance of the cache hierarchy has been the nonuniform cache architecture (NUCA) [7]–[21]. NUCA tackles the wire delay problem in LLCs¹ by merging the L2 and the L3 into a mesh of cache banks and enabling inter-bank block migration. Hence, blocks located in the banks close to the cache controller have a lower latency than those located further apart [7].

Nevertheless, there are almost no examples of NUCA caches in high-performance embedded SoCs. A close approach is the L3 cache of NetLogic XLP832 [5]. This LLC is divided into eight independent banks, and each bank is attached to a bidirectional ring that also communicates with four DDR ports and eight private L2 caches. Since L3 cache lines are statically mapped into banks, the NetLogic solution resembles the static NUCA proposal, but with a topological change: from the original mesh to a ring.

¹Driving a signal from the cache controller to the banks takes more time than the bank access itself.

- [42] M. Shah, J. Barren, J. Brooks, R. Golla, G. Grohoski, N. Gura, R. Hetherington, P. Jordan, M. Luttrell, C. Olson, B. Sims, D. Sheehan, L. Sprakler, and A. Wynn, "UltraSPARC T2: A highly-treaded, power-efficient, SPARC SOC," in *Proc. Asian Solid-State Circuits Conf.*, 2007, pp. 22–25.
- [43] P. Conway, N. Kannanandaram, G. Donley, K. Lepak, and B. Hughes, "Cache hierarchy and memory subsystem of the AMD opteron processor," *IEEE Micro*, vol. 30, no. 2, pp. 16–29, 2010.
- [44] T. Lundqvist and P. Stenström, "A method to improve the estimated worst-case performance of data caching," in *Proc. 6th Int. Conf. Real-Time Computing Syst. Applications*, 1999, pp. 255–262.
- [45] J. Staschula and R. Ernst, "Worst case timing analysis of input dependent data cache behavior," in *Proc. 18th Euromicro Conf. Real-Time Syst.*, 2006, pp. 227–236.
- [46] H. Ramapand and F. Mueller, "Bounding preemption delay within data cache reference patterns for real-time tasks," in *Proc. 12th Real-Time Embedded Technol. Applications Symp.*, 2006, pp. 71–80.
- [47] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Berner, C. Ferder, R. Heckmann, T. Mära, F. Mueller, I. Puaut, P. Puschner, J. Staschula, and P. Stenström, "The worst-case execution time problem: overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, pp. 36:1–36:53, 2008.
- [48] X. Vera, B. Lisper, and J. Xue, "Data cache locking for higher program predictability," in *Proc. Int. Conf. Meas. Modeling Comput. Syst.*, 2003, pp. 272–282.
- [49] V. Suhendra and T. Mitra, "Exploring locking and partitioning for predictable shared caches on multi-cores," in *Proc. 45th Annu. Design Autom. Conf.*, 2008, pp. 300–303.



Dario Suárez Gracia (S'08) received the M.S. degree in computer engineering from the Universidad de Zaragoza, Zaragoza, Spain, in 2003, where he is currently working toward the Ph.D. degree.

His interests include memory microarchitecture and implementation, low-power design, on-chip networks, multicore, and multithreaded processors, and efficient simulation.

Mr. Suárez Gracia is a student member of the Association for Computing Machinery and the IEEE Computer Society.



Giorgos Dimitrakopoulos (M'07) received the Dipl.-Ing in computer engineering, M.Sc. degree in integrated hardware-software systems, and Ph.D. degree from the University of Patras, Patras, Greece, in 2001, 2003, and 2007, respectively.

The next two years he was a Postdoctoral Researcher with the Institute of Computer Science of the Foundation for Research and Technology—Hellas (FORTH). During that period, he also taught digital-design-related courses to the Computer Science Department of the University of Crete. In September 2009, he moved to the Informatics and Communication Engineering Department, University of West Macedonia, Kozani, Greece, where he is now a Lecturer and is actively involved in the design of low cost on-chip interconnection networks architectures.



Teresa Monreal Arnal received the M.S. degree in mathematics and Ph.D. degree in computer science from the University of Zaragoza, Zaragoza, Spain, in 1991 and 2003, respectively.

She is an Associate Professor with the Computer Architecture Department, Universitat Politècnica de Catalunya, Catalonia, Spain. Her research covers register files, memory hierarchy, and high-performance architectures. Until 2007, she was with the Informática e Ingeniería de Sistemas Department, University of Zaragoza, Zaragoza, Spain. She collaborates actively with the Grupo de Arquitectura de Computadores from the University of Zaragoza (gaZ).



Manolis G. H. Katevenis received the Ph.D. degree from the University of California, Berkeley, in 1983.

After a brief term on the faculty of Computer Science, Stanford University, Stanford, CA, he is currently a Professor with the University of Crete and with FORTH, Heraklion, Crete, since 1986. After RISC, his research has been on interconnection networks and interprocessor communication. In packet switch architectures, his contributions since 1987 have been mostly in per-flow queuing, credit-based flow control, congestion management, scheduling, buffered crossbars, and non-blocking switching fabrics. In multiprocessing and clustering, his contributions since 1993 have been on remote-write-based, protected, user-level communication.

Dr. Katevenis was the recipient of the ACM Doctoral Dissertation Award in 1984 for his thesis on "Reduced Instruction Set Computer Architectures for VLSI."



Victor Vinals Yífera (M'92) received the M.S. degree in telecommunications and Ph.D. degree in computer science from the Universitat Politècnica de Catalunya, Catalonia, Spain, in 1982 and 1987, respectively.

He was an Associate Professor with the Facultat d'Informàtica de Barcelona (UPC) during 1983–1988. Currently, he is a Full Professor with the Informática e Ingeniería de Sistemas Department, University of Zaragoza, Zaragoza, Spain. His research interests include processor microarchitecture, memory hierarchy, and parallel computer architecture.

Dr. Yífera is a member of the Association for Computing Machinery and the IEEE Computer Society, as well as the Computer Architecture Group of the University of Zaragoza.

Dynamic Construction of Circuits for Reactive Traffic in Homogeneous CMPs

Marta Ortín¹, Darío Suárez¹, María Villarroya¹, Cruz Izu², Víctor Viñals¹

¹Departamento de Informática e Ingeniería de Sistemas, I3A, University of Zaragoza, Spain.

Email: {ortin, dario, mvg, victor}@unizar.es

²School of Computer Science, University of Adelaide, Australia. Email: cruz@cs.adelaide.edu.au

Abstract—Networks on Chip (NoCs) have a large impact on system performance, area and energy. Considering the characteristics of the memory subsystem while designing the NoC helps identify improvement opportunities and build more efficient designs. Leveraging the frequent request-reply pattern, our proposal dynamically builds the reply path in advance, is able to share circuits between messages, and even removes some implicit replies, significantly reducing NoC latency. A careful implementation of this circuit reservation mechanism achieves an average 17% reduction in router energy consumption, 8% smaller router area and a 2% system performance increase, compared with its baseline counterpart.

I. INTRODUCTION

The design of multicore Networks on Chip (NoCs) can take advantage of the reactive nature of the traffic among nodes. This paper presents and evaluates a novel approach of dynamic virtual circuits for homogeneous chip multiprocessors (CMPs) with 16 and 64 cores connected by a mesh.

Analyzing the coherency protocol in a standard wormhole 4-stage pipeline router, we detected that the request-reply pattern dominates over the rest. In average, 53% of the messages are a reply to another message and, therefore, we know their source and destination before the message is injected into the network. With this information, routers can reserve in advance crossbar path and output virtual channels, removing those stages from the critical path. We also observed that the network is lightly loaded (nodes inject, in average, less than 4 fits every 100 cycles) suggesting it will be feasible to reserve resources for longer periods of time.

This simple yet efficient dynamic circuit approach reduces network latency and achieves better performance than the baseline router. At the same time, it significantly reduces router area and energy consumption by removing buffer space. These results emphasize the importance of considering the system as a whole and studying how all the elements interact [1].

II. STATE OF THE ART

Several works have proposed hybrid packet-circuit switching techniques to speed up certain messages. Most mechanisms establish circuits between nodes using dedicated networks. Some proposals have separate networks for packet and circuit switched messages [2], [3], while others implement a single network that supports both types of traffic [4], [5]. A different technique preallocates resources in advance to allow faster data transmission [6], [7].

Another common approach to reduce network latency involves routers that speculate by using paths without prior reservation, which only work if there is no contention [8].

978-3-9815370-2-4/DATE14/©2014 EDAA

TABLE I: Main characteristics of the chip multiprocessor.

Processors	16 y 64, Ultrasparc III Plus, in order, IPC 1, single-threaded, 1.6 GHz
Coherence	Directory based, MESI, directory distributed in the L2 banks
L1 cache	32KB data and instruction caches, 4-way assoc., 2-cycle hit, 64B lines, private, pseudo-LRU replacement
L2 cache	Distributed, 1 bank/node, 1MB/bank, 16-way assoc., 7-cycle hit, 64B lines, shared, inclusive, pseudo-LRU replacement
Memory	4 memory controllers distributed in the edges of the chip (in both 16 and 64-node chips), 160-cycle latency

TABLE II: Messages generated by the coherence protocol.

Event	Sequence of messages
L1 miss	1 ^o Request from L1 to L2 2 ^o L2_Replies: Reply data from L2 to L1 3 ^o L1_DATA_ACK: Data reception ACK from L1 to L2
L1 miss, another L1 owns the data exclusively	1 ^o Request from L1 to L2 2 ^o L2 forwards the request to L1 owner 3 ^o L1_TO_L1: L1 owner sends data to L1 requestor 4 ^o L1_DATA_ACK: Data ACK from L1 requestor to L2
Invalidation (write or L2 repl)	1 ^o Invalidations from L2 to L1 sharers 2 ^o L1_INV_ACK: ACK from L1s to L2
L1 replacement	1 ^o Replacement information from L1 to L2 2 ^o L2_WB_ACK: ACK from L2 to L1
L2 miss	1 ^o Request from L2 to main memory 2 ^o MEMORY: Data from main memory to L2
L2 replacement	1 ^o Replacement information from L2 to main memory 2 ^o MEMORY: ACK from main memory to L2

[9]. These routers are more complex and may require reduced network frequency or result in energy and performance penalties when the implemented shortcuts cannot be used.

Contrary to previous approaches, our work does not require extra networks, additional messages, gathering statistics, or modifying the coherence protocol. Our proposal leverages the memory hierarchy behaviour to efficiently reserve network resources in advance with minimal changes in the routers.

III. DYNAMIC CIRCUIT CONSTRUCTION

This section presents the characteristics of the CMP and the baseline interconnection network. After that, it explains the mechanism to dynamically build and use circuits to reduce communication latency.

A. System architecture

This work focuses on a homogeneous CMP where each tile is composed of a single-threaded core with private first level cache and a bank of the shared second-level cache, both connected directly to the router. Table I summarizes the key parameters of the architecture and Table II details the messages exchanged by our MESI coherency protocol.

The baseline NoC is built as a mesh with simple 4-stage routers, dimension order routing and wormhole flow control. Table III includes the detailed configuration of the baseline NoC.

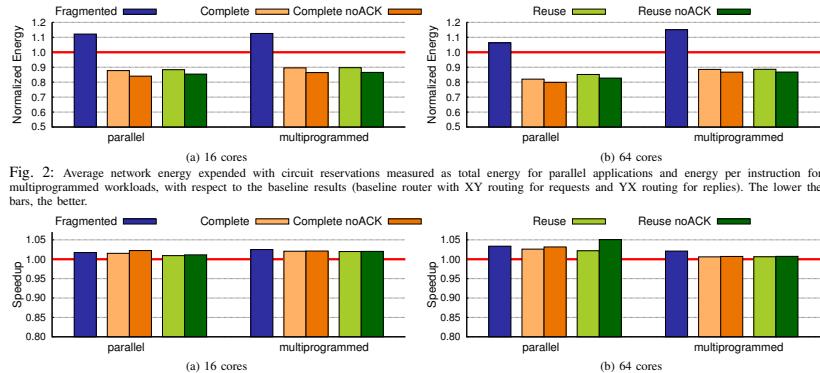


Fig. 2: Average network energy expended with circuit reservations measured as total energy for parallel applications and energy per instruction for multiprogrammed workloads, with respect to the baseline results (baseline router with XY routing for requests and YX routing for replies). The lower the bars, the better.

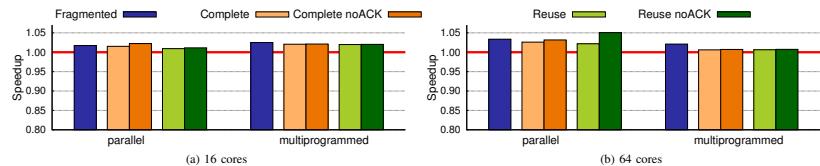


Fig. 3: Average speedup achieved with circuit reservations measured as number of execution cycles for parallel applications and number of completed instructions for multiprogrammed workloads, normalized to the baseline results (baseline router with XY routing for requests and YX routing for replies).

the implementation of our mechanism because all replies that do not have a circuit must contend for a single virtual channel, causing our overall average latency reduction to drop to 7%. Techniques that reduce traffic in that virtual channel, such as eliminating redundant coherence messages, help reduce this undesirable effect and improve performance.

Comparing the results for the 16-node and the 64-node NoCs, we see that both speedup and energy savings are higher in the latter case, even though less percentage of replies manage to get a complete circuit. That is because the impact of the NoC is bigger on a larger chip; more messages are sent per cycle and their latency is larger. This shows that the method scales well and shows a lot of promise for future larger chips.

We also checked that the effectiveness of our mechanism is independent from the L2 latency, even though circuits remain built during the L2 cache access. The difference in speedups for latencies of 1, 4, 7 (reported by DSENT), 12, and 20 cycles in 16 and 64-core chips with the best configuration (building complete circuits and removing coherence messages) is always smaller than 0.01.

V. CONCLUSIONS

This paper was inspired by the observation that most of the traffic follows a request-reply pattern, which helps anticipate the path for most replies. We have used this information to reserve network resources and dynamically build the circuit for the reply while the request travels through the network. Consequently, reply messages with a set-up circuit can go through the router in a single cycle, compared with the 4 cycles needed in the baseline router. Guaranteeing complete circuits for data messages has also enabled us to predict when they will reach their destination, and elegantly eliminate the need for their acknowledgement. To evaluate the proposal, we have performed full-system simulation with realistic parallel and multiprogrammed workloads. For a 64-core chip, our proposal achieves an average energy reduction of 17% at the router, 8% smaller area, and speedups of 2%. Results are better for

the 64-core configuration than for the 16-core one and are not dependent on L2 access latency, which shows that the mechanism scales well and will benefit future designs.

ACKNOWLEDGMENTS

This work was supported in part by grants TIN2010-21291-C02-01 (Spanish Government, European ERDF), gaZ: T48 research group (Aragón Government and European ESF), Consolider CSD 2007-00050 (Spanish Government), and HiPEAC-3 NoE (European FP7/ICT 217068).

REFERENCES

- [1] R. Kumar, V. Zyuban, and D. M. Tullsen, "Interconnections in Multi-Core architectures: Understanding mechanisms, overheads and scaling," in *Int. Symp. on Computer Architecture*, 2005, pp. 408–419.
- [2] F. Palumbo, D. Pani, A. Congiu, and L. Raffo, "Concurrent hybrid switching for massively parallel systems-on-chip: the cyber architecture," in *Proc. of the 9th conf. on Computing Frontiers*, 2012, pp. 173–182.
- [3] J. Duato, P. Lopez, F. Sillar, and S. Yalamanchili, "A high performance router architecture for interconnection networks," in *Proc. of the Int. Conf. on Parallel Processing*, 1996, pp. 61–68 vol.1.
- [4] N. D. E. Jerger, L.-S. Peh, and M. H. Lipasti, "Circuit-switched coherence," in *Proc. of the Int. Symp. on Networks-on-Chip*, 2008, pp. 193–202.
- [5] A. Abousamra, A. Jones, and R. Melhem, "Codesign of NoC and cache organization for reducing access latency in chip multiprocessors," *IEEE Trans. on Parallel and Distributed Systems*, pp. 1038–1046, 2012.
- [6] L.-S. Peh and W. Dally, "Flit-reservation flow control," in *Int. Symp. on High-Performance Computer Architecture*, 2000, pp. 73–84.
- [7] C. Lee and N. Jha, "Variable-pipeline-stage router," in *IEEE Trans. on Very Large Scale Integration Systems*, 2012, pp. 1–1.
- [8] R. Mullins, A. West, and S. Moore, "The design and implementation of a low-latency on-chip network," in *Proc. of the Asia and South Pacific Design Automation Conference*, 2006, pp. 164–169.
- [9] L.-S. Peh and W. J. Dally, "A delay model and speculative architecture for pipelined routers," in *Proc. of the Int. Symp. on High-Performance Computer Architecture*, 2001, pp. 255–.

Block Disabling Characterization and Improvements in CMPs Operating at Ultra-low Voltages

Alexandra Ferrerón*, Darío Suárez-Gracia†, Jesús Alastruey-Benedé*, Teresa Monreal‡, and Víctor Viñals*

*Universidad de Zaragoza, Spain, Email: {ferrerón, jalastro, victor}@unizar.es

†Qualcomm Research Silicon Valley, USA, Email: dgracia@qti.qualcomm.com

‡Universidad Politécnica de Cataluña, Spain, Email: teresa@ac.upc.edu

Abstract—Power density has become the limiting factor in technology scaling as power budget restricts the amount of hardware that can be active at the same time. Reducing supply voltage to ultra-low voltage ranges close to the threshold region has the promise of great energy savings. However, the potential savings of voltage scaling are limited by the correct operation of SRAM cells, which is not guaranteed below $V_{dd_{min}}$, the minimum voltage in which cache structures operate reliably.

Understanding the effects of operating below $V_{dd_{min}}$ requires complex modeling, so we introduce an updated probability failure model of SRAM cells at 22nm and explore the reliability impact of lowering the chip voltage supply below $V_{dd_{min}}$ in shared-memory coherent chip-multiprocessors (CMP) running a variety of parallel workloads. A microarchitectural technique to cope with cache reliability at ultra-low voltages is block disabling; however, in many cases, the savings in on-chip cache do not compensate for the consumption in the rest of the system, as the consumption increase of the off-chip memory may offset the on-chip gain.

We make the case that existing coherence mechanisms can provide the substrate to improve energy savings with block disabling and propose two low-complexity techniques. Taking the best of both techniques we can scale voltage below $V_{dd_{min}}$ and reduce system energy up to 39%, and system energy-delay up to 10%. Besides, by lowering the CMP consumption in a power-constrained scenario, we could activate offline cores, reaching a potential speedup between 3.7 and 4.4.

I. INTRODUCTION

Power density has become the limiting factor in technology scaling. Technology improvements allow to increase the number of transistors and integration density following Moore's Law, but the power budget caps the amount of hardware that can be active at the same time, leading to dark silicon [1].

Voltage scaling is one of the most effective mechanisms to reduce microprocessor power consumption, as dynamic power scales quadratically with voltage. Modern systems include dynamic voltage and frequency scaling (DVFS) capabilities, and run at different predefined energy/performance points to trade-off performance for power consumption. Scaling voltage beyond to approach the threshold voltage would allow further reductions into the power consumption of the chip [2].

However, the increased severity of manufacturing-induced parameter variations at lower voltages limits voltage scaling to a minimum voltage, $V_{dd_{min}}$, below which a processor cannot operate reliably. Logic is more voltage scaling friendly than

memory, as memory is sized more aggressively to satisfy high-density requirements. Thus, parameter variations particularly impact memory cells, and failures in memory structures typically determine the $V_{dd_{min}}$ of the whole processor, or implies using different voltage domains for logic and memory. For example, Intel uses 0.7V for logic and 1.05V for memory in one of its latest processors [3].

Overcoming $V_{dd_{min}}$ would allow to operate at lower voltages, improving power consumption and battery life. In the literature different solutions have been proposed to deal with faulty memory cells, such as spare and robust cells, frequency/voltage binning, error correcting codes, or remapping mechanisms to couple faulty entries in order to recreate fault-free cache blocks [4], [5], [6]. In general, these methods have a noticeable area overhead or sacrifice an important fraction of the cache capacity and associativity.

First-level caches in chip multiprocessors (CMPs) are usually private, occupy little area, and their access time often determines the processor cycle time. The use of complex remapping mechanisms may imply penalizing the access time, already critical, and the loss of cache capacity might degrade performance, as the miss rate increases. Commercial processors, such as the Intel Nehalem family, use robust cells (8T SRAM cells) in the first-level caches to improve resilience [7]. On the other hand, last-level caches (LLCs) are usually shared and have bigger size and associativity, occupying a great percentage of the chip area. Reducing LLC capacity and associativity potentially increases the off-chip memory accesses, translating into extra energy consumption that might spoil the energy savings coming from voltage reduction.

Our goal is twofold: to explore the implications of lowering $V_{dd_{min}}$ in shared-memory coherent CMPs and to provide system-conscious block disabling mechanisms enabling ultra-low voltage operation without penalizing LLC associativity.

This work makes the following contributions. First, we present an up-to-date SRAM cell failure probability model (P_{fail}) based on previous models, studies, and future technology predictions, taking into account emerging circuit implementations and topologies. Second, for the best of our knowledge, we provide the first evaluation of block disabling techniques in a shared-memory coherent CMP running parallel workloads with a complete and detailed memory model, including different P_{fail} points and main memory. Off-chip memory energy consumption at lower chip voltages plays an important role, and not considering its contribution might lead to suboptimal design points. Finally, we introduce two low-complexity techniques that allow blocks disabled in the LLC to be present in the L1 caches. The first relies on the existing cache coherence

This work was supported in part by grants TIN2010-21291-C02-01, TIN2012-34557, TIN2013-46957-C2-1-P (Spanish Gov. and European ERDF), gaZ-T48 research group (Aragón Gov. and European ESF), and HiPEAC-3 NoE (European FET FP7/ICT 287759).

TABLE III. MAXIMUM SPEEDUP. THE FIRST EIGHT COLUMNS TO THE RIGHT OF BENCHMARKS NAMES ARE ENERGY-VOLTAGE PAIRS (JOULES-VOLTS) OF MINIMUM ENERGY (BASELINE, BD, BDOT, AND BDOT-C2C, RESPECTIVELY). THE RIGHTMOST THREE COLUMNS ARE THE ENERGY REDUCTION FACTORS, AND HENCE A BOUND ON THE MAXIMUM SPEEDUP. BOLDED NUMBERS HIGHLIGHT THE BEST CONFIGURATIONS.

Benchmark	Base E	Base V_{dd}	BD E	BD V_{dd}	BDOT E	BDOT V_{dd}	C2C E	C2C V_{dd}	Base/BD	Base/BDOT	Base/C2C
blackholes	0.62		0.21	0.55	0.17		0.17		2.94	3.71	3.69
bodytrack	7.99		2.62	0.50	2.02		1.97		3.05	3.96	4.05
cannae	1.05		0.35	0.55	0.28		0.28		3.01	3.69	3.73
dedup	6.04		1.58	0.50	1.39		1.37		3.81	4.34	4.41
ferret	16.59		4.11	0.45	4.22		4.20		4.04	3.94	3.95
fluidanimate	3.98	0.80	1.34	0.55	0.94		0.93		2.96	4.23	4.26
raytrace	1.31		0.43	0.55	0.30		0.30		3.02	4.38	4.38
swaptions	2.54		0.86	0.55	0.61		0.60		2.94	4.18	4.23
vips	9.56		2.44	0.45	2.39		2.38		3.91	3.99	4.01
x264	3.09		0.77	0.45	0.71		0.71		4.04	4.38	4.37

because of the extra main memory accesses, due to the reduction of cache capacity and associativity.

This work proposes two techniques to improve block disabling in CMPs leveraging current coherence mechanisms. First, associativity degradation causes unnecessary invalidations in inclusive hierarchies; however, inclusion is tracked in the directory and the tag array. So we propose to use robust cells in the directory tags to avoid unnecessary invalidations: we call it block disabling with operational tags (BDOT). Second, directories also track replicated blocks among caches, and therefore, request to faulty entries in the LLC cache can be forwarded to a L1 sharer of the block with a valid copy. We call this technique BDOT with cache-to-cache service (BDOT-C2C).

Block disabling (BD), without and with operational tags has been assessed through three different metrics, namely, system energy, system energy-delay product, and CMP energy. Regarding system energy, BD reaches the absolute minimum at 0.55V in most of workloads, while the other two techniques reduce energy consumption towards lower voltages. Regarding system energy-delay, the baseline configuration running at 0.8 is improved by 10% and 1% at 0.7V and 0.6V, respectively, equally well by all three techniques; no need to use operational tags is therefore concluded. Finally, searching for CMP energy reduction in a power-constrained CMP seeks to wake up off-line cores. To this end, both BDOT and BDOT-C2C excel, showing the potential of speedups ranging from 3.7 to 4.4.

REFERENCES

- [1] M. Taylor, "A landscape of the new dark silicon design regime," *IEEE Micro*, vol. 33, no. 5, pp. 8–19, Sept 2013.
- [2] R. Dreslinski *et al.*, "Near-threshold computing: Reclaiming moore's law through energy efficient integrated circuits," *Proceedings of the IEEE*, vol. 98, no. 2, pp. 253–266, Feb 2010.
- [3] R. Zahir *et al.*, "The medfield smartphone: Intel architecture in a handheld form factor," *IEEE Micro*, vol. 33, no. 6, pp. 38–46, Nov 2013.
- [4] A. Ansari *et al.*, "Archipelago: A polyomorphic cache design for enabling robust near-threshold operation," in *IEEE 17th Int. Symp. on High Performance Computer Architecture*, 2011, pp. 539–550.
- [5] C. Wilkerson *et al.*, "Trading off cache capacity for reliability to enable low voltage operation," in *Proc. of the 35th Annual Int. Symp. on Computer Architecture*, 2008, pp. 203–214.
- [6] Z. Chiishi *et al.*, "Improving cache lifetime reliability at ultra-low voltages," in *Proc. of the 42nd Annual IEEE/ACM Int. Symp. on Microarchitecture*, 2009, pp. 89–99.
- [7] R. Kumar *et al.*, "A family of 45nm IA processors," in *IEEE Int. Solid-State Circuits Conf. Digest of Technical Papers*, Feb 2009, pp. 58–59.
- [8] S. R. Nassif *et al.*, "A resilience roadmap (invited paper)," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2010, pp. 1011–1016.
- [9] S. Damaraju *et al.*, "A 22nm IA multi-CPU and GPU System-on-Chip," in *IEEE Int. Solid-State Circuits Conf. Digest of Technical Papers*, Feb 2012, pp. 56–57.
- [10] S.-T. Zhou *et al.*, "Minimizing total area of low-voltage SRAM arrays through joint optimization of cell size, redundancy, and ECC," in *IEEE Int. Conf. on Computer Design*, Oct 2010, pp. 112–117.
- [11] J. Kulkarni *et al.*, "A 160mV' robust schema trigger based subthreshold SRAM," *IEEE Journal of Solid-State Circuits*, vol. 42, no. 10, pp. 2303–2313, 2007.
- [12] J. Chase *et al.*, "The 65-nm 16-MB Shared On-Die L3 Cache for the Dual-Core Intel Xeon Processor 7100 Series," *IEEE Journal of Solid-State Circuits*, vol. 42, no. 4, pp. 846–852, April 2007.
- [13] H. Lee *et al.*, "Performance of graceful degradation for cache faults," in *IEEE Computer Society Annual Symp. on VLSI*, March 2007, pp. 409–415.
- [14] N. Ladis *et al.*, "Performance-effective operation below Vec-min," in *IEEE Int. Symp. on Performance Analysis of Systems Software*, March 2010, pp. 223–234.
- [15] A. BanaiyanMofrad *et al.*, "REMEDIE: A scalable fault-tolerant architecture for low-power NUCA cache in tiled CMPs," in *Int. Green Computing Conf.*, 2013, pp. 1–10.
- [16] U. Karpuzu *et al.*, "Coping with parametric variation at near-threshold voltages," *IEEE Micro*, vol. 33, no. 4, pp. 6–14, July 2013.
- [17] P. Conway *et al.*, "Cache hierarchy and memory subsystem of the AMD opteron processor," *IEEE Micro*, vol. 30, pp. 16–29, 2010.
- [18] M. M. K. Martin *et al.*, "Why on-chip cache coherence is here to stay," *Communications of the ACM*, vol. 55, no. 7, pp. 78–89, Jul 2012.
- [19] J. L. Baer *et al.*, "On the inclusion properties for multi-level cache hierarchies," in *Proc. of the 15th Annual Int. Symp. on Computer Architecture*, 1988, pp. 73–80.
- [20] P. Magnusson *et al.*, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, Feb 2002.
- [21] M. M. K. Martin *et al.*, "MultiThread's General Execution-driven Multi-processor Simulator (GEMS) toolkit," *SIGARCH Comput. Archit. News*, vol. 33, no. 4, pp. 92–99, Nov. 2005.
- [22] N. Agarwal *et al.*, "GARNET: A detailed on-chip network model inside a full-system simulator," in *IEEE Int. Symp. on Performance Analysis of Systems and Software*, 2009, pp. 33–42.
- [23] P. Rosenfeld *et al.*, "DRAMSim2: A cycle accurate memory system simulator," *Computer Architecture Letters*, vol. 10, no. 1, pp. 16–19, Jan 2011.
- [24] S. Li *et al.*, "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *42nd Annual IEEE/ACM Int. Symp. on Microarchitecture*, 2009, pp. 469–480.
- [25] C. Bienia *et al.*, "The PARSEC benchmark suite: characterization and architectural implications," in *Proc. of the 17th Int. Conf. on Parallel Architectures and Compilation Techniques*, 2008, pp. 72–81.
- [26] S. Schuster, "Multiple word/bit line redundancy for semiconductor memories," *IEEE Journal of Solid-State Circuits*, vol. 13, no. 5, pp. 698–703, Oct 1978.
- [27] S. M. Khan *et al.*, "Improving multi-core performance using mixed-cell cache architecture," in *IEEE 19th Int. Symp. on High Performance Computer Architecture*, 2013, pp. 119–130.

Revisiting LP-NUCA Energy Consumption: Cache Access Policies and Adaptive Block Dropping

DARÍO SUÁREZ GRACIA, Qualcomm Research Silicon Valley
 ALEXANDRA FERRERÓN, Universidad de Zaragoza and HiPEAC
 LUIS MONTESANO DEL CAMPO, Universidad de Zaragoza
 TERESA MONREAL ARNAL, Universitat Politècnica de Catalunya and HiPEAC
 VÍCTOR VIÑALS YÚFERA, Universidad de Zaragoza and HiPEAC

Cache working-set adaptation is key as embedded systems move to multiprocessor and Simultaneous Multithreaded Architectures (SMT) because interthread pollution harms system performance and battery life. Light-Power NUCA (LP-NUCA) is a working-set adaptive cache that depends on temporal-locality to save energy. This work identifies the sources of energy waste in LP-NUCAs parallel access to the tag and data arrays of the tiles and low locality phases with useless block migration. To counteract both issues, we prove that switching to serial access reduces energy without harming performance and propose a machine learning Adaptive Drop Rate (ADR) controller that minimizes the amount of replacement and migration when locality is low.

This work demonstrates that these techniques efficiently adapt the cache drop and access policies to save energy. They reduce LP-NUCA consumption 22.7% for 1SMT. With interthread cache contention in 2SMT, the savings rise to 29%. Versus a conventional organization, energy-delay improves 20.8% and 25% for 1- and 2SMT benchmarks, and, in 65% of the 2SMT mixes, gains are larger than 20%.

Categories and Subject Descriptors: B.3.2 [Memory Structures]: Design Styles—*Cache memories*

General Terms: Design, Experimentation, Performance

Additional Key Words and Phrases: NUCA, energy, embedded processors, locality of reference, hill climbing

ACM Reference Format:

Darío Suárez Gracia, Alexandra Ferrerón, Luis Montesano del Campo, Teresa Monreal Arnal, and Victor Viñals Yúfera. 2014. Revisiting LP-NUCA energy consumption: Cache access policies and adaptive block dropping. ACM Trans. Architec. Code Optim. 11, 2, Article 19 (June 2014), 26 pages.
 DOI: <http://dx.doi.org/10.1145/2632217>

1. INTRODUCTION

Due to locality of reference, cache hierarchies speed up applications and extend battery life [Borkar and Chien 2011]. Recent cache approaches, such as NUCA or Light Power NUCA (LP-NUCA), advocate for heavily tiled designs interconnected with scalable networks [Kim et al. 2002; Suárez Gracia et al. 2012]. The key feature is that cache

This work was supported in part by grants TIN2010-21291-C02-01, DPI2011-25892, TIN2012-34557,

TIN2013-46957-C2-1-P, and Consolider NoE TIN2014-52608-REDIC (Spanish Gov; gaZ: T48 research group (Aragón Gov. and European ESF); and HiPEAC-3 NoE (European FET FP7/ICT 287759).

Author's addresses: D. Suárez Gracia (Corresponding author), Qualcomm Research Silicon Valley, Santa Clara, CA, USA; email: dgracia@qti.qualcomm.com; A. Ferrerón, L. Montesano del Campo, and V. Viñals Yúfera, Universidad de Zaragoza, Zaragoza, Spain; T. Monreal Arnal, Universitat Politècnica de Catalunya, Barcelona, Spain.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax 1 (212) 869-0481, or permissions@acm.org.

© 2014 ACM 1544-3566/2014/06-ART19 \$15.00

DOI: <http://dx.doi.org/10.1145/2632217>

ACM Transactions on Architecture and Code Optimization, Vol. 11, No. 2, Article 19, Publication date: June 2014.

- Alex Settle, Dan Connors, Enric Gibert, and Antonio González. 2006. A dynamically reconfigurable cache for multithreaded processors. *Journal of Embedded Computing*, 2, 2 (2006), 221–233.
- Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architecture Support for Programming Languages and Operating Systems*. 45–57.
- D. Suárez Gracia, G. Dimitrakopoulos, T. Monreal Arnal, M. G. H. Katevenis, and V. Viñals Yúfera. 2012. LP-NUCA: Networks-in-cache for high-performance low-power embedded processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 20, 8 (Aug. 2012), 1510–1523.
- Karthik T. Sundararajan, Timothy M. Jones, and Nigel Topham. 2011. Smart cache: A self adaptive cache arch. for energy efficiency. In *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS'11)*. 1–10.
- D. M. Tullsen, S. J. Eggers, and H. M. Levy. 1995. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. 392–403.
- Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lu, and Rebecca L. Stamm. 1996. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, Vol. 24. ACM, 191–202.
- Chuanjun Zhang, Frank Vahid, and Walid Najjar. 2003. A highly configurable cache arch. for embedded systems. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA'03)*. ACM, 136–146.

Received May 2013; revised February 2014; accepted March 2014



Concurrency in Mobile Browser Engines

Călin Cașcaval, Pablo Montesinos Ortego, Behnam Robatmili, and Dario Suárez
Gracia, Qualcomm Research Silicon Valley

Web browsers are our main window into the wealth of information available on the Internet. All consumer computing platforms, including smartphones and tablets, rely on a browser to provide news, entertainment, and services. We use the term *Web apps* to refer to applications designed and implemented using Web technologies. Some Web apps require users to launch their Web browsers, while others appear to the user as native applications, even though they are just an API layer on top of a browser engine. Using Web technologies as the application back end is a convenient way of building portable applications across a variety of platforms.

However, this presents two main challenges. First, browsers must provide a smooth user experience—fast page load, satisfactory scroll and zoom performance, and uniform behavior regardless of the underlying hardware. The browsers' JavaScript engines thus must provide close-to-native application performance. The second challenge is that when running on mobile devices, browsers must adapt to the related energy and connectivity constraints.

As a result, browsers have been evolving to exploit the underlying hardware. Most current smartphones and tablets have systems on a chip (SoCs), with two to eight cores and powerful GPUs, and they rely on a plethora of techniques to maximize the performance/power ratio. Such techniques include

power and clock gating, dynamic voltage and frequency scaling, and offloading work to specialized cores. On the network side, Long-Term Evolution (LTE) offers 100 Mbps bandwidth, yet network latency continues to be high. Web browsers must exploit all available capabilities to address performance and energy challenges.

Here, we focus in particular on how Web browsers can use concurrency to improve per-tab (or per-page) processing. We use the Zoomm browser engine¹ and its MuscalietJS JavaScript engine² to illustrate how parallel processing improves performance and hides network latency for faster page loads.

EXPLOITING CONCURRENCY

Desktop browsers, such as WebKit (www.webkit.org) and Firefox (www.mozilla.org/firefox), typically exploit multiple cores by running each tab as a separate collection of processes and relying on the OS scheduler to place processes on different cores. The Zoomm browser architecture was designed with a different goal: take advantage of multicore processing for each browser tab. This is in line with typical mobile device usage, and it lets a more constrained platform meet its performance and energy goals.

A Parallel Browser Architecture

A Web browser has several major components: parsers (HTML, CSS,

JavaScript) that create the Document Object Model (DOM), a Cascading Style Sheets (CSS) engine to format and style the DOM, a layout engine to produce the image that will be displayed to the user, a rendering engine to display the page, and a JavaScript engine to enable interactivity and dynamic behavior.

Figure 1 shows the breakdown of execution time by component, excluding the network time. Our measurements, similar to other work,³ show that the network time is 30–50 percent of the total execution time. As the Web evolves, we're seeing remarkable changes in complexity and dynamic behavior. For example, in 2010, Leo Meyerovich and Rastislav Bodík measured WebKit execution and observed that JavaScript took approximately 5 percent of the execution time.⁴ One year later, the fraction of JavaScript execution increased to 30 percent, and for most webpages, it has since plateaued.

Even more significantly, we're observing a major trend to support application development using Web technologies such as HTML5, CSS, and JavaScript. Given this breakdown of computation, it is clear that to optimize the browser execution using concurrent processing, all major components must be addressed, because the gains from optimizing the components in isolation are bounded.

Our goal is to exploit concurrency at multiple levels: parallel algorithms

concurrency. These include asynchronous and deferred script processing directives in HTML, Web workers, and several efforts to express concurrency in JavaScript. In addition, the declarative nature of CSS makes it ripe for exploiting parallelism through concurrent implementations. □

ACKNOWLEDGMENTS

We thank Nayeem Islam and the Qualcomm Research Executive team for the opportunity to build the Zoomm and MuscalietJS engines. We thank Mehrdad Reshad, Michael Weber, Wayne Piekarzki, Seth Fowler, Vrajesh Bhavsar, Alex Shye, and Madhukar Kedlaya for their contributions.

REFERENCES

- C. Cascajal et al., "ZOOMM: A Parallel Web Browser Engine for Multicore Mobile Devices," *Proc. 18th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (POPP)*, 2013, pp. 271–280.
- B. Robatmili et al., "MuscalietJS: Rethinking Layered Dynamic Web Run-times," *Proc. 10th ACM SIGPLAN/SIGART Workshop on Parallel Execution of Sequential Programs on Multi-Core Architectures (PESPM)*, 2009, pp. 47–54.
- SIGOPS Int'l Conf. Virtual Execution Environments (VEE), 2014, pp. 77–88.
- Z. Wang et al., "Why Are Web Browsers Slow on Smartphones?" *Proc. ACM Int'l Workshop on Mobile Computing Systems and Applications*, 2011, pp. 91–96.
- I. A. Meyerovich and R. Bodik, "Fast and Parallel Webpage Layout," *Proc. Int'l Conf. World Wide Web*, 2010, pp. 711–720.
- M. Hölttä and D. Vogelheim, "New JavaScript Techniques for Rapid Page Loads," blog, 18 Mar. 2015; <http://blog.chromium.org/2015/03/new-javascript-techniques-for-rapid.html>.
- J.-D. Dalton, G. Seth, and L. Lafreniere, "Announcing Key Advances to Javascript Performance in Windows 10 Technical Preview," blog, Oct. 2014; <http://blogs.msdn.com/b/e/archive/2014/10/09/announcing-key-advances-to-javascript-performance-in-windows-10-technical-preview.aspx>.
- J. Ha et al., "A Concurrent Trace-Based Just-in-Time Compiler for Single-Threaded JavaScript," *Workshop on Parallel Execution of Sequential Programs on Multi-Core Architectures (PESPM)*, 2009, pp. 47–54.

Calin Cascajal is a senior director at Qualcomm Research Silicon Valley. Contact him at cascaval@qti.qualcomm.com.



Pablo Montesinos Ortego is a senior staff engineer/manager at Qualcomm Research Silicon Valley. Contact him at pablol@qti.qualcomm.com.



Behnam Robatmili is a staff research engineer at Qualcomm Research Silicon Valley. Contact him at behnamr@qti.qualcomm.com.



Dario Suárez Gracia is a staff engineer at Qualcomm Research Silicon Valley. Contact him at dgracia@qti.qualcomm.com.





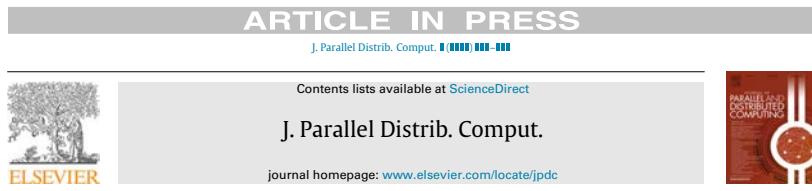
IEEE Pervasive COMPUTING
MOBILE AND UBIQUITOUS SYSTEMS



IEEE Pervasive Computing explores the many facets of pervasive and ubiquitous computing with research articles, case studies, product reviews, conference reports, departments covering wearable and mobile technologies, and much more.

Keep abreast of rapid technology change by subscribing today!

www.computer.org/pervasive



Reactive circuits: Dynamic construction of circuits for reactive traffic in homogeneous CMPs

Marta Ortín-Obón^{a,*}, Darío Suárez-Gracia^a, María Villarroya-Gaudó^a, Cruz Izquierdo^b, Víctor Viñals^a

^a Departamento de Informática e Ingeniería de Sistemas, University of Zaragoza, Spain

^b School of Computer Science, University of Adelaide, Australia

HIGHLIGHTS

- We implement on-demand reactive circuit reservations for NoCs.
- The use of circuits allows us to eliminate unnecessary coherence messages.
- We perform full-system simulation with realistic applications.
- For 64-cores, reactive circuits save 21% NoC energy, reduce area and improve performance.

ARTICLE INFO

Article history:
Received 4 September 2015
Received in revised form
17 February 2016
Accepted 1 April 2016
Available online xxxx

Keywords:
Chip multiprocessor
Interconnection network
Coherence protocol

ABSTRACT

Networks on Chip (NoCs) have a large impact on system performance, area, and energy. NoCs convey request and response messages among cores following the message patterns dictated by the cache banks. Such patterns do not only guarantee a coherent memory state, but also provide an opportunity for NoC optimization. Request messages can smartly reserve the resources to dynamically build a circuit for replies, thus reducing their network latency. Starting from this simple idea, which we denote Reactive Circuits, we evaluate several implementations of the mechanism: with and without sharing circuits between messages, performing timed reservations, and removing the implicit coherence messages. A careful implementation of this circuit reservation mechanism in a wormhole router achieves an average 20.8% reduction in network energy consumption, 5.8% smaller router area and a 4.8% system performance increase in a 64-core chip, compared with a conventional network.

© 2016 Elsevier Inc. All rights reserved.

1. Introduction

Chip multiprocessors (CMPs) are now a common design to improve performance by exploiting thread parallelism. They are built by replicating simple small cores that share a coherent memory space and are connected via an interconnection network. All the components of these chips must be carefully co-designed to achieve the desired performance while maintaining reasonable power consumption. We focus on optimizing the network-on-chip (NoC) by customizing it for the traffic it has to support, keeping in mind that the most relevant metric we must take care of is latency [32]. The traffic is generated by the coherence protocol to transport information among caches located in different nodes,

and mostly consists of data requests and replies. This paper introduces Reactive Circuits, a NoC design that takes advantage of this reactive nature of traffic. This novel design leverages the predictable network traffic behaviour to dynamically build virtual circuits for replies. We evaluate it in a homogeneous chip multiprocessor (CMP) with 16 and 64 cores connected by a mesh.¹

Analysing the coherence protocol in a standard wormhole 4-stage pipeline router, we note that the request–reply pattern

¹ A short version of this article was published in [27]. The new contributions are: more comprehensive state of the art review and detailed baseline system architecture description, new diagrams to illustrate the mechanism, new reactive circuit configurations (timed circuits, timed circuits with slack, timed circuits with slack and delay, and postponed timed circuits), ideal circuit reservation included for comparison, detailed analysis of circuits that can or cannot be built and used, network latency results, and performance results for each application for the best configuration run in 64 cores.

* Corresponding author.
E-mail address: ortin.marta@unizar.es (M. Ortín-Obón).
<http://dx.doi.org/10.1016/j.jpdc.2016.04.002>
0743-7315/© 2016 Elsevier Inc. All rights reserved.

Please cite this article in press as: M. Ortín-Obón, et al., Reactive circuits: Dynamic construction of circuits for reactive traffic in homogeneous CMPs, *J. Parallel Distrib. Comput.* (2016), <http://dx.doi.org/10.1016/j.jpdc.2016.04.002>

ARTICLE IN PRESS

12

M. Ortín-Obón et al. / J. Parallel Distrib. Comput. 1 (2016) 400–411



Cruz Izu received a Bachelor in Computer Science and Ph.D. in Computer Architecture from the University of the Basque Country. After a brief stint in industry, she joined the University of Adelaide in 1996. Her research interests include interconnection network design, modelling and simulation, parallel architectures and traffic characterization.



Víctor Viñals-Yéfara received the M.S. degree in Telecommunications, and the Ph.D. degree in Computer Science from the Universitat Politècnica de Catalunya (UPC) in 1982 and 1987, respectively. He was associate professor in the Facultat d'Informàtica de Barcelona (UPC) from 1983 to 1988. Currently, he is full professor in the Informàtica e Ingeniería de Sistemas Department at the University of Zaragoza, in Zaragoza (Spain). His research interests include computer microarchitectures, memory hierarchy, and parallel computer architecture. He is member of the ACM and the IEEE Computer Society. He also belongs to the Computer Architecture Group of the University of Zaragoza.

Please cite this article in press as: M. Ortín-Obón, et al., Reactive circuits: Dynamic construction of circuits for reactive traffic in homogeneous CMPs, *J. Parallel Distrib. Comput.* (2016), <http://dx.doi.org/10.1016/j.jpdc.2016.04.002>



Analysis of network-on-chip topologies for cost-efficient chip multiprocessors



Marta Ortín-Obón^{a,*}, Darío Suárez-Gracia^b, María Villarroya-Gaudó^a, Cruz Izu^c, Víctor Viñals-Yúfera^a

^a Departamento de Informática e Ingeniería de Sistemas, Universidad de Zaragoza, María de Luna 1, 50018, Zaragoza, Spain

^b Qualcomm Research Silicon Valley, California, US

^c Department of Computer Science, University of Adelaide, South Australia, 5005, Australia

ARTICLE INFO

Article history:

Received 8 May 2015

Revised 22 December 2015

Accepted 13 January 2016

Available online 1 February 2016

Keywords:

Interconnection networks

Chip multiprocessor

Topology

Mesh

Torus

Ring

ABSTRACT

As chip multiprocessors accommodate a growing number of cores, they demand interconnection networks that simultaneously provide low latency, high bandwidth, and low power. Our goal is to provide a comprehensive study of the interactions between the interconnection network and the memory hierarchy to enable a better co-design of both components. We explore the implications of the interconnect choice on overall performance by comparing the behaviour of three topologies (mesh, torus, and ring) and their concentrated versions. Simply choosing the concentrated mesh over the ring improves performance by over 40% in a 64-core chip.

The key strength of this work is the holistic analysis of the network-on-chip and the memory hierarchy. Experiments are carried out with a full-system simulator that carefully models the processors (single and multithreaded), memory hierarchy, and interconnection network, and executes realistic parallel and multiprogrammed workloads. We corroborate conclusions from several previous works: network diameter is critical, the concentrated mesh offers the best area-energy-delay trade-off, and traffic is very light and highly unbalanced. We also provide interesting insights about application-specific features that are hidden when studying only average results. We include a fairness analysis for multiprogrammed applications, and refute the idea of the memory controller placement greatly affecting performance.

© 2016 Elsevier B.V. All rights reserved.

1. Introduction

Nowadays, a single chip may contain multiple processors and a significant amount of memory. A popular trend consists of interconnecting several nodes, each of them with a core and one or more levels of private and/or shared cache memories. Nodes communicate through an interconnection network that allows them to exchange coherence messages and cache blocks, and has a major impact on overall performance, energy consumption, and area. We focus on general purpose CMPs, where both high-performance and low-power are required in equal shares.

Only a few works study the interconnect by modelling in detail the processors, memory hierarchy, and interconnection network. However, those analysis are often performed with synthetic traffic or application traces that do not entirely capture the behaviour

of a real execution [6,10,25,30]. This work simulates both parallel and multiprogrammed workloads with real applications, carefully modelling all the components above-mentioned. This allows us to study the effect of the interconnection network configuration on the whole system and the real interactions between the memory subsystem and the interconnect. We revisit the comparison of several topologies with our detailed simulation framework to update the results, validate or refute previous conclusions, and complete them with further analysis. We present an analysis of three topologies with varying degrees of complexity, performance, power, and area: mesh, torus, and ring. We model CMPs with 16 and 64 single-threaded cores, including a configuration with 16 4-threaded cores, and explore the effect of modifying the location and number of memory controllers. Our goal is to draw meaningful conclusions on the studied network configurations and study the details, pointing out the best choice from an integrated performance, area, and energy standpoint.

The rest of this document is organized as follows: Section 2 presents the related work; Section 3 describes the CMP architecture and the interconnection network configuration;

* Corresponding author. Tel.: +34 876555341.

E-mail addresses: ortin.marta@unizar.es, ortin.marta@gmail.com (M. Ortín-Obón), dario@unizar.es (D. Suárez-Gracia), mvg@unizar.es (M. Villarroya-Gaudó), cruz@cs.adelaide.edu.au (C. Izu), victor@unizar.es (V. Viñals-Yúfera).



María Villarroyo-Gaudó received the Ph.D. degree in electronic engineering from the Autonomous University of Barcelona, Spain, in 2005. She is Assistant Professor in the Computer Science and Systems Engineering Department, University of Zaragoza. Her research fields are on-chip networks and memory hierarchies.



Cruz Izquierdo received a BS degree in Computer Science and Ph.D. in Computer Architecture from the University of the Basque Country. After a brief stint in industry she joined the University of Adelaide in 1996. Her research interests include interconnection network design, modelling and simulation, parallel architectures and traffic characterisation.



Víctor Viñals-Yúfera received the MS degree in Telecommunications, and the Ph.D. degree in Computer Science from the Universitat Politècnica de Catalunya (UPC) in 1982 and 1987, respectively. He was associate professor in the Faculty of Computer Science from 1987 to 1990. Currently, he is full professor in the Informática e Ingeniería de Sistemas Department at the University of Zaragoza, in Zaragoza (Spain). His research interests include processor microarchitecture, memory hierarchy, and parallel computer architecture. He is member of the ACM and the IEEE Computer Society. He also belongs to the Computer Architecture Group of the University of Zaragoza.

Concertina: Squeezing in Cache Content to Operate at Near-Threshold Voltage

Alexandra Ferrerón, Student Member, IEEE, Darío Suárez-Gracia, Member, IEEE,
Jesús Alastruey-Benedé, Teresa Monreal-Arnal, and Pablo Ibáñez, Member, IEEE

Abstract—Scaling supply voltage to values near the threshold voltage allows a dramatic decrease in the power consumption of processors; however, the lower the voltage, the higher the sensitivity to process variation, and, hence, the lower the reliability. Large SRAM structures, like the last-level cache (LLC), are extremely vulnerable to process variation because they are aggressively sized to satisfy high density requirements. In this paper, we propose Concertina, an LLC designed to enable reliable operation at low voltages with conventional SRAM cells. Based on the observation that for many applications the LLC contains large amounts of null data, Concertina compresses cache blocks in order that they can be allocated to cache entries with faulty cells, enabling use of 100 percent of the LLC capacity. To distribute blocks among cache entries, Concertina implements a compression- and fault-aware insertion/replacement policy that reduces the LLC miss rate. Concertina reaches the performance of an ideal system implementing an LLC that does not suffer from parameter variation with a modest storage overhead. Specifically, performance degrades by less than 2 percent, even when using small SRAM cells, which implies over 90 percent of cache entries having defective cells, and this represents a notable improvement on previously proposed techniques.

Index Terms—Near-threshold voltage, SRAM variability, fault-tolerance, on-chip caches

1 INTRODUCTION

FROM tiny wearable devices to massive data centers, power density has become the *de facto* limiter for improving performance. Unfortunately, neither increasing transistor count nor reducing integration scale can fuel advances in performance anymore [37]; the number of active transistors has topped off, and voltage does not scale with technology.

The most straightforward way to reduce power density is to scale supply voltage (V_{dd}), but scaling is limited by the tight functionality margins of SRAM cells in last-level cache (LLC) transistors. Below a minimum operating voltage ($V_{dd_{min}}$), typically of the order of 0.7–1.0 V for regular six-transistor (6T) SRAM cells, process parameter variation in nanoscale technology has become so severe that SRAM cells no longer operate reliably.

Existing approaches to improving SRAM reliability can be classified into two groups: circuit- and architecture-based. Circuit-level techniques improve the reliability of the SRAM cell in low-voltage operation by increasing its size or by adding assist circuitry [24], [42]. These come at the cost

of an increase in power consumption and array area, which is not practical for large structures such as the on-chip LLC. At the architectural level, cache-tolerant designs rely on: (a) correcting defective bits through error correction codes (ECCs), increasing their complexity to enable them to detect and repair more defective bits per block [14]; (b) disabling faulty resources (words, lines, ways), marking a resource as defective whenever one faulty cell is found [27]; or (c) combining faulty resources to create functional ones, relying on the fact that several defective entries can be combined to store a block in a distributed manner [5], [39]. In all cases, a significant fraction of the memory structure is either disabled or sacrificed to store redundant information, degrading the overall effective cache capacity.

In this paper, we present Concertina, an efficient and fault-tolerant LLC that operates with unreliable SRAM cells at ultra-low voltages. Unlike previous architectural schemes, Concertina enables all the cache entries, even the faulty ones. Our key idea is to compress cache blocks, in order that they fit within the functional elements of faulty entries. Since not all cache blocks can be evenly compressed and not all faulty entries can store the same amount of information, it is not possible to use existing cache management policies based on the premise that a block can be stored in any entry of a set. To address this issue, we study different insertion/replacement policies that are aware of the nature of both the incoming block (degree of compression) and the corresponding cache set entries (number of defective cells).

Compression has been proposed in related literature as a promising technique to increase the effective on-chip cache capacity [2], [13], [34], [40], or to reduce the energy consumption of the cache [12], [22], but to the best of our knowledge, it has not been explored as a way to enhance

- A. Ferrerón, J. Alastruey-Benedé, and P. Ibáñez are with the Departamento de Informática e Ingeniería de Sistemas and the Instituto de Investigación en Ingeniería de Aragón (I3A), Universidad de Zaragoza, Spain, and HiPEAC. E-mail: {ferreron, jalastru, imarini}@unizar.es.
- D. Suárez-Gracia is with the Qualcomm Research Silicon Valley, Santa Clara, CA, E-mail: dgracia@qti.qualcomm.com.
- T. Monreal-Arnal is with the Departamento de Arquitectura de Computadores, Universitat Politècnica de Catalunya, Spain, and HiPEAC. E-mail: teresa@ac.upc.edu.

Manuscript received 15 Jan. 2015; revised 14 Aug. 2015; accepted 20 Aug. 2015. Date of publication 17 Sept. 2015; date of current version 10 Feb. 2016. Recommended for acceptance by C. Bolchini, S. Kundu, and S. Pontarelli. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TC.2015.2479585

0018-9340 © 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.



Alexandra Ferrerón (S'13) received the BS and MS degrees in computer engineering in 2010 and 2012 from the Universidad de Zaragoza, Spain, where she is currently working toward the PhD degree in systems engineering and computing. Her interests include high-performance low-power on-chip memory hierarchies, ultra-low and near-threshold voltage computing, and high performance computing. She is a member of the Instituto de Investigación en Ingeniería de Aragón (I3A), and the European HiPEAC NoE. She is a student member of the IEEE.



Dario Suárez-Gracia (S'08, M'12) received the PhD degree in computer engineering from the Universidad de Zaragoza, Spain, in 2011. Since 2012, he has been working at Qualcomm Research Silicon Valley on power aware parallel and heterogeneous computing for mobile devices. His research interests include parallel programming, heterogeneous computing, memory hierarchy design, networks-on-chip, and processor microarchitecture. He is also a member of the IEEE Computer Society and the Association for Computing Machinery. He is a member of the IEEE.



Jesús Alastruey-Benedé received the telecommunications engineering degree and the PhD degree in computer science from the Universidad de Zaragoza, Spain, in 1997 and 2009, respectively. He is a lecturer in the Departamento de Informática e Ingeniería de Sistemas (DIIS), Universidad de Zaragoza, Spain. His research interests include processor microarchitecture, memory hierarchy, and high performance computing (HPC) applications. He is a member of the Instituto de Investigación en Ingeniería de Aragón (I3A) and the European HiPEAC NoE.



Teresa Monreal-Arnal received the MS degree in mathematics and the PhD degree in computer science from the Universidad de Zaragoza, Spain, in 1991 and 2003, respectively. Until 2007, she was with the Departamento de Informática e Ingeniería de Sistemas (DIIS), Universidad de Zaragoza. She is currently an associate professor with the Departamento de Arquitectura de Computadores (DAC), Universitat Politècnica de Catalunya (UPC), Spain. Her research interests include processor microarchitecture, memory hierarchy, and parallel computer architecture. She collaborates actively with the Grupo de Arquitectura de Computadores (gaZ) from the Universidad de Zaragoza.



Pablo Ibáñez received the MS degree in computer science from the Universitat Politècnica de Catalunya in 1989, and the PhD degree in computer science from the Universidad de Zaragoza in 1998. He is an associate professor in the Departamento de Informática e Ingeniería de Sistemas (DIIS), Universidad de Zaragoza, Spain. His research interests include processor microarchitecture, memory hierarchy, parallel computer architecture, and high performance computing (HPC) applications. He is a member of the Instituto de Investigación en Ingeniería de Aragón (I3A) and the European HiPEAC NoE. He is a member of the IEEE.

▷ For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.



A fault-tolerant last level cache for CMPs operating at ultra-low voltage



Alexandra Ferrerón ^{a,c,1}, Jesús Alastruey-Benedé ^{a,c,*}, Darío Suárez Gracia ^{a,c},
Teresa Monreal Arnal ^{b,c}, Pablo Ibáñez Marín ^{a,c}, Víctor Viñals Yúfera ^{a,c}

^a Departamento de Informática e Ingeniería de Sistemas, Instituto de Investigación en Ingeniería de Aragón, Universidad de Zaragoza, Spain

^b Universitat Politècnica de Catalunya, BarcelonaTech, Spain

^c HiPEAC European Network of Excellence

HIGHLIGHTS

- Fault-Tolerant Last Level Cache for CMPs Operating at Ultra-Low Voltage.
- Mechanism that exploits redundancy and reuse to enhance block disabling performance.
- Fault-aware LLC management that maps critical blocks to operative cache entries.
- Detailed evaluation of block disabling techniques in a shared-memory coherent CMP.

ARTICLE INFO

Article history:
Received 19 December 2017
Received in revised form 23 July 2018
Accepted 22 October 2018
Available online 7 November 2018

Keywords:
Near-threshold voltage
SRAM reliability
Fault-tolerance
On-chip caches
Cache management

ABSTRACT

Voltage scaling to values near the threshold voltage is a promising technique to hold off the many-core power wall. However, as voltage decreases, some SRAM cells are unable to operate reliably and show a behavior consistent with a hard fault. Block disabling is a micro-architectural technique that allows low-voltage operation by deactivating faulty cache entries, at the expense of reducing the effective cache capacity. In the case of the last-level cache, this capacity reduction leads to an increase in off-chip memory accesses, diminishing the overall energy benefit of reducing the voltage supply. In this work, we exploit the reuse locality and the intrinsic redundancy of multi-level inclusive hierarchies to enhance the performance of block disabling with negligible cost. The proposed fault-aware last-level cache management policy maps critical blocks, those not present in private caches and with a higher probability of being reused, to active cache entries. Our evaluation shows that this fault-aware management results in up to 37.3% and 54.2% fewer misses per kilo instruction (MPKI) than block disabling for multiprogrammed and parallel workloads, respectively. This translates to performance enhancements of up to 13% and 34.6% for multiprogrammed and parallel workloads, respectively.

© 2018 Elsevier Inc. All rights reserved.

1. Introduction

For recent CMOS technologies, power density is the main performance limiting factor across most computing segments. Moore's law continues to hold, with a doubling of the number of transistors and integration density in each new process generation, but Dennard scaling no longer applies, and we are not able to keep a constant power density across technology generations. Power budgets prevent us from utilizing all the available transistors, leading to dark silicon [44].

For years, industry has relied on scaling the supply voltage (V_{dd}) to reduce power consumption, but this trend has dramatically slowed since the 90 nm generation because of leakage. Reducing operating voltages to values near the threshold voltage (V_{th}) would minimize leakage and switching power consumption. The resulting power reduction could be used to activate more chip resources and potentially achieve performance improvements [14].

Unfortunately, V_{dd} scaling is limited by the tight margins of the on-chip cache SRAM transistors. Excessive parameter variations in SRAM cells limit the voltage scaling of memory structures to a minimum voltage, $V_{dd_{min}}$, below which SRAM cells may not operate reliably. $V_{dd_{min}}$ usually determines the minimum voltage of the whole processor, and in current technologies is typically of the order of 0.7–1.0 V, when regular 6T SRAM cells are employed.

In the literature, various solutions have been proposed to enable reliable cache operation at low voltages. At the circuit level, the use

* Corresponding author at: Departamento de Informática e Ingeniería de Sistemas, Instituto de Investigación en Ingeniería de Aragón, Universidad de Zaragoza, Spain.
E-mail address: jalastru@unizar.es (J. Alastruey-Benedé).

¹ Now at Google.

- [47] S.-T. Zhou, S. Katariya, H. Chasemi, S. Draper, N.S. Kim, Minimizing total area of low-voltage SRAM arrays through joint optimization of cell size, redundancy, and ECC, in: IEEE Int. Conf. on Computer Design, 2010, pp. 112–117, <http://dx.doi.org/10.1109/ICCD.2010.5647605>.



Alexandra Ferrerón received the M.S. and Ph.D. degrees in computer science engineering from the Universidad de Zaragoza, Spain, in 2013 and 2016, respectively. Her research interests include high-performance low-power on-chip memory hierarchies, ultra-low and near-threshold voltage computing, and High Performance Computing. She currently works as Site Reliability Engineer for BigQuery (Google Cloud Platform) at Google Switzerland.



Jesús Alastruey-Benedé received the Telecommunications Engineering degree and the Ph.D. degree in Computer Science from the Universidad de Zaragoza, Spain, in 1997 and 2008, respectively. He is a Lecturer in the Departamento de Informática e Ingeniería de Sistemas (DIIS), Universidad de Zaragoza, Spain. His research interests include processor microarchitecture, memory hierarchy, and High Performance Computing (HPC) applications. He is a member of the Instituto de Investigación en Ingeniería de Aragón (I3A) and the European HiPEAC NoE.



Dario Suárez Gracia (S'08, M'12) received the Ph.D. degree in computer engineering from the Universidad de Zaragoza, Spain, in 2012. From 2012 to 2014 he worked at Qualcomm Research Silicon Valley on power aware parallel and heterogeneous computing for mobile devices. Currently, he is an interim associate professor at the Universidad de Zaragoza in Spain. His research interests include parallel programming, heterogeneous computing, memory hierarchy design, networks-on-chip, and accelerators for computer vision applications. He is also a member of the Instituto de Investigación en Ingeniería de Aragón (I3A), the IEEE, the IEEE Computer Society, and the Association for Computing Machinery.



Teresa Monreal Arnal received the M.S. degree in Mathematics and the Ph.D. degree in Computer Science from the Universidad de Zaragoza, Spain, in 1991 and 2003, respectively. Until 2007, she was with the Departamento de Informática e Ingeniería de Sistemas (DIIS) at the Universidad de Zaragoza, Spain. Currently, she is an Associate Professor with the Computer Architecture Department (DAC) at the Universitat Politècnica de Catalunya (UPC), Spain. Her research interests include processor microarchitecture, memory hierarchy, and parallel computer architecture. She collaborates actively with the Grupo de Arquitectura de Computadores from the Universidad de Zaragoza (gaZ).



Pablo Ibáñez Marín received the M.S. degree in computer science from the Universitat Politècnica de Catalunya in 1989, and the Ph.D. degree in computer science from the Universidad de Zaragoza in 1998. He is an Associate Professor in the Departamento de Informática e Ingeniería de Sistemas (DIIS) at the Universidad de Zaragoza, Spain. His research interests include processor microarchitecture, memory hierarchy, parallel computer architecture, and High Performance Computing (HPC) applications. He is a member of the Instituto de Investigación en Ingeniería de Aragón (I3A) and the European HiPEAC NoE.



Víctor Vilàs Yáñez received the M.S. degree in Telecommunications and the Ph.D. degree in Computer Science from the Universitat Politècnica de Catalunya (UPC) in 1982 and 1987, respectively. He was associate professor in the Facultat d'Informàtica de Barcelona from 1983 to 1988. Currently, he is full professor in the Departamento de Informática e Ingeniería de Sistemas at the Universidad de Zaragoza (Spain). His research interests include processor microarchitecture, memory hierarchy, and parallel computer architecture. He is member of the ACM, the IEEE Computer Society, and HiPEAC. He also belongs to the Computer Architecture Group and the I3A Institute of the University of Zaragoza.



Exploring heterogeneous scheduling for edge computing with CPU and FPGA MPSoCs

Andrés Rodríguez^a, Angeles Navarro^a, Rafael Asenjo^{a*}, Francisco Corbera^a, Rubén Gran^b, Dario Suárez^c, Jose Nunez-Yanez^c

^aDepartment of Computer Architecture, Universidad de Málaga, Spain

^bComputer Architecture Group, Universidad de Zaragoza, Spain

^cDepartment of Electrical & Electronic Engineering, University of Bristol, UK

A B S T R A C T

This paper presents a framework targeted to low-cost and low-power heterogeneous MultiProcessors that exploits FPGAs and multicore CPUs, with the overarching goal of providing developers with a productive programming model and runtime support to fully use all the processing resources available. FPGA productivity is achieved using a high-level programming model based on OpenCL, the standard for cross-platform parallel heterogeneous programming. In this work, we focus on the parallel pattern, and as part of the runtime support for this pattern, we leverage a new scheduler that strives to minimize the number of iterations per joule by dynamically and adaptively partitioning the iteration space between the multicore and the accelerator when working simultaneously. A total of 7 benchmarks are ported and optimized for a low-cost DEI board. The results show that the heterogeneous solution can improve performance up to 2.9× and increases energy efficiency up to 2.7× compared to the traditional approach of keeping all the CPU cores idle while the accelerator computes the workload. Our results also demonstrate two interesting insights: first, an adaptive scheduler able to find at runtime the right chunk size for each type of application and device configuration is an essential component for these kinds of heterogeneous platforms, and second, device configurations that provide higher throughput do not always achieve better energy efficiency when only the running power (excluding the idle power component) is considered.

1. Introduction and motivation

The trend towards embedding connected sensors everywhere pushes for a computational model where part of the compute is done at the edge instead of on cloud. Therefore, we see more powerful devices at the edge that also have to drain a limited amount of power. For example the analysis in [1] highlights that edge computing is well equipped to handle privacy and connectivity issues but some advantages such as low latency are only realizable if enough local computation power is provided. A possible solution to these performance requirements is low-power heterogeneous systems suitable for edge computing.

In heterogeneous architectures, specialized hardware units accelerate complex tasks. A good example of this trend is the introduction of GPUs (Graphics Processing Units) for general purpose computing combined with multicore CPUs. Recent research projects in the context of exascale have proposed new architectures based on multicore CPUs and integrated reconfigurable resources (e.g. FPGAs) that target to significantly improve the performance over power dissipation ratio [2,3]. Moreover, processor vendors have integrated general-purpose multicore CPUs and FPGA into the same chip for data center acceleration or specialized embedded applications [4].

In this research work we target low-cost and low-power heterogeneous embedded systems that tightly couple an ARM CPU multicore and an on-chip FPGA, usually called heterogeneous MultiProcessor System-on-Chip or MPSoC.

ARM processors are receiving significant attention and have gained a lot of traction in the market as low-power alternatives to the X86 architecture. The selected low-cost platform combines a dual-core 32-bit ARM processor and a small low-power FPGA fabric. This makes it suitable to work at the edge near data-collecting sensors instead of a classical high-power and high-performance computing set-up. In this arrangement, the near sensor device runs the same algorithms as the high-end server-class device but over coarse data, and it identifies events that contain anomalies that are then sent to the server over the network for further processing at higher resolution. For example, unexpected temperature variations in the surface of a device that exceed a threshold and require further processing to identify a possible fault condition. For this reason, in this work, we also select some benchmarks from the high performance computing world to run in the near-edge heterogeneous platform. An alternative to use accelerators tightly coupled to the ARM processor is to build SoCs with processors with different levels of complexity, power and performance based on a single instruction set

* Corresponding author.

E-mail addresses: andres@ac.uma.es (A. Rodríguez), angeles@ac.uma.es (A. Navarro), asenjo@ac.uma.es (R. Asenjo), corbera@ac.uma.es (F. Corbera), rgran@unizar.es (R. Gran), dario@unizar.es (D. Suárez), J.L.Nunez-Yanez@bristol.ac.uk (J. Nunez-Yanez).

<https://doi.org/10.1016/j.sysarc.2019.06.006>
Received 26 February 2019; Received in revised form 22 May 2019; Accepted 11 June 2019
Available online 13 June 2019
1383-7621/© 2019 Elsevier B.V. All rights reserved.

- [27] J. Gómez-Luna, I. El Hajji, V. Chang Li-Wen García-Flores, S. García de Gonzalo, T. Jablin, A.J. Pena, W.-m. Hwu, Chai: Collaborative heterogeneous applications for integrated-architectures. *Intl. Symp. on Performance Analysis of Systems and Software (ISPASS)*, 2017.
- [28] I. Corporation, Monte carlo pricing of asian options on FPGAs using OpenCL, 2014, (<https://www.altera.com/support/support-resources/design-examples/design-software/asic-blaze.html>).
- [29] W. Wang, S. Ghosh, S. Veerasamy, K. Sankaranarayanan, K. Skadron, M.R. Stan, Hotspot: a compact thermal modeling methodology for early-stage VLSI design, *IEEE Trans. Very Large Scale Integr. Syst.* 14 (5) (2006).
- [30] Z. Wang, B. He, W. Zhang, S. Jiang, A performance analysis framework for optimizing OpenCL applications on FPGAs, in: *Intl. Symp. on High Performance Computer Architecture (HPCA)*, 2016, pp. 114–125.
- [31] K. Krommydas, R. Sasanka, W. c. Feng, Bridging the FPGA programmability-portability gap via automatic OpenCL code generation and tuning, in: *Intl. Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, 2016, pp. 213–216.
- [32] F.D. Iglesias, L.M. Jara, J.I. Gómez-Pérez, L. Pitied, M. Prieto-Matías, A power measurement environment for PCIe accelerators, *Comput. Sci. Res. Dev.* 30 (2) (2015) 115–124.
- [33] S. Barrachina, M. Barreda, S. Catalán, M.F. Dolz, G. Fabregat, R. Mayo, E. Quintana-Ortí, An integrated framework for power-performance analysis of parallel scientific workloads, *Energy* (2013) 114–119.
- [34] K. Czechowski, V.W. Lee, E. Grochowski, R. Ronen, R. Singhali, R. Viduc, P. Dubey, Improving the energy efficiency of big cores, *Intl. Symp. on Computer Architecture, ISCA’14*, 2014.



Andrés Rodríguez obtained a Ph.D. in Computer Science Engineering from the University of Málaga, Spain, in 2000. From 1996 to 2002, he was an Assistant Professor in the Computer Architecture Department at University of Málaga, being an Associate Professor from 2001 until 2003. He lectures on operating system design, mobile devices architectures and IoT. His research interests are in parallel programming models and tools for heterogeneous architectures.



Angeles Navarro obtained a Ph.D. in Computer Science from the University of Málaga, Spain, in 2000. She was an Associate Professor in the Computer Architecture Department at University of Málaga from 2001 until 2019 and Full Professor since then. Currently she is the Vice Dean for Postgraduate Studies in the Computer Science School at the University of Málaga. She was Visiting Scholar in the University of Illinois at Urbana-Champaign (UIUC) in 1996 and 1997, and Visiting Research Associate in the same University in 1998. She was a Research Visitor at IBM T.J. Watson in 2008 and at Cray Inc. in 2011. She has contributed to the Cray’s Chapel runtime development since 2011. She has served as a program committee member for several High Performance Computing related conferences as PPoPP, SC, ICS, PACT, IPDPS, ICPP, EuroPar, ISPA. She is the co-leader of the Parallel Programming Models and Compilers group at the University of Málaga. Her research interests are in programming models for heterogeneous systems, analytical modeling, compiler and runtime optimizations.



Rafael Asenjo obtained a Ph.D. in Telecommunication Engineering from the University of Málaga, Spain in 1997. From 1994 to 2001, he was an Assistant Professor in the Computer Architecture Department at University of Málaga, being an Associate Professor from 2001 until 2017 and Full Professor since then. He was Visiting Scholar in the University of Illinois at Urbana-Champaign (UIUC) in 1996 and 1997, and Visiting Research Associate in the same University in 1998. He also was Research Visitor at IBM T.J. Watson in 2008 and at Cray Inc. in 2011. He collaborated on the IBM XL-UPC compiler in 2008 and contributed to the Cray’s Chapel runtime development. He has served as General Chair, Program Chair and Session Chair Committees member as well as a Programme Committee member for several HPC related conferences among others PPoPP, SC, PACT, IPDPS, HPCA, EuroPar and SBAC-PAD. He is the co-leader of the Parallel Programming Models and Compilers group at the University of Málaga. His research interests are in parallel programming models and tools for heterogeneous architectures. He is ACM Member.



Francisco Corbera received the BS and MS degrees in computer science in 1994, from the University of Granada, Spain, and the Ph.D. degree in computer science in 2001, from the University of Málaga, Spain. From 1996 to 2001, he was an associate professor in the Computer Architecture Department at University of Málaga. He has been an associate professor in the same department since 2002. He lectures on computer technology and architecture. His research interests are in parallelizing compilers and multiprocessor architectures.



Rubén Grau graduated in Computer Science from the University of Zaragoza (Spain) and held his Ph.D. in 2010 from the University of Zaragoza (Spain). Since 2010 he is an assistant professor in the Informática e Ingeniería de Sistemas Department in the University of Zaragoza. He is member of the Computer Architecture group (g32) of the University of Zaragoza and his research interests are worst-case in hard real-time systems, microarchitecture, optimizing compilers for accelerators and load balancing on heterogeneous systems.



Dario Suárez (S’08, M’12) received the Ph.D. degree in computer engineering from the Universidad de Zaragoza, Spain, in 2011. From 2012 to 2015, he was working at Qualcomm Research Silicon Valley on power aware parallel and heterogeneous computing for mobile devices. Currently, he is an interim associate professor at the Departamento de Informática e Ingeniería de Sistemas, Universidad de Zaragoza, Spain. His research interests include parallel and heterogeneous computing, memory hierarchy design, networks-on-chip, and processor microarchitecture. He is also a member of the Instituto de Investigación en Ingeniería de Aragón and the European Network on High Performance Embedded Architecture and Compilation (HiPEAC).



José Núñez-Yáñez is a Reader (associate professor) in adaptive and energy efficient computing at the University of Bristol and member of the microelectronics group. He holds a Ph.D. in hardware-based parallel data compression from the University of Loughborough, UK, with three patents awarded on the topic of high-speed parallel data compression. His main area of expertise is in the design of reconfigurable architectures for signal processing with a focus on run-time adaptation, parallelism and energy-efficiency. In 2006–2007 he was a Marie Curie Research Fellow at STMicroelectronics, Italy working on the automatic design of accelerators for video processing. In 2011 he was a Royal Society research fellow at ARM Ltd, Cambridge, UK working on high-level modelling of the energy consumption of heterogeneous many-core systems. He is the PI in several industrial and UK research council projects obtaining best papers awards as part of this work. He is currently involved as a co-investigator in the EU TEAMPLAY and Energy-ICT FP7 projects.

Simultaneous multiprocessing in a software-defined heterogeneous FPGA

Jose Nunez-Yanez¹ · Sam Amiri¹ · Mohammad Hosseiniabady¹ ·
Andrés Rodríguez² · Rafael Asenjo² · Angeles Navarro² ·
Dario Suarez³ · Ruben Gran³

Published online: 16 April 2018
© The Author(s) 2018, corrected publication May 2018

Abstract Heterogeneous chips that combine CPUs and FPGAs can distribute processing so that the algorithm tasks are mapped onto the most suitable processing element. New software-defined high-level design environments for these chips use general purpose languages such as C++ and OpenCL for hardware and interface generation without the need for register transfer language expertise. These advances in hardware compilers have resulted in significant increases in FPGA design productivity. In this paper, we investigate how to enhance an existing software-defined framework

✉ Sam Amiri
ma17215@bristol.ac.uk
Jose Nunez-Yanez
j.l.nunez-yanez@bristol.ac.uk
Mohammad Hosseiniabady
m.hosseiniabady@bristol.ac.uk
Andrés Rodríguez
andres@ac.uma.es
Rafael Asenjo
asenjo@ac.uma.es
Angeles Navarro
angeles@ac.uma.es
Dario Suarez
dario@unizar.es
Ruben Gran
rgran@unizar.es

¹ University of Bristol, Bristol, UK

² Universidad de Málaga, Málaga, Spain

³ Universidad de Zaragoza, Zaragoza, Spain

7. Auerbach J et al (2012) A compiler and runtime for heterogeneous computing. DAC '12, pp 271–276
8. Pandit P, Govindarajan R (2014) Fluidic kernels: cooperative execution of OpenCL programs on multiple heterogeneous devices. CGO '14, pp 273:273–283
9. Dolbeau R, Bodin F, de Verdier GC (2013) One OpenCL to rule them all? In: MuCoCoS '13, pp 1–6
10. Vilches A et al (2016) Mapping streaming applications on commodity multi-CPU and GPU-on-chip processors. IEEE Trans Parallel Distrib Syst 27(4):1099–1115
11. Meng P, Jacobsen M, Kastner R (2012) FPGA-GPU-CPU heterogeneous architecture for real-time cardiac physiological optical mapping. In: FPT '12, pp 37–42
12. Prongnuch S, Wiantong T (2014) Heterogeneous computing platform for data processing. In: ISPACS '16, pp 1–4
13. Korinth J, de la Chevallerie D, Koch A (2015) An open-source tool flow for the composition of reconfigurable hardware thread pool architectures. In: FCCM '15, pp 195–198
14. Tsai KH, Luk W (2010) Axel: A heterogeneous cluster with FPGAs and GPUs. FPGA '10, pp 115–124
15. SDSoC environment user guide. www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/tg1027-sdsooc-user-guide.pdf. Accessed 2018 Jan 22
16. Aldinucci M, Danielutto M, Kilpatrick P, Torquati M (2017) Fastflow: high-level and efficient streaming on multicore. Wiley, Hoboken, pp 261–280
17. Navarro A, Vilches A, Corbera F, Asenjo R (2014) Strategies for maximizing utilization on multi-CPU and multi-GPU heterogeneous architectures. J Supercomput 70:756–771
18. Luk CK, Hong S, Kim H (2009) Qilin: exploring parallelism on heterogeneous multiprocessors with adaptive mapping. In: Proceedings of the Micro, pp 45–55
19. Wang Z, Zheng L, Chen Q, Guo M (2014) CPU+GPU scheduling with asymptotic profiling. Parallel Comput 2:107–115
20. Dagum L, Menon R (1998) OpenMP: an industry standard API for shared-memory programming. IEEE Comput Sci Eng 5(1):46–55
21. Rudolph DC, Polychronopoulos CD (1989) An efficient message-passing scheduler based on guided self scheduling. ICS '89, pp 50–61
22. Vilches A et al (2015) Adaptive partitioning for irregular applications on heterogeneous CPU-GPU chips. Procedia Comput Sci 51:140–149
23. Jia Q, Zhou H (2016) Tuning Stencil codes in OpenCL for FPGAs. In: ICCD '16, pp 249–256



Parallel multiprocessing and scheduling on the heterogeneous Xeon+FPGA platform

Andrés Rodríguez¹ · Angeles Navarro¹ · Rafael Asenjo¹ · Francisco Corbera¹ ·
Rubén Gran² · Darío Suárez² · José Núñez-Yáñez³

© Springer Science+Business Media, LLC, part of Springer Nature 2019

Abstract

Heterogeneous computing that exploits simultaneous co-processing with different device types has been shown to be effective at both increasing performance and reducing energy consumption. In this paper, we extend a scheduling framework encapsulated in a high-level C++ template and previously developed for heterogeneous chips comprising CPU and GPU cores, to new high-performance platforms for the data center, which include a cache coherent FPGA fabric and many-core CPU resources. Our goal is to evaluate the suitability of our framework with these new FPGA-based platforms, identifying performance benefits and limitations. We target the state-of-the-art HARP processor that includes 14 high-end Xeon classes tightly coupled to a FPGA device located in the same package. We select eight benchmarks from the high-performance computing domain that have been ported and optimized for this heterogeneous platform. The results show that a dynamic and adaptive scheduler that exploits simultaneous processing among the devices can improve performance up to a factor of 8 × compared to the best alternative solutions that only use the CPU cores or the FPGA fabric. Moreover, our proposal achieves up to 15% and 37% of improvement compared to the best heterogeneous solutions found with a dynamic and static schedulers, respectively.

Keywords Heterogeneous architecture · FPGA · Parallel_for template · Heterogeneous scheduling · Hybrid algorithm · Adaptive chunk size

1 Motivation

To bring up to date Herb Sutter's quote “the free lunch is over,” we would say that lunch is becoming prohibitively expensive. Now it is not the multicore architecture the one being democratized, but the heterogeneous on-chip processor is the one

✉ Rafael Asenjo
asenjo@ac.uma.es

Extended author information available on the last page of the article

21. Prabhakar R, Koeplinger D, Brown KJ, Lee H, De Sa C, Kozyrakis C, Olukotun K (2016) Generating configurable hardware from parallel patterns. SIGOPS Oper Syst Rev 50(2):651–665. <https://doi.org/10.1145/2954680.2872415>
22. Remírez Garzáñ MJ, Asenjo R, Navarro AG (2018) Exploiting social network graph characteristics for efficient BFS on heterogeneous chips. J Parallel Distrib Comput 120:282–294. <https://doi.org/10.1016/j.jpdc.2017.11.003>
23. Rudolph D, Polychronopoulos C (1989) An efficient message-passing scheduler based on guided self scheduling. In: Proceedings of the 3rd International Conference on Supercomputing. ICS'89 1.pdf. Accessed 17 June 2019
24. Sun Y, Gong X, Ziabari AK, Yu L, Li X, Mukherjee S, McCardwell C, Villegas A, Kaeli D (2016) Hetero-mark, a benchmark suite for CPU-GPU collaborative computing. In: Int'l Symp. on Workload Characterization (IISWC), pp 1–10
25. Umuoglu Y, Morrison D, Jahre M (2015) Hybrid breadth-first search on a single-chip FPGA-CPU heterogeneous platform. Int Conf Field Programm Log Appl. <https://doi.org/10.1109/FPL.2015.7293939>
26. Vilches A, Asenjo R, Navarro A, Corbera F, Gran R, Garzaran MJ (2015) Adaptive partitioning for irregular applications on heterogeneous CPU-GPU chips. Procedia Comput Sci 51:140–149
27. Wang Z, He B, Zhang W, Jiang S (2016) A performance analysis framework for optimizing OpenCL applications on FPGAs. In: 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp 114–125
28. Windh S, Ma X, Halestad RJ, Budhkar P, Luna Z, Hussaini O, Najjar WA (2015) High-level language tools for reconfigurable computing. Proc IEEE 103(3):390–408. <https://doi.org/10.1109/JPROC.2015.2399275>
29. Zhou S, Prasanna VK (2017) Accelerating graph analytics on CPU-FPGA heterogeneous platform. In: 2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), pp 137–144. <https://doi.org/10.1109/SBAC-PAD.2017.25>

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Affiliations

Andrés Rodríguez¹ · Angeles Navarro¹ · Rafael Asenjo¹ · Francisco Corbera¹ · Rubén Gran² · Darío Suárez² · Jose Núñez-Yáñez³

Andrés Rodríguez
andres@ac.uma.es

Angeles Navarro
angeles@ac.uma.es

Francisco Corbera
corbera@ac.uma.es

Rubén Gran
rgran@unizar.es

Darío Suárez
dario@unizar.es

José Núñez-Yáñez
J.L.Nunez-Yanez@bristol.ac.uk

¹ Department of Computer Architecture, Universidad de Málaga, Andalucía Tech, Málaga, Spain

² Computer Architecture Group, Universidad de Zaragoza, Zaragoza, Spain

³ Department of Electrical and Electronic Engineering, University of Bristol, Bristol, UK



Cooperative CPU, GPU, and FPGA heterogeneous execution with EngineCL

Mara Angélica Díaz-Vila Guzmán¹ · Raúl Nozal² · Rubén Gran Tejero¹ ·
Mara Villarroya-Gaudí¹ · Darío Suárez Gracia¹ · José Luis Bosque²

Published online: 7 February 2019
© Springer Science+Business Media, LLC, part of Springer Nature 2019

Abstract

Heterogeneous systems are the core architecture of most of the high-performance computing nodes, due to their excellent performance and energy efficiency. However, a key challenge that remains is programmability, specifically, releasing the programmer from the burden of managing data and devices with different architectures. To this end, we extend EngineCL to support FPGA devices. Based on OpenCL, EngineCL is a high-level framework providing load balancing among devices. Our proposal fully integrates FPGAs into the framework, enabling effective cooperation between CPU, GPU, and FPGA. With command overlapping and judicious data management, our work improves performance by up to 96% compared with single-device execution and delivers energy-delay gains of up to 37%. In addition, adopting FPGAs does not require programmers to make big changes in their applications because the extensions do not modify the user-facing interface of EngineCL.

Keywords Heterogeneous scheduling · FPGA · Load balancing · OpenCL

✉ Mara Angélica Díaz-Vila Guzmán
angelicadg@unizar.es
Raúl Nozal
raul.nozal@unican.es
Rubén Gran Tejero
rgran@unizar.es
Mara Villarroya-Gaudí
maria.villarroya@unizar.es
Darío Suárez Gracia
dario@unizar.es
José Luis Bosque
bosquejl@unican.es

¹ Universidad de Zaragoza, Zaragoza, Spain

² Universidad de Cantabria, Santander, Spain

11. Katranovet A et al (2016) Intel threading building block (TBB) & w graph as a software infrastructure layer for OpenCL-based computations. In: ACM IWOCCL, pp 9–13
12. Koch D et al (eds) (2016) FPGAs for software programmers. Springer, Cham
13. Lee J et al (2016) Orchestrating multiple data-parallel kernels on multiple devices. In: International Conference on Parallel Architectures and Compilation Techniques, pp 355–66
14. Luk C-K et al (2009) Qflin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. IEEE/ACM Micro 42: p 45
15. Mittal S et al (2015) A survey of CPU/GPU heterogeneous computing techniques. ACM Comput Surv 47(4):185
16. Momeni A et al (2016) Hardware thread reordering to boost OpenCL throughput on FPGAs. In: ICCD, pp 257–64
17. Muslim FB et al (2017) Efficient FPGA implementation of OpenCL high-performance computing applications via high-level synthesis. IEEE Access 5:2747–762
18. Nane R et al (2016) A survey and evaluation of FPGA high-level synthesis tools. IEEE Trans Comput Aided Des Integr Circuits Syst 35(10):1591–604
19. Nozal R et al (2018) EngineeCL: usability and performance in heterogeneous computing. arXiv: abs/1805.02755
20. Nozal R et al (2018) Load balancing in a heterogeneous world: Cpu-Xeon Phi co-execution of data-parallel kernels. J Supercomput 73(1):330–42
21. Nunez-Yanez J (2018) Simultaneous multiprocessing in a software-defined heterogeneous FPGA. J Supercomput
22. Pandit P et al (2014) Fluidic kernels: cooperative execution of OpenCL programs on multiple heterogeneous devices. In: Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization
23. Párez B (2017) Energy efficiency of load balancing for data-parallel applications in heterogeneous systems. J Supercomput 73(1):330–42
24. Párez B et al (2016) Simplifying programming and load balancing of data parallel applications on heterogeneous systems. In: GPGPU. ACM, New York, pp 42–51
25. Qualcomm Snapdragon Heterogeneous Compute SDK (2018). <https://developer.qualcomm.com/software/heterogeneous-compute-sdk>
26. Rehingirgi SK et al (2015) Trigeneous platforms for energy efficient computing of HPC applications. In: International Conference on High Performance Computing Trigeneous. IEEE
27. SDSoC Environment User Guide. www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug1027-sdsc-user-guide.pdf
28. Tsoi KH et al (2010) Axel: a heterogeneous cluster with FPGAs and GPUs. In: ACM/SIGDA FPGA, ACM, New York, pp 115–24
29. Vilches A et al (2015) Adaptive partitioning for irregular applications on heterogeneous CPU/GPU chips. Procedia Comput Sci ICCS 51:140–49
30. Wang Z et al (2016) A performance analysis framework for optimizing OpenCL applications on FPGAs. In: Proceedings of HPCA, pp 114–25
31. Zhou S et al (2017) Accelerating graph analytics on CPU-FPGA heterogeneous platform. In: SBAC-PAD, pp 137–44
32. Zohouri HR et al (2016) Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs. In: SC. IEEE Press, Piscataway, pp 35:1–5:12

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

An Aging-Aware GPU Register File Design Based on Data Redundancy

Alejandro Valero[✉], Francisco Candel[✉], Darío Suárez-Gracia[✉], Member, IEEE,
Salvador Petit[✉], Member, IEEE, and Julio Sahuquillo[✉], Member, IEEE

Abstract—Nowadays, GPUs sit at the forefront of high-performance computing thanks to their massive computational capabilities. Internally, thousands of functional units, architected to be fed by large register files, fuel such a performance. At deep nanometer technologies, the SRAM memory cells that implement GPU register files are very sensitive to the Negative Bias Temperature Instability (NBTI) effect. NBTI ages cell transistors by degrading their threshold voltage V_{th} over the lifetime of the GPU. This degradation, which manifests when a cell keeps the same logic value for a relatively long period of time, compromises the cell read stability and increases the transistor switching delay, which can lead to wrong read values and eventually exceed the processor cycle time, respectively, so resulting in faulty operation. This work proposes architectural mechanisms leveraging the redundancy of the data stored in GPU register files to attack NBTI aging. The proposed mechanisms are based on data compression, power gating, and register address rotation techniques. All these mechanisms working together balance the distribution of logic values stored in the cells along the execution time, reducing both the overall V_{th} degradation and the increase in the transistor switching delays. Experimental results show that a conventional GPU register file suffers the worst case for NBTI, since a significant fraction of the cells maintain the same logic value during the entire application execution (i.e., a 100 percent '0' and '1' duty cycle distributions). On average, the proposal reduces these distributions by 58 and 68 percent, respectively, which translates into V_{th} degradation savings by 54 and 62 percent, respectively.

Index Terms—Data compression, duty cycle, GPU architectures, NBTI, register files, threshold voltage degradation

1 INTRODUCTION

THE role GPUs play in high-performance computing is growing in importance due to their excellent performance per watt compared to conventional processors [1]. For instance, the most energy-efficient supercomputers in the world, ranked in the Green500 list [2], include GPU devices.

GPUs are designed for improving system throughput, and their design is aimed at exploiting Thread Level Parallelism (TLP) by supporting the concurrent execution of a vast number of threads. The number of threads that a GPU can simultaneously execute exceeds, in several orders of magnitude, the number of hardware contexts supported by advanced processors like the IBM Power9 [3] or the Intel Knights Landing [4]. This feature is especially important for the execution of parallel scientific applications that rely on a high number of threads.

In this regard, GPUs have dramatically evolved during the last years to support massive numbers of threads, which

implies that they must incorporate huge register files to feed the computation performed by these threads. For example, the NVIDIA Tesla P100 (Pascal GP100) GPU features a 14 MB register file, 3.5 times larger than its shared L2 cache (4 MB) and several orders of magnitude bigger than typical CPU register files or CPU private L1 and L2 caches.

On the other hand, technology advances are allowing the semiconductor industry to implement fabrication nodes whose size is so small that process variations threaten system reliability. Process variations make transistors less reliable in low-power modes and intensify transistor aging phenomena, which affects modern computing devices, especially those that implement a large number of transistors, such as GPUs.

This work focuses on attacking aging in those transistors implementing the SRAM memory cells of GPU register files. In particular, the effect that most accelerates aging in SRAM cells is known as Negative Bias Temperature Instability (NBTI). NBTI degrades the threshold voltage V_{th} of the PMOS transistors that are on (i.e., their gate is connected to a logic '0'). In an SRAM cell, this happens when a logic value is stored for a relatively long period of time, known as duty cycle. In turn, the V_{th} degradation, or simply dV_{th} , compromises the cell read stability, which is measured as Static Noise Margin (SNM), and can lead to wrong read values. In addition, the dV_{th} slows down the transistor switching delay T_s , and, since this effect can affect several transistors along the critical path of a digital circuit, it can cause operation faults if the critical path delay exceeds the clock cycle. Overall, mitigating the dV_{th} results in a reduction of the SNM degradation and T_s . In fact, both SNM and T_s closely follow

• A. Valero and D. Suárez-Gracia are with the Departamento de Informática e Ingeniería de Sistemas, Instituto Universitario de Ingeniería de Aragón, Universidad de Zaragoza, Zaragoza 50009, Spain. E-mail: {alvalero, dario}@unizar.es.

• F. Candel, S. Petit, and J. Sahuquillo are with the Department of Computer Engineering, Universidad Politécnica de Valencia, Valencia 46022, Spain. E-mail: {frcanma@inf.upv.es, lspetit, jsahuqui}@dsic.upv.es.

Manuscript received 2 Dec. 2017; revised 18 Apr. 2018; accepted 11 June 2018. Date of publication 24 June 2018; date of current version 19 Dec. 2018. (Corresponding author: Alejandro Valero.)

Recommended for acceptance by S. Huang.

For information on obtaining reprints of this article, please send e-mail to: reprints@iee.org, and reference the Digital Object Identifier below.

Digital Object Identifier no. 10.1109/TC.2018.2849376

0018-9340 © 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

Authorized licensed use limited to: Universidad de Zaragoza. Downloaded on March 12, 2020 at 22:16:01 UTC from IEEE Xplore. Restrictions apply.

- [19] F. Candel, A. Valero, S. Petit, D. Suárez-Gracia, and J. Sahuquillo, "Exploiting data compression to mitigate aging in GPU register files," in *Proc. 29th Int. Symp. Comput. Archit. High Perform. Comput.*, 2017, pp. 57–64.
- [20] J. Abella, X. Vera, and A. González, "Penelope: The NBTI-aware processor," in *Proc. 40th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2007, pp. 85–96.
- [21] S. Wang, T. Jin, C. Zheng, and G. Duan, "Low power aging-aware register file design by duty cycle balancing," in *Proc. Des. Autom. Test Eur. Conf. Exhib.*, 2012, pp. 546–549.
- [22] R. Vattikonda, W. Wang, and Y. Cao, "Modeling and minimization of PMOS NBTI effect for robust nanometer design," in *Proc. 43rd ACM/IEEE Des. Autom. Conf.*, 2006, pp. 1047–1052.
- [23] M. Powell, S.-H. Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar, "Gated-Vdd: A circuit technique to reduce leakage in deep-submicron cache memories," in *Proc. Int. Symp. Low Power Electron. Des.*, 2000, pp. 90–95.
- [24] S. Kaxiras, Z. Hu, and M. Martonosi, "Cache decay: Exploiting generational behavior to reduce cache leakage power," in *Proc. 28th Annu. Int. Symp. Comput. Archit.*, 2001, pp. 240–251.
- [25] A. Calimera, E. Macii, and M. Poncino, "Analysis of NBTI-induced SNM degradation in power-gated SRAM cells," in *Proc. IEEE Int. Symp. Circuits Syst.*, 2010, pp. 785–788.
- [26] H. Tabkhi and G. Schirner, "Application-guided power gating reducing register file static power," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 22, no. 12, pp. 2513–2526, Dec. 2014.
- [27] NVIDIA OpenCL Best Practices Guide, Nvidia Corporation, 2009.
- [28] Intel®FPGA SDK for OpenCL Best Practices Guide, Intel Corporation, Tech. Rep. UC-20152, 2017.
- [29] OpenCL Optimization Guide, Advanced Micro Devices, Inc., 2013.
- [30] P. Xiang, Y. Yang, M. Mantor, N. Rubin, L. R. Hsu, and H. Zhou, "Exploiting uniform vector instructions for GPGPU performance, energy efficiency, and opportunistic reliability enhancement," in *Proc. 27th Int. ACM Conf. Supercomput.*, 2013, pp. 433–442.
- [31] AMD Accelerated Parallel Processing (APP) Software Development Kit (SDK), Advanced Micro Devices, Inc., 2016.
- [32] S. Ganapathy, R. Canal, A. González, and A. Rubio, "IRMW: A low-cost technique to reduce NBTI-dependent parametric failures in L1 data caches," in *Proc. IEEE 32nd Int. Conf. Comput. Des.*, 2014, pp. 68–74.
- [33] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques," in *Proc. Int. Conf. Comput.-Aided Des.*, 2011, pp. 694–701.
- [34] AMD Graphics Cores Next (GCN) Architecture, Advanced Micro Devices, Inc., 2012.
- [35] Various Methods of DRAM Refresh, Micron Technology, Inc., Tech. Rep. TN-04-30, 1999.
- [36] H. Pilo, C. A. Adams, I. Arsovski, R. M. Houle, S. M. Lamprier, M. M. Lee, F. M. Pavlik, S. N. Sambatur, A. Seferagic, R. Wu, and M. I. Younus, "A 64Mb SRAM in 22nm SOI technology featuring fine-granularity power gating and low-energy power-supply-partition techniques for 37% leakage reduction," in *Proc. IEEE Int. Solid-State Circuits Conf. Dig. Tech. Papers*, 2013, pp. 322–323.
- [37] AMD Graphics Core Next Architecture, Generation 3, Reference Guide, Advanced Micro Devices, Inc., 2016.
- [38] R. Ubai, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2Sim: A simulation framework for CPU-GPU computing," in *Proc. 21st Int. Conf. Parallel Archit. Compilation Techn.*, 2012, pp. 335–344.
- [39] N. Gong, S. Jiang, J. Wang, B. Aravamudhan, K. Sekar, and R. Sridhar, "Hybrid-cell register files design for improving NBTI reliability," *Elsevier Microelectron. Rel.*, vol. 52, no. 9/10, pp. 1865–1869, 2012.
- [40] A. Valero, N. Miralaei, S. Petit, J. Sahuquillo, and T. M. Jones, "On microarchitectural mechanisms for cache wearout reduction," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 25, no. 3, pp. 857–871, Mar. 2017.
- [41] E. Mintarino, V. Chandra, D. Pietromonaco, R. Aitken, and R. W. Dutton, "Workload-dependent NBTI and PBTI analysis for a sub-45 nm commercial microprocessor," in *Proc. IEEE Int. Rel. Physics Symp.*, 2013, pp. 1–6.
- [42] S. Kothawade, K. Chakraborty, and S. Roy, "Analysis and mitigation of NBTI aging in register file: An end-to-end approach," in *Proc. 12th Int. Symp. Quality Electron. Des.*, 2011, pp. 1–7.

▷ For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.



Alejandro Valero received the PhD degree in computer engineering from the Universitat Politècnica de Valencia, Spain, in 2013. From 2013 to 2015, he was a visiting researcher with Northeastern University, Boston, Massachusetts, and the University of Cambridge, United Kingdom. Since 2016, he has been an assistant professor with the Department of Computer Science and Systems Engineering, Universidad de Zaragoza, Spain. His current research interests include GPU architectures, memory hierarchy design, energy efficiency, and reliability. He is a member of the Aragon Institute of Engineering Research (I3A) and the HiPEAC European NoE.



Francisco Candel received the BS and MS degrees in computer engineering from the Universitat Politècnica de Valencia (UPV), Spain, in 2012 and 2014, respectively. He is currently working toward the PhD degree in the Department of Computer Engineering (DISCA), Universitat Politècnica de Valencia. His PhD research focuses on GPU modeling and efficient memory hierarchies for future GPUs.



Dario Suárez-Gracia (S08,M12) received the PhD degree in computer engineering from the Universidad de Zaragoza, Spain, in 2011. From 2012 to 2015, he was with Qualcomm Research Silicon Valley. Since 2015, he has been an interim associate professor with the Universidad de Zaragoza. His research interests include parallel programming, heterogeneous computing, memory hierarchy design, and energy-efficient processor and network-on-chip microarchitectures. He is a member of the IEEE, the IEEE Computer Society, the ACM, and the HiPEAC European NoE.



Salvador Petit (M'07) received the PhD degree in computer engineering from the Universitat Politècnica de Valencia (UPV), Spain. Since 2009, he has been an associate professor with the Computer Engineering Department, UPV, where he has been teaching several courses on computer organization. He has authored more than 100 refereed conference and journal papers. His current research interests include multi-threaded and multicore processors, memory hierarchy design, task scheduling, and real-time systems. He is a member of the IEEE and the IEEE Computer Society. In 2013, he received the Intel Early Career Faculty Honor Program Award.



Julio Sahuquillo (M'04) received the BS, MS, and PhD degrees from the Universitat Politècnica de Valencia, Spain, all in computer engineering. He is a full professor with the Department of Computer Engineering, Universitat Politècnica de Valencia. He has taught several courses on computer organization and architecture. He has authored more than 120 refereed conference and journal papers. His current research interests include multi- and many-core processors, memory hierarchy design, cache coherence, GPU architecture, and architecture-aware scheduling. He is a member of the IEEE and the IEEE Computer Society.

6.2 Trabajos presentados en congresos nacionales o internacionales

Improving performance by merging cache levels

Darío Suárez Gracia, Teresa Monreal Arnal, Víctor Viñals Yúfera
gaZ - DIIS - I3A – Universidad de Zaragoza
C/ María de Luna 1, 50018 Zaragoza
{dario, tmonreal, victor}@unizar.es

Abstract

Memory hierarchy plays a very important role in the performance of microprocessors. Each new generation, caches occupy a growing percentage of the processor area. To maximize performance, this area is split into multiple cache levels. Those levels close to the processor are faster but smaller while the further ones are slower but bigger. When the cache hierarchy does not fit the working set, i. e., the size of the working set is between levels one and two, performance decreases notably.

This paper proposes a cache organization that merges multiple cache levels into a single structure to blur the differences in latency and size among them; therefore, the new organization fits easier the working sets and improves performance.

Preliminary results give a hint about the potential of gradual caches, g-caches, for short. The simulation of SPEC CPU 2K in a cycle-accurate environment with consumption and delay models shows that when comparing to a conventional L1-L2 cache, g-cache contributes to increase IPC while reducing energy consumption. G-caches improve IPC up to 14.1% and 7.4% for integer and floating point, respectively, while reducing the average energy in around 20%, mostly due the reduction in the static consumption.

1 Introduction

During the last decades, the processor speed growth rate has outpaced the one of main memory. This fact together with the improvements in instruction level parallelism, which put more pressure on the memory hierarchy, have forced the inclusion of more and more caches inside the chips. First, a small L1 cache was added, but this was not enough to keep

the pace; hence, the L1 was backed with a larger multicycle L2. Nowadays, a large multi-megabyte third level has been included [14], or the L2 has grown considerably [13].

Some researchers have shown the limitations of these large on-chip cache hierarchies in deep-submicron technologies. Kim *et al.* proposed NUCA which exposes the sub-banking of L2/L3 caches to the microarchitecture and enables block migration within the cache [11]. Chishiti *et al.* take advantage of the sequential access between the tag and data arrays to decouple the placement in the data banks. They also migrate the most referenced blocks to the fastest banks [6]. Both proposals increase the cache hierarchy performance by closing the on-die cache gap between L2 and L3 caches. Balasubramonian *et al.* have provided evidence of the trade-off between the latency and the size of L1 and L2 caches [4]. Their reconfigurable cache joints the L1 and L2 to adapt the cache to the working set of each execution phase. These papers prove that a conventional cache hierarchy is not optimal for any given set of applications. This work follows this line and empirically proves that the large latency step between the first two cache levels causes inefficiencies in performance and energy which can be mitigated with more gradual cache hierarchies.

Ideally, one could envision low latency, high bandwidth, and large size cache to close the on-die cache gap but such cache is technologically unfeasible; however, first level caches offer low latency and high bandwidth. In order to add the third factor, size, our approach replicates the L1 cache in such a way that each cycle a growing number of replicas is visited. Replicas are connected by means of several networks tuned for the cache operations.

Figure 1 shows a conventional and a gradual cache hierarchy. In Figure 1(b), the L1/L2 caches

cache, where it comes from the cache plus the routing delay. The routing complexity in NUCA hinders its coupling with L1 caches. G-caches already integrate the L1 and use three networks to reduce delay and congestion inside the cache.

6 Conclusions and Future work

This paper introduces the gradual cache organization as a novel design that diffuses the on-die cache gap between the first two levels of conventional cache hierarchies. G-caches are oriented to offer increasing capacity by climbing up short latency steps. G-cache structure consists of a set of cache tiles connected by three networks, and its complexity-effective design is based on cache replication and on simple network policies.

From our results, a small sized g-cache, 112 KB, outperforms any conventional configuration, with up to 1 MB L2 caches. For the SPEC CPU 2K integer programs this g-cache reaches a speed-up average over its conventional counterpart of 14.1%. This figure is 7.4% for the floating point codes when comparing the higher conventional and g-cache configurations. This advantage in IPC is also translated into a reduction of the static energy consumption, mainly due to the reduction in execution time and number of transistors. The evaluated g-cache design is able to reduce the consumption up to a 25%. The increment observed in the dynamic consumption is override by the static energy savings that g-cache obtains.

Finally, our future work will focus on the full flattening of the on-die cache hierarchy with the use of designs which merge g-caches with NUCA, NuRAPID, or TLC designs.

Acknowledgements

This work has been supported by the Diputación General de Aragón grant "Grupo Consolidado de Investigación" (BOA 20/04/2005), the Spanish Ministry of Education and Science grant TIN2004-07739-C02-02, and the European Union Network of Excellence HiPEAC (High-Performance Embedded Architectures and Compilers, FP6-IST-004408).

References

- [1] T. Austin and D. Burger. *SimpleScalar Tutorial (for tool set release 2.0)*. SimpleScalar LCC, 1997.
- [2] N. Azizi, F. N. Najm, and A. Moshovos. Low-leakage asymmetric-cell SRAM. *IEEE Trans. VLSI Syst.*, 11(4), 2003.
- [3] P. Bai *et al.* A 65 nm logic technology featuring 35 nm gate length, enhanced channel strain, 8 Cu interconnect layers, low-k ILD and 0.57 μm SRAM cell. In *Proc. of the IEEE Int'l Symp. on Device Process and Materials*, 2003.
- [4] R. Balasubramonian *et al.* Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *Proc. of the 33rd Ann. Int'l Symp. on Microarchitecture(MICRO-33)*, 2000.
- [5] B. M. Beckerman and D. A. Wood. TLC: Transmission line caches. In *Proc. of the 36th Ann. IEEE/ACM Int'l Symp. on Microarchitecture(MICRO-36)*, pages 25–34, 2003.
- [6] Z. Chisholm, M. D. Pease, and T. N. Vijayumar. Distance associativity for high-performance energy-efficient non-uniform cache architectures. In *Proc. of the 36th Ann. Int'l Symp. on Microarchitecture(MICRO-36)*, 2003.
- [7] W. J. Henning and B. P. Tocino. *Handbook of Practices of Interconnection Networks*. Morgan Kaufmann, 2004.
- [8] J. L. Henning. SPEC CPU2006: Measuring cpu performance in the new millennium. *Computer*, 33(7):28–35, 2000.
- [9] R. Ho, K. W. Mai, and A. Raghunathan. The future of wires. *Proc. of the IEEE*, 89(4):490–504, April 2001.
- [10] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and perfect prefetching. In *Proc. of the 21st Ann. Int'l Symp. on Computer Architecture(ISCA'90)*, pages 364–373, 1990.
- [11] C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proc. of the 10th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems(ASPLOS-XI)*, pages 211–222, 2002.
- [12] N. S. Kim *et al.* Leakage current: Moore's law meets static power. *IEEE Computer*, 36(12):68–75, 2003.
- [13] J. McNamee. The Sixth Generation. *Microprocessor Report*, 2:1–10, 2006.
- [14] C. McNary and R. Bhatia. Montecito: A Dual-Core, Dual-Thread Itanium Processor. *IEEE Proc. 25(02):10–20*, 2005.
- [15] P. R. Pachence, E. Petrich, and N. P. Jouppi. Configurable caches and their application to media processing. In *Proc. of the 27th Ann. Int'l Symp. on Computer Architecture(ISCA'00)*, pages 214–224, 2000.
- [16] T. Sheppard, E. Petrich, G. Hamerly, and B. Calder. Automatically characterizing large-scale program behavior. In *Proc. of the 10th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems(ASPLOS-XI)*, pages 45–57, 2002.
- [17] T. Stoenescu. Xeon-3 aims at MP servers. *Microprocessor Report*, 10:1–6, October 1997.
- [18] V. Stejanovic and V. G. Oklobdzija. Comparative analysis of master-slave latches and flip-flops: High-performance and low-power designs. *IEEE J. of Solid-State Circuits*, 34(4):536–548, 1999.
- [19] D. Tarjan, S. Thoziyoor, and N. P. Jouppi. CACTI 4.0: An integrated cache access time, cycle time, area, aspect ratio and power model. Technical Report HPL-2006-80, HP Laboratories Palo Alto, 2006.
- [20] J. Wu, D. Weiss, C. Morganti, and M. Dresen. The Asynchronous 24 MB On-Chip Level 2 Cache for Dual-Core Itanium Processors. In *Proc. of the Int'l Solid State Circuits Conf. Digest of Technical Papers*, 2005.
- [21] C. Zhang, F. Vahid, and W. Najm. A highly configurable cache architecture for embedded systems. In *Proc. of the 30th Ann. Int'l Symp. on Computer Architecture(ISCA'03)*, pages 136–146, 2003.
- [22] W. Zhao and Y. Cao. New generation of predictive technology model for sub-45nm des. exploration. In *Proc. of the 7th Int'l Symp. on Quality Electronic Design(ISOQED'06)*, pages 585–590, 2006.

Introducing Energy and Power in Computer Architecture Laboratories

Alicia Asín Pérez Dario Suárez Gracia, Víctor Viñals Yúfera
Libelium Comunicaciones Distribuidas giz - DIIS - I3A - Universidad de Zaragoza
C\, María de Luna 11, 50018 Zaragoza C\, María de Luna 1, 50018 Zaragoza
a.asin@libelium.com {dario, victor}@unizar.es

Abstract

Power has emerged as a major concern in the microprocessor industry. From embedded to high-performance processors, all designs employ power optimization techniques at the circuit and the architectural levels.

While introductory computer architecture books and courses are starting to cover power concepts, proposals to offer students a practical experience with power issues are still scarce. To do so, we advocate for the inclusion of energy and power concepts in computer architecture courses by means of laboratory experiments. These experiments build upon concepts presented in preceding physics and/or electronics courses.

This paper outlines our experience with the development of such hardware-based energy laboratories. We propose experiments on a simple, yet powerful hardware-software platform capable of live energy measurements in a desktop computer processor. The proposed laboratory setup can help to teach students the basics of power-aware computer architectures. The performed experiments demonstrate the viability of our approach. For example, our experiments show that students can estimate the dynamic and static power dissipation of the Intel Pentium 4. Information not documented in the processor's datasheet.

1 Introduction

Technology scaling has permitted an outstanding growth pace in microprocessor performance. Smaller technologies make the transistors both faster and smaller allowing more of them to be integrated on the same die area. The integration of millions of transistors in a few square centimeters

results into a significant increase in power density. This power has to be dissipated [18]. To address this challenge, energy and power have to be considered from the very beginning of the microprocessor design. Accordingly, it is necessary for undergraduate education to expose our students to these concepts. However, undergraduate students are often unaware of energy and power concepts after completing their computer architecture courses. Neither the joint ACM & IEEE Computing Curricula 2001, Computer Science, nor the Computer Engineering 2004 Curriculum guidelines include any reference to energy and power in computer architecture courses [13, 14]. Even many of our students have forgotten the physics concept of current or voltage when they arrive to the computer architecture design courses.

A typical computer architecture course syllabus is quite large. Instead of removing existing concepts from the curriculum to introduce energy and power, we propose to do so in a laboratory setting. Via experiments students can gain insight on the trade-offs amongst processor architecture, compilation, and energy consumption. Students could understand the impact that power can have on a design and basic concepts related to power and energy. For example, consider two different processors for a battery operated device. Even if one of them consumes less power than the other, i.e., smaller average power, it may not be the best choice. If it spends more time to perform the same task, it may consume more energy reducing the operation time of the device.

These trade-offs could be easily understood with real-life experiments where students carry out energy measurements. Also, laboratory experiments can allow students to understand the energy-optimizations incorporated in modern processors, such as the Intel Core DUO whose microar-

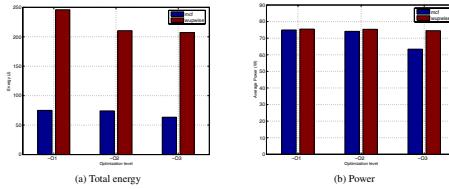


Figure 6: Mcf and wupwise varying the compilation optimization level

We describe the experimental setup and several simple experiments to show some of the platform applications. Many other possibilities exists, including more specialized studies on the efficiency and role of the cooling device. Breaking down the dynamic and static components of the total consumption is a very interesting experiment that discloses a fact not included in the public datasheets. The results shows that for the LU matrix computation, the static energy represents the 42.3 % of the total at the maximum frequency, 2.8 GHz, for a 130 nm Intel Pentium 4.

Since power is a real concern in computer architecture, we consider that its gradual inclusion in computer architecture courses will be beneficial. After finishing this project, we believe that an energy laboratory within the computer architecture courses taught at the University of Zaragoza will be useful for the students, and we will try to introduce it during the following years.

Acknowledgements

This work has been supported by the Diputación General de Aragón grant "Grupo Consolidado de Investigación" (BOA 20/04/2005), the Spanish Ministry of Education and Science grant TIN2004-07739-C02-01/02, and the European Union Network of Excellence HiPEAC (High-Performance Embedded Architectures and Compilers, FP6-IST-004408).

References

- [1] Analog Devices. *ADP3180. 6-Bit Programmable 2-, 3-, 4-Phase Synchronous Buck Controller*. Analog Devices, 2003.
- [2] A. Asín, D. Suárez, and V. Viñals. A proposal to introduce power and energy notions in computer architecture laboratories. In *Proceedings of the Workshop on Computer Architecture Education (WCAE'07)*, 2007.
- [3] J. Dongarra, L. London, S. Moore, P. Mucci, and D. Terpstra. Using papi for hardware performance monitoring on linux systems. In *Proceedings of the Conference on Linux Clusters: The HPC Revolution*, 2001.
- [4] J. L. Henning. Spec cpu2000: Measuring cpu performance in the new millennium. *Computer*, 33(7):28–35, 2000.
- [5] C. Hu, J. McCabe, D. A. Jiménez, and U. Kremer. Infrequent basic block-based program phase classification and power behavior characterization. In *Proceedings of The 10th IEEE Annual Workshop on Interaction between Compilers and Computer Architectures*. ACM Press, 2006.
- [6] Intel. *Voltage Regulator-Down (VRD) 10.0 Design Guide For Desktop Socket 478*. Intel Corporation, February 2004.
- [7] C. Isci and M. Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *Proceedings of the Int. Symp. on Microarchitecture, MICRO'03*, page 93, Los Alamitos, CA, USA, 2003. IEEE Computer Society.

Forge: A Multi-purpose Platform for Measuring Energy and Temperature in Commodity PCs

Sergio Gutierrez, Octavio Benedí, Darío Suárez, Jose María Marín, and Víctor Viñals

Abstract—Performance has driven the microprocessor industry for the last twenty years. Its effort has enabled to multiply by several orders of magnitude the available computational power; e.g., the Intel 8080 was able to execute 0.64 MIPS and the newest Core i7 can execute 6400 MIPS. The cost of this fabulous improvement has been a similar rise in energy consumption. Nowadays, we have reached a point in which the most limiting factor for improving performance is energy dissipation.

In order to keep the performance improvement during the next years, it is necessary to study energy and temperature in depth. Nevertheless, most current computer architecture curricula include neither energy nor temperature. The lack of simple experimental platforms contributes to the difficulty in teaching these topics. Willing to ease this problem, we propose in this work Forge, a platform for easily measuring energy/power and temperature in real processors. Forge is oriented for academic and research-oriented experiments as well. For instance, we describe an interesting undergraduate laboratory that analyzes the interaction between compiler optimizations and energy. With this laboratory, students can learn that performance optimizations usually reduce energy but may increase power.

Index Terms—Energy, Temperature, Measuring, Evaluation, Power, Compiler Optimizations, and Education.

I. INTRODUCTION

ENERGY and temperature have emerged as main constraints for processor design, and also as a limiting factor in the end user productivity. On one hand, in the embedded domain, lowering the energy consumed by the processor increments the device uptime. On the other hand, in the commodity segment, the cooling system affects the performance when it is not able to dissipate all the generated heat and forces a frequency/voltage reduction in the processor.

While the evaluation of many design constrains, such as performance, may be done by means of a simulator of the processor micro-architecture, the evaluation of energy and temperature requires much more sophisticated simulators often not very accurate due to the lack of technology process information. Moreover, energy not only depends on the executed instructions but also in the input data because different values may produce a different number of switches in the transistors increasing even more the complexity of its simulators.

Another way of obtaining energy or temperature measures consists of getting the data by instrumenting real hardware. Many authors have performed power measurements [1], [2], and others such us Mesa-Martínez *et al.* have measured

Gutiérrez, Benedí, Suárez, and Viñals are with the Dpto. de Informática e Ingeniería de Sistemas, Universidad de Zaragoza, e-mail : {sirgutii, octavio.benedi}@gmail.com, {dario, victor}@unizar.es.
Marín is with the Dpto. de Ingeniería Mecánica, Universidad de Zaragoza, e-mail: jmm@unizar.es.

temperature in commodity PCs [3]. This paper follows this path and describes Forge, a platform able to measure at the same time energy and temperature in a commodity PC, an Intel Pentium 4. The description gives all the details for the platform replication, and two use cases are studied to see its potential for academic and research purposes. The first case deals with the interaction between compiler optimizations and energy, and it is oriented to be a laboratory session in a undergraduate course on computer architecture. The second one studies how each instruction of the ISA warms the processor because this information could be very useful for power/temperature-aware instruction schedulers.

This paper is organized as follows. Section II comments on the related work. Section III describes the measure platform in detail. Section IV explores two possible uses of Forge, and Section V concludes and present some possible future work lines.

II. RELATED WORK

Energy and temperature have aroused the interest in energy and power in both industry and academia. In the industrial side, SPEC has introduced SPECpower_ss2008 focusing on the consumption of server computers [4], and EEMBC has defined EnergyBench establishing a framework for adding energy to the metrics of the EEMBC's performance benchmarks [5].

Many studies have been conducted in the academic side. Regarding energy, Isci and Martonosi describes a methodology for obtaining per-unit power estimations combining real power measurements with performance counters [1]. Other authors have proposed infrastructures based on an Intel Pentium 4 for characterizing program phases, evaluating compiler optimizations, or studying energy [6], [7], [2]. Temperature measurements requires more sophisticated setup; e.g., Mesa-Martínez *et al.* have proven some power estimations using IR thermal imaging [3]. Forge differs from all previous approaches in the combination of energy/power and temperature in a single platform, allowing to perform more complex studies on the relationships between both magnitudes.

III. PLATFORM DESCRIPTION

The measurement platform is based in our previous work and consists of two commodity PCs [2]. One, named computer under test (CUT), is monitored, and another, named data acquisition and storage computer (DASC), acquires and saves all the power and temperature samples gathered from the CUT. Both computers are shown in Figure 1a, the CUT in the left and the DASC in the right.

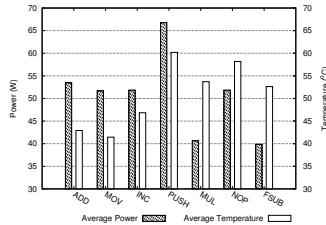


Figure 5: Average Power and Temperature for a small subset of the Intel x86 ISA.

The second one describes a preliminary study of the power and temperature of individual instructions and shows that stack operations consume and heat up more than any other instructions in the Intel Pentium 4.

Our future work will try to extend Forge reducing the granularity of the sampling process. Now, the platform does not know at which function each sample belongs. We believe that this ability will help us finding the most heat-producing instruction sequences to continue our studies on per instruction energy estimations.

ACKNOWLEDGEMENTS

Darío Suárez and Víctor Viñals were supported by the Spanish Ministry of Education and Science under contracts TIN2007-66423 and Consolider CSD2007-00050, by the Gobierno de Aragón grant “gaZ: Grupo Consolidado de Investigación”, and by the European Union Network of Excellence HiPEAC-2 (FP7/ICT 217068).

REFERENCES

- [1] Canturk Isci and Margaret Martonosi, “Runtime power monitoring in high-end processors: Methodology and empirical data,” in *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, Washington, DC, USA, 2003, p. 93, IEEE Computer Society.
- [2] Alicia Asín Pérez, Darío Suárez Gracia, and Víctor Viñals Yáñez, “A proposal to introduce power and energy metrics in computer architecture laboratories,” in *WCAC ’07: Proceedings of the 2007 workshop on Computer architecture education*, New York, NY, USA, 2007, pp. 52–57, ACM.
- [3] Francisco Javier Mesa-Martínez, Joseph Nayach-Battilana, and Jose Renau, “Power model validation through thermal measurements,” in *ISCA ’07: Proceedings of the 34th annual international symposium on Computer architecture*, New York, NY, USA, 2007, pp. 302–311, ACM.
- [4] Standard Performance Evaluation Corporation, “SPECpower_ssj2008 benchmark suite. http://www.spec.org/power_ssj2008/.” 2008.
- [5] EEMBC. The Embedded Microprocessor Benchmark Consortium, “EnergyBench™ version 1.0 power/energy benchmarks. http://www.eembc.org/benchmark/power_sl.php.” 2008.
- [6] Ming Hu, Michael McCabe, Daniel A. Jiménez, and Ulrich Kremer, “Inferring basic block-based program phase classification and power behavior characterization,” in *Proceedings of The 10th IEEE Annual Workshop on Interaction between Compilers and Computer Architectures*. 2006, ACM Press.
- [7] John S. Seng and Dean M. Tullsen, “The effect of compiler optimizations on pentium 4 power consumption,” in *Seventh Annual Workshop on Interaction between Compilers and Computer Architectures (INTERACT’03)*, 2003, p. 51.
- [8] Analog Devices, *ADP3180, 6-Bit Programmable 2-, 3-, 4-Phase Synchronous Clock Controller*, Analog Devices, 2008.
- [9] Tektronix, “Tektronix tpc-312 current probe. <http://www2.tek.com/camcorp/pd/details.htm?xt=tpcPs&ci=135404cccepsaklc=EN>.” 2008.
- [10] Adlink Technology Inc., “Adlink pei-9112 data acquisition card. http://www.adlinktech.com/PD/web/PD_detail.php?cKind=&pId=29&seq=&id=&size=.” 2008.
- [11] Intel, *Intel® Pentium® 4 Processor in the 478-Pin Package Thermal Design Guidelines*, Intel Corporation, 1st edition, May 2002.
- [12] Pico Technologies, *USB TC-08 Temperature Logger User’s Guide*, Pico Technologies Limited, 2007.
- [13] Jas M. Rabay, Anantha Chandrakasan, and Borivoje Nikolic, *Digital Integrated Circuits. A design perspective*, Prentice Hall Electronics and VLSI series. Prentice Hall, second edition, 2003.
- [14] David Brooks, Robert P. Dick, Russ Joseph, and Li Shang, “Power, thermal, and reliability modeling for microarchitectural-scale microprocessors,” *IEEE Micro*, vol. 28, no. 3, pp. 49–62, May-June 2007.
- [15] W. Zhang, J. S. Hu, V. Degalaiah, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, “Compiler-directed instruction cache leakage optimization,” in *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture*, 2002, p. 208, IEEE Computer Society.
- [16] Stefanos Karaisa and Margaret Martonosi, *Computer Architecture Techniques for Power-Efficiency*, Number 4 in Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2008.
- [17] John L. Henning, “Spec cpus2006 benchmark descriptions,” *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, 2006.
- [18] Aashish Phansalkar, Ajay Joshi, and Lizy K. John, “Analysis of redundancy and application balance in the spec cpus2006 benchmark suite,” in *ISCA ’07: Proceedings of the 34th annual international symposium on Computer architecture*, New York, NY, USA, 2007, pp. 412–423, ACM.
- [19] Intel Performance Tuning Utility 3.1 Update 3. <http://software.intel.com/en-us/articles/intel-performance-tuning-utility-31-update-3>, 2007 edition.
- [20] Gcc team, *GCC 4.1.2 Manual*, <http://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/>, Free Software Foundation, February 2008.
- [21] Intel, *Intel C++ Compiler 10.1 Professional edition*, <http://www.intel.com/cd/software/products/asmo-na/eng/277618.htm>, 2007 edition.
- [22] Madhavi Valluri and Lizy John, “Is compiling for performance == compiling for power?,” in *Fifth Annual Workshop on Interaction between Compilers and Computer Architectures (INTERACT’00)*, 2001, p. 51.
- [23] Wei Wu, Lingling Jin, Jun Yang, Pu Liu, and Sheldon X.-D. Tan, “A systematic approach for power-aware compilation of x86 microprocessors,” in *DAC ’06: Proceedings of the 43rd annual conference on Design automation*, New York, NY, USA, 2006, pp. 554–557, ACM.
- [24] Stefan Steinke, Markus Knauer, Lars Weinmeyer, and Peter Marwedel, “An accurate and fine grain instruction-level energy model supporting software optimizations,” in *In Proc. of The International Workshop Power And Timing Modeling, Optimization and Simulation*, 2001.

Nanotubos de Carbono para conexiones en Caches: Arquitecturas más allá del CMOS

Jaime Ortiz Cirugeda, Dario Suárez Gracia, Víctor Viñals Yúfera, María Villarroya-Gaudó¹

Resumen—En este artículo se presenta una nueva solución tecnológica al límite del escalado CMOS basada en la viabilidad de la sustitución de las interconexiones de cobre tradicionales en tecnología CMOS por nanotubos de carbono. En particular se demuestra como con los nanotubos multiplica se puede disminuir tanto el retardo de la conexión en las interconexiones globales como su consumo energético. Por tanto se pueden considerar futuros candidatos para interconexiones de larga distancia dentro del chip como pueden ser las caches de último nivel.

Palabras clave—Nanoarquitectura, Nanotubos de carbono, SWCNT, MWCNT, tecnología, interconexión

I. INTRODUCCIÓN: NUEVA TECNOLOGÍA O CAMBIO DE PARADIGMA

UNO de los grandes retos de la arquitectura actual está en analizar como los cambios tecnológicos pueden afectar a los diseños de los procesadores. Desde hace décadas se habla del final del CMOS, dado que el aumento de rendimiento conseguido hasta ahora basado en la reducción de las dimensiones de los transistores tiene un fin. Este fin se debe en parte a limitaciones tecnológicas, parcialmente superables con los avances en nanotecnología, y en parte a la limitación física de la anchura de canal, que al acercarse a dimensiones atómicas introduce efectos cuánticos que cambian drásticamente el funcionamiento de los transistores. Estas limitaciones no son superables sin un cambio de paradigma.

En las últimas décadas se está investigando en el desarrollo de nuevas tecnologías como posibles sustitutas del CMOS, así nos encontramos con la

computación cuántica, los computadores basados en microfluídica o los ordenadores biológicos por citar algunas de las líneas de trabajo. Todas estas aproximaciones suponen un cambio de paradigma, por ejemplo la computación cuántica se basa en una lógica cuaternaria [1,2]. Estos cambios de paradigma suponen una fuerte inversión para convertirse en sustitutos reales del CMOS y se encuentran todavía en fases muy básicas de desarrollo.

Recientemente investigaciones más aplicadas apuestan por modificaciones de las tecnologías CMOS de forma que se consiga aumentar el rendimiento sin reducir la escala de integración (en el sentido tradicional de disminuir la longitud de canal del transistor). Así por ejemplo encontramos la tecnología CMOL (Cmos-MOLEcular): que combina CMOS con moléculas [3] o el uso de nanotubos de carbono en el interconexionado (CNT, *carbon nanotubes*) debido a sus mejores propiedades respecto a las interconexiones tradicionales en cobre.

La tabla I presenta el estado del arte de fabricación de algunos nanodispositivos, donde los encabezados intentan indicar en términos cualitativos el nivel relativo de progreso para hacer chips relativamente grandes (con más de 100 dispositivos), no se distingue entre si son tecnologías para lógica o para memorias [2].

En este artículo se presentan los resultados de una nueva línea de investigación del Grupo de Arquitectura de Computadores de la Universidad de Zaragoza basado en

TABLA I
ESTADO DE DESARROLLO DE NANODISPOSITIVOS Y DISPOSITIVOS CON DIMENSIONES NANOMÉTRICAS

DISPOSITIVO	Único dispositivo	Circuito Simple	Puerta lógica/Celda de Memoria	Subsistema	Chip pequeño	Chip grande
CMOS						
Transistor orgánico						
Memoria de un único electrón						
Transistor de nanotubo/cable						
Lógica de un único electrón						
Molecular (hibrido electromecánico)						
Dispositivo electrónico cuántico						

Fase de Prefabricación Sin información Teoría Simulación
 Fabricación Agonizante Demostrado funcionamiento Disponible comercialmente

¹ Grupo de Arquitectura de Computadores (gaZ), Dpto. de Informática e Ingeniería de Sistemas, Universidad de Zaragoza, e-mail: jaimeortizcirujeda@gmail.com, {dario, victor, Maria.villarroya}@unizar.es.

La potencia dinámica observamos que se incrementa ligeramente en el caso de los MWCNT con repetidores. Este hecho que en principio podría considerarse perjudicial se mejorará dado que la estimación de repetidores a utilizar es la misma la que optimiza el caso del cobre, dado que la resistencia lineal de los nanotubos es menor, el número de repetidores necesarios disminuirá, mejorando así tanto el consumo como el área.

En relación con el área de la memoria decir que no se han encontrado variaciones significativas, dado que él área de una cache está fuertemente determinada por el tamaño de la celda y no el interconexión y ésta no ha sido modificada.

VII. CONCLUSIONES Y TRABAJO

En el apartado anterior se ha demostrado que para interconexiones semiglobales y globales los nanotubos de carbono MWCNT metálicos son competitivos sustitutos de las interconexiones de cobre, se pueden conseguir ahorros de tiempo de acceso del 63%.

En el caso de los SWCNT solo los completamente metálicos (con densidad metálica unitaria) serían ideales para conexiones semiglobales, si bien tecnológicamente hoy no pueden considerarse fabricables.

Como trabajo futuro será preciso revisar el posicionamiento de los repetidores de forma que sean optimizados en el caso de los nanotubos de carbono. Además las mejoras obtenidas permiten diseñar sistemas de memoria con mayores distancias de cableado,

VIII. AGRADECIMIENTOS

El presente trabajo ha sido financiado mediante los proyectos CICYT TIN2007-66423 y Consolider CSD2007-00050 del Ministerio de Educación y Ciencia y por la ayuda "gaZ. Grupo Consolidado de Investigación" del Gobierno de Aragón y por la red de excelencia europea HIPEAC-2 (FP7/ICT 217068).

VIII. REFERENCIAS

- [1] Communications of the ACM SPECIAL ISSUE: *Beyond silicon: new computing paradigm*, Septiembre 2007
- [2] M. Forshaw R. Stuller, The Status of Research into Architectures for Nanoelectronic Systems in the European Research Area (RANS), E Nano Newsletter, Junio 2005.
- [3] Shamik Das, Garrett S. Rose, Matthew M. Ziegler, Carl A. Picconetto, James C. Ellenbogen, *Architectures and Simulation for Nanoprocessors Systems Integrated on the Molecular Scale*, Lecture Notes Physics, 2005.
- [4] Hong Li, Wen-Yan Yin, Kaustav Banerjee, Jin-Fa Mao, Circuit Modeling and Performance Analysis of Multi-Walled Carbon Nanotube Interconnects, IEEE Transactions on electron devices, Junio 2008.
- [5] Banu Agrawal, Navin Srivastava, Frederic T. Chong, Kaustav Banerjee, Timothy Sherwood, *Nano-enhanced Architectures: Using Carbon Nanotube Interconnects in Cache Design*, Proceedings of the 4th workshop on Non-Silicon Computing (NSC-4) held in conjunction with the 2007 International Symposium on Computer Architecture (ISCA'07 workshop), Junio 2007.
- [6] W.I. Milne, M. Mann, J. Dijon, P. Bachmann, J. McLaughlin, J. Robertson, K.B.K. Teo, A. Lewalter, M. de Souza, P. Boggild, A. Briggs, K. Bo. Mogensen, J.-C. P. Gabriel, S. Roche, R. Baptist, *Carbon Nanotubes*, E Nano Newsletter, septiembre 2008.
- [7] W. Liang, M. Bockrath, D. Bozovic, J. H. Hafner, M. Tinkham, and H. Park, *Fabry- perot interference in a nanotube electron waveguide*, Nature, Junio. 2001.
- [8] O. Hjortstam, P Ibs, S. Söderholm, H. Dai, "Can we achieve ultra-low resistivity in carbon nanotube-based metal composites?" Applied Physics A: Materials Science & Processing, Mayo 2004.
- [9] S. Sato, M. Nihei, A. Mimura, A. Kawabata, D. Kondo, H. Shioya, T. Iwai, M. Mishima, M. Ohfuti, and Y. Awano, "Novel approach to fabricating carbon nanotube via interconnects using size-controlled catalyst nanoparticles," in Proc. International Interconnect Technology Conference, 2006
- [10] N. Srivastava and K. Banerjee, "Performance analysis of carbon nanotube interconnects via applications," in Proc. ICCAD 2005: Components and Design IEEE/ACM International Conference on, 6-10 Nov. 2005, pp. 383-390.
- [11] H. J. Li, W. G. Lu, J. Li, X. D. Bai and C. Gu, "Multi-channel ballistic transport in multiwall carbon nanotubes," Physical Review Letters, vol. 95, 2005
- [12] A. Naeemi and J. D. Meindl, "Compact physical models for multiwall carbon-nanotube interconnects," vol. 27, no. 5, pp. 338-340, May 2006.
- [13] M. Nihei, D. Kondo, A. Kawabata, S. Sato, H. Shioya, M. Sakae, T. Iwai, M. Ohfuti, and Y. Awano, "Low-resistance multi-walled carbon nanotube vias with parallel channel conduction of inner shells," Proc. IEEE 2005 International Interconnect Technology Conference Junio 2005.
- [14] C. Kim, D. Burger, and S. W. Keckler, An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches, In *10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 211-222, October 2002
- [15] <http://www.hpl.hp.com/research/cacti/> HP Labs. CACTI. An integrated cache and memory access time, cycle time, area, leakage, and dynamic power model

A Comparison of Cache Hierarchies for SMT Processors

Darío Suárez Gracia¹, Teresa Monreal Arnal², and Víctor Viñals Yífera¹

Abstract— In the multithread and multicore era, programs are forced to share part of the processor structures. On one hand, the state of the art in multithreading describes how efficiently manage and distribute inner resources such as reorder buffer or issue windows. On the other hand, there is a substantial body of works focused on outer resources, mainly on how to effectively share last level caches in multicores. Between these ends, first and second level caches have remained apart even if they are shared in most commercial multithreaded processors.

This work analyzes multiprogrammed workloads as the worst-case scenario for cache sharing among threads. In order to obtain representative results, we present a sampling-based methodology that for multiple metrics such as STP, ANTT, IPC throughput, or fairness, reduces simulation time up to 4 orders of magnitude when running 8-thread workloads with an error lower than 3% and a confidence level of 97%.

With the above mentioned methodology, we compare several state-of-the-art cache hierarchies, and observe that Light NUCA provides performance benefits in SMT processors regardless the organization of the last level cache. Most importantly, Light NUCA gains are consistent across the entire number of simulated threads, from one to eight.

Keywords— Cache Hierarchy, Multithreading, Simulation, Sampling, NUCA

I. INTRODUCTION

MULTITHREADING (MT) is supported by an ample spectrum of current processors devoted to uneven computing segments such as: embedded, high throughput, or high performance. Examples of representatives from these segments are Netlogic XLP832 (4-way multithreading, 8-cores), Oracle SPARC T3 (8-way multithreading, 16-cores), or IBM POWER7 (4-way multithreading, 4-6-8 cores), respectively [1], [2], [3].

All previous examples share a powerful multilevel cache hierarchy with large Last Level Caches (LLC), and only the XLP832 departs from the conventional organization including a ring for communicating the private L2 caches, the eight L3 cache banks, and the four DDR ports. While these LLCs seem able to accommodate the multiple working sets of SMT execution, sharing in the levels close to the processors proves to be more complex. On one hand, L1 and L2 caches deal with the latency-power vs. size trade-off. On the other hand, MT architectures add a new trade-off, number of threads in execution vs. miss rate. With many threads, cache misses can be tolerated executing instructions from other threads, but as

the number of threads grows, the collective working set becomes larger and more changing, resulting in miss ratios potentially harmful to performance. The larger their number, the larger and size changing the collective working set becomes and the larger the miss rate. So when the miss rate reaches a critical value in which threads execution fails to overlap, the processor stalls and has the same problem that single thread machines.

SMT architectures may be favored by caches designed to support working set awareness such as the L-NUCA [4]. L-NUCAs belong to a family of cache organizations that has received much attention for improving cache performance: Non-Uniform Cache Architecture (NUCA) [5]. The seminal NUCA work targets the wire delay problem¹, and proposes the melting of the L2 and L3 caches into a meshed array of caches banks. Nevertheless, to the best of our knowledge, there is little work on evaluating NUCA with simultaneous multithreading processors (SMT).

Part of the complexity of assessing multiple cache hierarchies lies in the required simulation framework. So to carry out the experiments, we propose a simple yet efficient MT simulation methodology ensuring the accuracy of the results abreast with a sort simulation time. The methodology is based on statistical sampling, and contrary to other alternatives does not require a prior long profiling of the applications.

This work gives two main contributions. The first one is introducing a powerful methodology to evaluate MT architectures. The second one is the comparison and evaluation of several state-of-the-art cache hierarchy organizations driven by the SPEC CPU2006 benchmark suite. From the results, we conclude that regardless the number of threads L-NUCAs outperform conventional multibanked and dynamic NUCA organizations, both in terms of throughput and fairness.

The rest of the paper is organized as follows. Section II elaborates on previous work. Section III presents the proposed evaluation methodology. Section IV describes our experimental framework and the hierarchies under test. Section V comments on the results, and Section VI concludes the paper.

II. BACKGROUND

Tullsen, Eggers, and Levy in their SMT seminal work compare the performance of private and shared L1 caches (for both instruction and data) and observe that regardless the number of threads (from 1 to 8) shared data caches are the best choice and private

¹Computer Architecture Group (gAZ). Dpto. de Informática e Ingeniería de Sistemas. Instituto de Investigación en Ingeniería de Aragón. Universidad de Zaragoza. e-mail: {darío, víctor}@unizar.es

²Department of Computer Architecture. Universitat Politècnica de Catalunya (UPC). e-mail: teresa@ac.upc.edu

¹The wire delay is longer than the bank delay and represents most part of the total cache latency in LLCs.

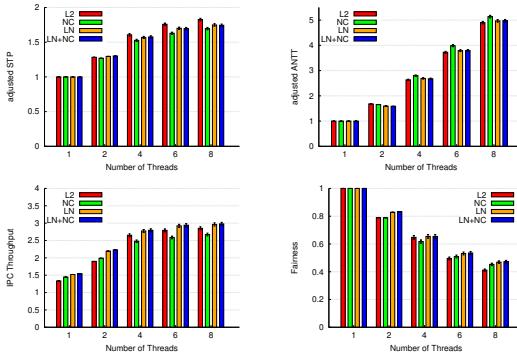


Fig. 5. Results for the best configuration of each organization: L2, NC, LN, and LN+NC correspond to the L2-256KB, NC-8x4-256KB, LN3-240KB, and LN2-NC-8x4, respectively

- ACM/IEEE international symposium on Microarchitecture*, Washington, DC, USA, 2001, pp. 318–327, IEEE Computer Society.
- [8] Sébastien Hily and André Seznec, “Contention on 2nd level cache may limit the effectiveness of simultaneous multithreading,” Tech. Rep. 1086, IRISA, février 1997.
 - [9] Alex Settle, Dan Connors, Enric Gibert, and Antonio González, “A dynamically reconfigurable cache for multi-threaded processors,” *J. Embedded Comput.*, vol. 2, no. 2, pp. 221–233, 2006.
 - [10] Mario Nedivošky and Wayne Yamamoto, “Quantitative studies of caches on a multi-named architecture,” in *In Workshop on Multithreaded Execution, Architecture and Compilation*, 1998.
 - [11] Hantak Kwak, Ben Lee, Ali R. Hurson, Suk-Han Yoon, and Woo-Jong Hahn, “Effects of multithreading on cache performance,” *IEEE Transactions on Computers*, vol. 48, pp. 176–184, 1999.
 - [12] Montse García, José González, and Antonio González, “Data cache for multithreaded processors,” in *Proc. of the Workshop on Multithreaded Execution, Architecture and Compilation*, 2000.
 - [13] Sébastien Hily and André Seznec, “Standard memory hierarchy does not fit simultaneous multithreading,” in *Proceedings of the 2nd Workshop on MULTI-THREADED EXECUTION: ARCHITECTURE AND COMPILATION (MTEAC-2)*, 1998.
 - [14] Subhraditya Sarkar and Dean M. Tullsen, “Data layout for cache performance on a multithreaded architecture,” in *Transactions on high-performance embedded architectures and compilers III*, Per Stenström, Ed., chapter Data layout for cache performance on a multithreaded architecture, pp. 43–68, Springer-Verlag, Berlin, Heidelberg, 2011.
 - [15] Sonia López, Steve Dropsho, David H. Albonesi, Oscar Garnica, and Juan Lanchares, “Dynamic capacity-speed tradeoffs in smt processor caches,” in *Proceedings of the 2nd annual conference on High performance embedded architectures and compilers*, Berlin, Heidelberg, 2007, HiPEAC’07, pp. 136–150, Springer-Verlag.
 - [16] Sonia Lopez, Oscar Garnica, David H. Albonesi, Steven Dropsho, Juan Lanchares, and Jose I. Hidalgo, “Adaptive cache memories for smt processors,” *Digital Systems Design, Euromicro Symposium on*, vol. 0, pp. 331–338, 2010.
 - [17] Steven E. Raasch and Steven K. Reinhardt, “The impact of resource partitioning on smt processors,” in *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, Washington, DC, USA, 2003, PACT ’03, pp. 15–, IEEE Computer Society.
 - [18] Michael Van Biesbrouck, Lieven Eeckhout, and Brad Calder, “Representative multiprogram workloads for multithreaded processor simulation,” in *IEEE Workload Characterization Symposium*, September 2007, pp. 193–203, IEEE Computer Society.
 - [19] F.J. Cazorla, A. Pajuelo, O.J. Santana, E. Fernandez, and M. Valero, “On the problem of evaluating the performance of multiprogrammed workloads,” *IEEE Trans. on Computers*, vol. 59, no. 12, pp. 1722–1728, 2010.
 - [20] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder, “Simpoint 3.0: Faster and more flexible program analysis,” in *Proc. of the Workshop on Modeling, Benchmarking and Simulation*, 2005.
 - [21] Thomas F. Wenisch, Roland E. Wunderlich, Michael Ferdman, Anastassia Ailamaki, Babak Falsafi, and James G. Hoe, “Simflex: Statistical sampling of computer system simulation,” *IEEE Micro*, vol. 26, pp. 18–31, July 2006.
 - [22] Raj Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*, John Wiley & Sons, Inc., April 1991.
 - [23] Ron Gabor, Shlomo Weiss, and Avi Mendelson, “Fairness and throughput in switch event multithreading,” in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, Washington, DC, USA, 2006, MICRO 39, pp. 149–160, IEEE Computer Society.
 - [24] S. Eggars and J. E. Eeckhout, “System-level performance metrics for multiprogram workloads,” *Micro, IEEE*, vol. 28, no. 3, pp. 42–53, may 2008.
 - [25] M. Ekman and P. Stenstrom, “Enhancing multiprocessor architecture simulation speed using matched-pair comparison,” in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, 2005, Washington, DC, USA, 2005, pp. 89–99, IEEE Computer Society.
 - [26] Todd Austin and Doug Burger, *SimpleScalar Tutorial (for tool set release 2.0)*, SimpleScalar LCC, 1997.
 - [27] Doug Hillman, Michael J. McKeown, Mike Upton, Daniel Rogge, Doug Commane, Alan Kyte, and Pradeep Dasgupta, “The microarchitecture of the Pentium® 4 processor,” *Intel Technology Journal*, vol. 1st quarter, pp. 1–13, 2001.
 - [28] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm, “Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor,” in *Proceedings. 23rd Annual International Symposium on Computer Architecture*, New York, NY, USA, 1996, vol. 24, pp. 191–202, ACM.
 - [29] John L. Henning, “Spec cpu2006 benchmark descriptions,” *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, 2006.

An Adaptive Controller to Save Dynamic Energy in LP-NUCA

Darío Suárez Gracia¹, Teresa Monreal Arnal², and Víctor Viñals Yúfera¹

Abstract—Portable devices often demand powerful processors to run computing intensive applications, such as video playing or gaming, and ultra low energy consumption to extend device uptime. Such conflicting requirements are hard to fulfil and appeal for adaptive hardware that only consumes energy when required.

LP-NUCA is a tiled cache organization aimed at high-performance low-power embedded processors that sequentially looks up for blocks ordered by temporal locality in groups of small tiles. Unfortunately, LP-NUCA has two main dynamic energy wasting sources: (a) blocks are continuously migrating among tiles even in low locality phases, (b) to reduce cache latency, the tag and data arrays of the tiles are always accessed in parallel.

This paper proposes a learning-based controller that dynamically tunes block migration and cache access policy between parallel and serial. During low temporal locality phases, the controller does not ask from the LP-NUCA root tile, L1, and forces a serial access to the tag and data arrays in the tiles, thus reducing the energy waste. Using a cycle-accurate simulator and energy estimations derived from an LP-NUCA layout, the proposed controller reduces dynamic energy by 20% on average for single and multi-thread workloads.

Keywords—Cache Hierarchy, Multithreading, Energy, Power, Embedded, NUCA

I. INTRODUCTION

THE way people use computers is partially shifting from personal computers with local data to mobile devices with data on the cloud. This “platform” displacement has not carried along “application” changes. Users almost demand the same performance in mobile devices that they used to experiment in desktop computers. Giving the same performance level with the tight energy constraints of mobile environments appeals for adaptive hardware that judiciously detects whether it is profitable to invest energy in order to satisfy the user.

One of the most energy-efficient mechanisms to achieve high-performance is the memory hierarchy [1], where several small caches pretend to be an unbound and fast storage thanks to the locality of programs. Non-Uniform Cache Architecture, NUCA, exploits locality at a finer granularity than conventional caches because they enable inter bank block migrations [2]. Light Power NUCA, LP-NUCA, is a variant of Light NUCA (L-NUCA) for high-performance low-power embedded processors, such as those of mobile devices, that conveys blocks through three specialized

Networks-in-Cache as L-NUCA does [3], [4], but also includes two static techniques for saving dynamic energy, Miss Wave Stopping and Sectoring. These techniques together with LP-NUCA ad-hoc network mechanism enable to outperform conventional and static NUCA organizations in terms of energy and performance.

The organization of LP-NUCA consists of many small tiles behaving as a very large distributed victim cache [5]. Blocks remain ordered by temporal locality (TL), so the L1, renamed root-tile (r-tile), recently evicted blocks have a lower service latency than those previously evicted. The LP-NUCA design relies on the temporal locality of programs along all their execution; hence, when the r-tile evicts a block, it triggers a chain of dominoes replacement for maintaining the TL block ordering. But an energy wasting problem can arise during low TL phases. During them, the r-tile floods the rest of tiles with blocks that will be seldom requested. Besides, these blocks degrade older ones that may be re-referenced in the near future. Moreover, LP-NUCA always accesses in parallel tag and data arrays to reduce cache latency. Since a data array access roughly consumes more than 5× the energy of the tag in LP-NUCA [4], this parallel access is a major waste of energy for requests with high likelihood of being a miss. Ideally, we would like to detect low locality phases to prevent the r-tile for evicting low locality blocks and to dynamically switch between parallel and serial access in the rest of tiles.

LP-NUCAs were conceived for single-thread processors; however, to increase their performance/energy ratio, current advanced embedded processors rely on extracting parallelism from multiple threads rather than from a single one. For example, the Intel Xeon LC3528, the MIPS MIPS32-1004K, or the Netlogic XLP832 simultaneously execute between 2 and 4 threads [6], [7], [8]. Traditionally, multi-threaded processors (MT) have shared all the cache hierarchy [9] increasing the chances of polluting the cache with useless blocks and evicting useful blocks from other threads. LP-NUCA in MT environments would suffer from this problem and would benefit from a controller able to drop low locality blocks and to retain high locality ones. Finally, in this case we can expect little performance improvements because high locality threads will experiment more hits in the LP-NUCA.

This paper extends LP-NUCA in several significant ways. First, we identify that LP-NUCA wastes dynamic energy during low locality phases by continuously degrading blocks among tiles and by accessing

¹Computer Architecture Group (gaz). Dpto. de Informática e Ingeniería de Sistemas. Instituto de Investigación en Ingeniería de Aragón. Universidad de Zaragoza. e-mail: {dario, vitor}@unizar.es

²Department of Computer Architecture. Universitat Politècnica de Catalunya (UPC). e-mail: teresa@ac.upc.edu

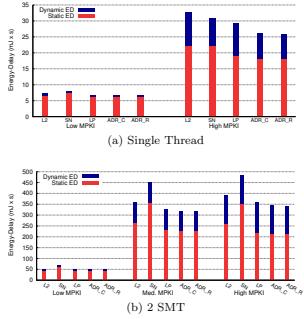


Fig. 6. Energy-Delay. This figure includes L3 cache consumption as well

change their number of ways, sets, or both at run time [13], [14], [15], [16]. For an updated state of the art please refer to Sundararajan *et al.* [16]. Previous works adapt the cache at a finer granularity than this work, and most proposed techniques can be easily applied to the LP-NUCA. Contrary to the original LP-NUCA design [4], this work proposes a proactive dynamic technique to save energy while previous ones, Sectoring and Miss Wave Stopping, were completely static and application agnostic. Besides, this work analyzes SMT workloads which have not been extensively studied.

Regarding the learning based approach, the Hill Climbing algorithm has been employed for distributing resources in SMT processors [17], but not for cache reconfiguration.

VI. CONCLUSIONS

Ultra-portable mobile devices demand quasi-desktop performance with a fraction of energy consumption. Since application behaviour changes during execution, processors require adaptive mechanism wasting the minimum amount of energy when necessary.

This paper proposes an adaptive controller for LP-NUCA, a tiled organization for high-performance low-power processors, that automatically decides when cache blocks are not reused and can be dropped reducing the cache activity. Besides, during high dropping phases, the controller is able to change the cache array access from parallel to serial further reducing the energy consumption.

With representative workloads, a cycle-accurate simulator, and implementation based energy estimations, we observe that the proposed controller reduces dynamic energy on average by 20% for single-thread and 2-threaded workloads without increasing the execution time.

ACKNOWLEDGEMENT

The authors would like to thank Luis Montesano del Campo for his helpful comments on the learning strategies. This work was supported in part by grants TIN2010-21291-C02-01 and TIN2007-60625 (Spanish Government), gaZT48 research group (Aragon Government and European ESF), Consolider CSD2007-00050 (Spanish Government), and HiPEAC-2 NoE (European FP7/ICT 217068).

REFERENCES

- [1] Shekhar Borkar and Andrew A. Chien, "The future of microprocessors," *Commun. ACM*, vol. 54, pp. 67–77, May 2011.
- [2] Changkyu Kim, Doug Burger, and Stephen W. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," in *Proc. of ASPLOS-X*, 2002.
- [3] Dario Suárez, Teresa Monreal, Fernando Vallejo, Ramón Beviéde, and Víctor Viñals, "Light NUCA: a proposal to bridge the inter-cache latency gap," in *Proc. of DATE'11*, 2011.
- [4] Dario Suárez, Gracia Giorgos Dimitrakopoulos, Teresa Monreal Arnal, Manolis G.H. Katevenis, and Víctor Viñals Yífera, "LP-NUCA: Networks-in-Cache for high-performance low-power embedded processors," to appear in *IEEE Trans. on VLSI Systems*, 2011.
- [5] Norman P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proc. of ISCA '90*, 1990.
- [6] Intel Embedded, "Intel® Xeon® processor C5500/C3500 series Datasheet - Volume 1," February 2010.
- [7] MIPS Technologies, "MIPS32® 1004K™ coherent processing system (CPS)" v1.0, 2010.
- [8] Tom R. Halfhill, "Netlogic broadens XLP family," *Microprocessor Report*, vol. 24, no. 7, pp. 1–11, 2010.
- [9] D.M. Tullsen, S.J. Eggers, and H.M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *Proc. of ISCA '95*, 1995.
- [10] Tom R. Halfhill, "The rise of licensable SMP," *Microprocessor Report*, vol. 24, no. 2, pp. 11–18, 2010.
- [11] LSI Corporation, "PowerPC™ processor (476FP) embedded core product brief," <http://www.lsi.com/DistributionSystem/AssetDocument/PPC476FP-PB-v7.pdf>, January 2010.
- [12] Yingmin Li, David Brooks, Zhigang Hu, Kevin Sladron, and Pradeep Bose, "Understanding the energy efficiency of simultaneous multithreading," in *Proc. ISLPED'04*, 2004.
- [13] David H. Albonesi, "Selective cache ways: on-demand cache resource allocation," in *Proc. of MICRO'32*, 1999.
- [14] Rajeev Balasubramonian, David Albonesi, Alper Buyuktosunoglu, and Sandhya Dwarkadas, "Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures," in *Proc. of MICRO'33*, 2000.
- [15] Chuanjun Zhang, Frank Vahid, and Walid Najjar, "A highly configurable cache architecture for embedded systems," *Proc. of ISCA '03*, 2003.
- [16] Karthik T. Raghavan, Timothy M. Jones, and Nigel Thorham, "Smart cache: A self-adaptive cache architecture for energy efficiency," in *Proc. of SAMOS'11*, 2011.
- [17] Seungryul Choi and Donald Yeung, "Hill-climbing smt processor resource distribution," *ACM Trans. Comput. Syst.*, vol. 27, no. 1, pp. 1–47, 2009.

Universidad Zaragoza
Instituto Universitario de Investigación de Ingeniería de Aragón
Universidad Zaragoza

A tiled instruction cache

A. Ferrerón, D. Suárez, and V. Viñals
Computer Architecture Group (gaZ) - University of Zaragoza, Spain
(ferreron, dario, victor)@unizar.es

Introduction

High-end embedded processors demand complex on-chip cache hierarchies with contradicting requirements:

- High performance operation
- Low energy consumption

Embedded processors' caches are responsible of an important part of the total power dissipation (e.g., the instruction cache of the ARM Strong is responsible of 27% of the total energy consumption [3]).

Specially critical in battery-based devices! How do we tackle the problem?

We proposed a tiled cache structure LP-NUCA [1] that:

- Increases performance in comparison with a conventional cache hierarchy
- Reduces the energy consumption

LP-NUCA has proved to be adequate for data cache hierarchies. We want to study the goodness of this design applied also to instruction caches in CMP and SMT contexts.

LP-NUCA Organization

The goal: to close the inter-cache latency gap

How? By enlarging the cache accessible by the processor at low latencies, exploiting temporal locality.

- without degrading bandwidth
- without requiring any complex change in the critical processor execution core

Key idea: use of three specialized interconnection networks.

- decoupling the functionality allows to simplify the hardware inside each tile: **we can access a tile in a single processor cycle**

The design has been verified by layout [1].

Results

ID-LP-NUCA for high-performance low-power CMP-SMT

New challenges, new opportunities

L3

Our objectives:

- Decrease: Area, Access latency, Energy consumption, Pressure on next level
- Increase: Performance, Capacity

We need to re-evaluate our design:

- New topologies, routing, and flow control
- Instructions and data share resources
- Coherence plays an important role
- New functionality and hardware: verify by layout!

A possible new-design:

How can we take advantage of this structure? Content allocation policies (hw/sw codesign)

	HW	Implementation	SW
Dynamic	ADR Controller	OS-based	
Allocation			User/Compiler-based
Static	Eviction Controller		

The use of NUCA cache hierarchies [4] is a well-known choice to reduce cache access latency. Migration policies may increase the performance of a conventional NUCA. However, in CMP contexts block migration policies turn to be complex [5]. Many proposals (e.g., [6, 7, 8]) use OS-directed placement to eliminate the migration-complexity problem.

LP-NUCA design and its specialized networks offer a wide range of possibilities:

	😊	😢
Eviction Controller (by restricting replacement)	<ul style="list-style-type: none"> Fairer distribution QoS 	<ul style="list-style-type: none"> Does not take into account the actual behavior of the program
Adaptive Drop Ratio Controller (ADR [2])	<ul style="list-style-type: none"> Transparent to the user Adapts to the behavior 	<ul style="list-style-type: none"> Energy and area overhead
OS-based	<ul style="list-style-type: none"> More flexibility in taking decisions 	<ul style="list-style-type: none"> Slower than hw approach
User/Compiler-based	<ul style="list-style-type: none"> Very interesting for RT systems 	<ul style="list-style-type: none"> Changing ISA

Conclusions and future lines

High-embedded systems = high-requirements demand: energy/performance trade-off.
We propose a tiled cache organization for instructions and data in coherent CMP - SMT designs.

Our main goal is to increase performance and reduce energy consumption; we need to re-evaluate our previous design.
Many opportunities for content allocation policies (hardware/software codesign).

References

[1] Suárez, D.; Dimitrakopoulos, G.; Monreal, T.; Katsenlis, M. G. H. & Viñals, V. "LP-NUCA: Networks-in-Cache for High-Performance Low-Power Embedded Processors". *IEEE Trans. on VLSI Systems*, 2011
[2] Suárez, D.; Monreal, T. & Viñals, V. "An Adaptive Controller to Save Dynamic Energy in LP-NUCA". *JWAP* 2011
[3] Montoro, J. et al. "A 160 MHz, 32-b, 0.5-W CMOS RISC microprocessor". *IEEE JSSC*, 1996.
[4] Kim, C., et al. "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches". *ASPLOS*, 2002.
[5] Beckmann, B. M. & Wood D. A. "Managing Wire Delay in Large Chip-Multiprocessor Caches". *IEEE MICRO* 27, 2004.
[6] Cha, S. & Jin, L. "Managing Distributed, Shared L2 Caches through OS-Level Page Allocation". *IEEE MICRO* 19, 2006.
[7] Hardavellas, N., et al. "Reducing NUCA: near-optimal block placement and replication in distributed caches". *ISCA* 36, 2009.
[8] Awasthi, M., et al. "Dynamic hardware-assisted software-controlled page placement to manage capacity allocation and sharing within large caches". *IEEE HPCA* 15, 2009.

Universidad Zaragoza
Sociedad Universitaria de Investigación de la Ingeniería de Aragón
Universidad Zaragoza

Behaviour Characterization of the PARSEC Benchmark Suite in the Processor's Memory Hierarchy

Marta Ortín, Jorge Albericio, María Villarroyo, Dario Suárez, Víctor Vilalta
Computer Architecture Group (gaZ), University of Zaragoza, Spain
{ortin.marta, j.alberic, maria.villarroyo, dario, victor}@unizar.es

Methodology

STEP 1 Choose metrics
 STEP 2 Design and execute the simulations
 Input → Benchmark → Simulator → Statistics
 STEP 3 Analyze the results

We choose which inputs to simulate so that the benchmark has the same behaviour as with the native input in relation to the chosen metrics

Introduction

Simulations are essential in the process of designing new computer architectures. We need detailed simulators and realistic workloads to get objective conclusions on which we can base our design decisions. The main obstacle we face is the extremely large simulation time.

Analysis of the Simulation Time - Simics+GEMS

We used two simulators: **Simics**[1] and **GEMS**[2] looking for bottlenecks which could be optimised:

- Most of the simulation time is due to the Ruby module
- The time is sparsely distributed among the functions

Any improvement wouldn't have had a major impact on the total simulation time

Impact of the Input Size on the Instruction Mix and Footprint

Instruction Mix

The proportion of loads and stores stays the same as we increase the input size

Footprint

Amount of memory touched by 50%, 90% and 100% of the data memory accesses.

Blackholes Fluidanimate

The benchmarks touch more memory data pages with bigger inputs

The benchmarks show strong locality

Impact of the Input Size on Memory Hierarchy for One Processor

Miss Rate

We obtained the miss rate for all the inputs of the benchmarks increasing the cache size

We don't necessarily get higher miss rates with bigger inputs

Miss Rate Pattern Detection

In some benchmarks we detect patterns directly related to the input parameters

Blackholes (Large Input)

Program Aware Input Selection

Method 1: Section of the native input	Method 2: Using a smaller input	Method 3: Using a new input
Example: Canneal	Example: Facesim	Example: Bodtrck
With bigger inputs we have higher miss rates Miss rate stays constant during execution We'll simulate a section of the native input Other benchmarks: blackholes, streamcluster, x264	The miss rate doesn't vary with input size, so we'll simulate the small input Other benchmarks: ferret, freqmine, swaptions, vips	We can establish a direct relationship between the miss rate pattern and the input parameters We'll simulate a small number of iterations of a complex problem Other benchmarks: fluidanimate, raytrace

On-Going Work: Ideas to Extend the Study to Multiprocessors

We have to take into account the coherence protocol
We can track the number of times each transition is used due to a request coming from a specific processor, which will allow us to identify different communication patterns
We'll use those values to compare the behaviour of the benchmark with each input

Example: We show the transitions in the shared cache caused by each processor in a producer-consumer communication pattern.

++ This transition is used many times
-- This transition is used very few times

Processor 1: Producer

Processor 2: Consumer

Conclusions and Future Lines

Simics+GEMS: We decided not to focus on optimizing the simulator because we observed there weren't any hot spots we could tackle to improve overall performance.
PARSEC: We discovered that, contrary to general belief, using bigger inputs does not necessarily mean they make a more efficient use of the memory hierarchy. This has allowed us to make a selection of the inputs to use so as to obtain results that are representative of the native input in a much more feasible simulation time.
We have already performed a full analysis executing the applications with one thread. We now want to focus on multithreaded execution and extend our conclusions to a multiprocessor environment.

References

- [1] P.S. Magnusson et al. Simics: A full system simulation platform. Computer, 35(2):50–58, feb 2002.
- [2] Mitra M. K. Martin et al. Multifacet's general execution-driven multiprocessor simulator (preliminary). SIGARCH Comput. Archit. News, 33:92–99, November 2005.
- [3] Christian Bienia et al. The parsec benchmark suite: characterization and architectural implications. In Proceedings of the 17th international conference on Parallel architectures and compilation techniques, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM.
- [4] C. Bienia et al. Fidelity and scaling of the parsec benchmark inputs. In Workload Characterization (ISWC), 2010 IEEE International Symposium on, pages 1–10, dec 2010.

Automatic Discovery of Performance and Energy Pitfalls in HTML and CSS

Adrian Sampson Călin Cașcaval Luis Ceze Pablo Montesinos Dario Suarez Gracia
University of Washington Qualcomm University of Washington Qualcomm Universidad de Zaragoza

Abstract—WebChar is a tool for analyzing browsers holistically to discover properties of HTML and CSS that lead to poor performance and high energy consumption. It analyzes a large collection of Web pages to mine a model for their performance based on static attributes of the content. An evaluation on two platforms, a netbook and a smartphone, demonstrates that WebChar can yield actionable conclusions for both content developers and browser implementors.

I. INTRODUCTION

Web browsing is an increasingly important part of the end-user computing experience. But the performance and energy consumption of Web technologies limits their growth in the mobile space where these factors are most crucial. Tools for understanding browsers' energy and performance characteristics are essential to creators of efficient Web content and to developers of mobile browsers.

Some studies have measured specific parts of the browser that are known to be bottlenecks, but Web technologies are complex, browsers change rapidly, and real Web content often uses the technologies in unexpected ways [1], [2]. Moreover, while JavaScript is amenable to traditional tools like performance profilers, fewer tools exist to analyze the declarative languages HTML and CSS—even though they represent a large portion of browsers' execution time [3]. A complete understanding of performance and energy on the Web requires empirical analysis of specific HTML and CSS implementations in the context of real-world usage. Web optimization tools should give broad, up-to-date answers to these central questions: What makes some Web pages slower than others? Why do some sites seem to guzzle battery life while others sip it?

This paper describes WebChar (for *Web characterization*), a model-mining system for analyzing browsers holistically. WebChar uses a large body of HTML and CSS content to build a model relating static page features to browser performance and energy consumption. It then mines this model to discover detailed, nonintuitive potential performance and energy problems reflective of common usage. Web content developers can use WebChar results as recommendations to avoid certain content-authoring pitfalls; to browser developers, WebChar is a tool for discovering new high-level optimization opportunities.

A prototype end-to-end WebChar implementation, along with an extended technical report, is available online at: <http://sampa.cs.washington.edu/sampa/WebChar>

II. APPROACH

The WebChar system consists of two main components. A data collection module takes snapshots of a large set of popular Web sites and measures the page load time and energy for each site. Then, the analysis step builds a model that predicts browser performance (or energy) based on page features. WebChar mines this model to produce a ranked list of likely expensive features.

A. Data Collection

The data collection tool downloads raw data from popular Web sites for analysis and measures the performance and power consumption of Web browsers while loading each page.

To obtain this input data, it would not suffice to download individual HTML documents from popular Web sites; instead, we need full *snapshots* of Web pages along with all linked linked content including embedded images and CSS stylesheets. A page snapshot includes all information necessary to replicate the experience of loading the page.

Using snapshots, the system collects performance or energy metrics for a particular browser, OS, and hardware setup. The measurement component consists of an HTTP server capable of replaying snapshots while simultaneously collecting timing and power data. From the browser's perspective, the replay server is indistinguishable from a "real" remote server; the HTTP responses are identical to those of the original host.

B. Analysis

Using the collected page data and performance measurements, WebChar's analysis component first summarizes the page data into a set of numerical *features*. It then correlates these feature values with the power and performance measurements to produce recommendations.

Features are Web page metrics that could potentially correlate with performance or energy usage. In other words, each feature is a candidate for identification as an "expensive" aspect of Web content. WebChar's features consist of metrics of each page's HTML and CSS abstract syntax trees, including tag, property, and selector frequencies. In all, the present implementation calculates 253 features per page.

The correlation step learns the relationship between a page's feature vector and its performance or energy consumption. Each page's feature vector and performance/energy data together constitute an example that WebChar uses to train its model. Any function estimation technique may be used to

produce this model; WebChar uses support vector regression. Once the model is trained, we apply an optimization heuristic, simulated annealing, to find a maximum of the trained function—a feature vector that leads to maximally poor performance or high energy consumption. The resulting optimized feature vector represents a set of feature values that, according to the trained model, result in bad browser behavior.

III. RESULTS

To evaluate WebChar, we analyzed data from 200 popular Web sites. We measured browsers on two systems: a mobile phone running Android 2.3.2 and an Atom-based netbook running Chromium 12.0.742.100. The phone's wireless communication hardware and display backlight were disabled to avoid measuring their power draw.

Using WebChar's output rankings, we distilled eight testable hypotheses regarding the performance and power consumption of HTML and CSS features. Of these, two hypotheses (the cost of images and CSS descendant selectors) reflected previously known performance pitfalls. We tested the remaining six hypotheses using microbenchmarks designed to exercise the feature that was hypothesized to be expensive. Of these, five effects were shown to be statistically significant and represent recommendations to browser and content authors:

- Laying out tables can be expensive on the Chrome desktop browser. Content developers should avoid placing content into tables where not strictly necessary.
- CSS opacity controls carry a significant performance impact, especially on the mobile browser we measured: translucent elements rendered 28% more slowly than opaque elements in our mobile phone tests. Web developers should avoid opacity effects; meanwhile, mobile browser developers should investigate using hardware-accelerated compositing for this feature.
- Across both platforms we measured, “floating” layout is expensive. Content developers should avoid it where possible; however, since modern Web page layouts frequently depend on these elements for their structure, browser implementors should optimize for this common pattern.
- Background fills cost significant energy on Android. Even when pages do not seem to load slowly on that platform, they may spend a large amount of energy drawing backgrounds. Mobile content developers should avoid using unnecessary background fills.
- The Android browser exhibits a performance and energy penalty when using HTML elements. The developers should investigate the element's unnecessary inefficiency as its presence does not affect the page's appearance.

Several of these findings are surprising and nonintuitive. The energy (but not performance) cost of element backgrounds, for instance, represents an unexpected discrepancy between performance and energy that can be exploited to conserve battery life on mobile devices. The cost of elements in the Android browser is also counterintuitive and is likely the result of a performance bug in the platform's implementation.

These unexpected results are where WebChar is most useful: the technique can identify costly factors without relying on human intuition.

IV. RELATED WORK

Existing performance- and energy-related resources for content developers generally consist of best practices provided by experts [4]–[6]. Tools like WebChar can make such best-practice advice more complete and allow it to grow and change along with the Web. Case studies [7] can also provide some insight into Web application workloads but do not scale to capture large and diverse portions of the Web.

Some work has focused on new implementation techniques for Web technologies [3], [8]–[11]. Even improved Web browsers, however, will likely exhibit hard-to-predict performance and energy pitfalls.

WebChar's approach is similar to analyses for distributed systems that use data mining techniques to debug failures [12] and pathologies [13]. WebChar treats browsers as complex black-box systems and, like the prior work, analyzes externally observable metrics to infer internal problems.

V. CONCLUSION

WebChar is a new technique for developing understanding of performance and energy in Web browsers, filling a crucial role in the improvement of the browsing experience on resource-constrained mobile devices. WebChar eliminates the slow process of manually identifying bottlenecks from among the thousands of features that browsers implement. As the Web continues to grow as a platform and as browser implementations continue to evolve, automated analyses like WebChar will be necessary to help identify new pitfalls when they appear.

REFERENCES

- [1] G. Richards, S. Lebresne, B. Burg, and J. Vitek, “An analysis of the dynamic behavior of JavaScript programs,” in *PLDI*, 2010.
- [2] P. Ramamurthy, B. Livshits, and B. G. Zorn, “JSMeter: comparing the behavior of JavaScript benchmarks with real web applications,” in *WebApps*, 2010.
- [3] L. A. Meyerovich and R. Bodik, “Fast and parallel webpage layout,” in *WWW*, 2010.
- [4] S. Souders, *High performance web sites: essential knowledge for front-end engineers: 14 steps to faster-loading web sites*, O'Reilly, 2007.
- [5] Yahoo, Best practices for speeding up your web site, <http://developer.yahoo.com/performance/rules.html>.
- [6] Google, Web performance best practices, http://code.google.com/speed/page-speed/docs/rules_intro.html.
- [7] S. Xu, B. Huang, J. Ding, and J. Dai, “Browser workload characterization for an Ajax-based commercial online service,” in *IISWC*, 2009.
- [8] O. Buyukkokten, H. Garcia-Molina, A. Paepcke, and T. Winograd, “Power browser: efficient web browsing for mobile devices,” in *CHI*, 2000.
- [9] J. Michal, J. Elsayed, J. Howell, and J. Lorch, “From: faster web browsing using speculative requests,” in *NSDI*, 2010.
- [10] M. Dong and L. Zhong, “Chameleon: a color-adaptive web browser for mobile OLED displays,” in *MobiSys*, 2011.
- [11] E. Fortuna, O. Anderson, L. Czer, and S. Eggers, “A limit study of JavaScript parallelism,” in *IISWC*, 2010.
- [12] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, “Pinpoint: problem determination in large dynamic Internet services,” in *DSN*, 2002.
- [13] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen, “Performance debugging for distributed systems of black boxes,” in *SOSP*, 2003.

Shrinking L1 Instruction Caches to Improve Energy–Delay in SMT Embedded Processors

Alexandra Ferrerón-Labari, Marta Ortín-Obón, Dario Suárez-Gracia,
Jesús Alastruey-Benedé, and Víctor Viñals-Yúfera

gaZ—DIIS—I3A, Universidad de Zaragoza, Spain
{ferreron,ortin.marta,dario,jalastru,victor}@unizar.es

Abstract. Instruction caches are responsible for a high percentage of the chip energy consumption, becoming a critical issue for battery-powered embedded devices. We can potentially reduce the energy consumption of the first level instruction cache (L1-I) by decreasing its size and associativity. However, demanding applications may suffer a dramatic performance degradation, specially in superscalar multi-threaded processors, where, in each cycle, multiple threads access the L1-I to fetch instructions.

We introduce iLP-NUCA (*Instruction Light Power NUCA*), a new instruction cache that substitutes the conventional L2, improving the Energy-Delay of the system. iLP-NUCA adds a new tree-based transport network topology that reduces latency and energy consumption, regarding former LP-NUCA implementations.

With iLP-NUCA we reduce the size of the L1-I outperforming conventional cache hierarchies, and reducing the overall consumption, independently of the number of threads.

1 Introduction

Superscalar execution cores demand a continuous instruction supply to feed their functional units. Delays due to instruction cache misses affect the instruction flow speed and instruction issue, and hence, performance. Simultaneous Multi-Threading (SMT) is a technique to hide long latency operations, such as cache misses, by the execution of several threads [1]. SMT aims to have all the functional units highly utilized by using a more powerful front-end (fetch unit) that supplies instructions from several threads. Consequently, the aggregated demand of instructions added by SMT makes on-chip instruction caches even more critical.

Instruction caches are responsible for a high amount of the energy consumption of the system. For example, StrongARM SA-100 and ARM 920TTM dissipate the 27% and 25% of the total power in the instruction cache, respectively [2],[3].

Ideally, we would like to have an instruction cache big enough to fit the footprint of the most demanding applications in order to increase their hit rate. However, bigger caches come at the expense of higher access latencies and higher energy consumption per access. Thus, there is a complex trade-off between size, on the one hand, and latency and energy consumption, on the other hand.

C. Hochberger et al. (Eds.): ARCS 2013, LNCS 7767, pp. 256–267, 2013.
© Springer-Verlag Berlin Heidelberg 2013

3. Segars, S.: Low power design techniques for microprocessors. ISSCC Tutorial note (February 2001)
4. Gwenmap, L.: What's inside the Krait. Microprocessor Report 26, 1–9 (2012)
5. Sundararajan, K.T., Jones, T.M., Topham, N.: Smart cache: A self adaptive cache architecture for energy efficiency. In: Proc. of the Int. Conference on Embedded Comp. Systems: Architectures, Modeling, and Simulation, pp. 41–50 (July 2011)
6. Zhang, C., Vahid, F., Najjar, W.: A highly configurable cache for low energy embedded systems. ACM Trans. Embed. Comput. Syst. 4, 363–387 (2005)
7. Bellas, N., Hajj, I., Polychronopoulos, C., Stamoulis, G.: Architectural and compiler techniques for energy reduction in high-performance microprocessors. IEEE Trans. on Very Large Scale Integration Systems 8, 317–326 (2000)
8. Kin, J., Gupta, M., Mangione-Smith, W.: The filter cache: an energy efficient memory structure. In: Proc. of the 30th Ann. IEEE/ACM Int. Symp. on Microarchitecture, pp. 184–193 (1997)
9. Suárez, D., Dimitrakopoulos, G., Monreal, T., Katevenis, M.G.H., Viñals, V.: LP-NUCA: Networks-in-cache for high- performance low-power embedded processors. IEEE Trans. on Very Large Scale Integration Systems 20, 1510–1523 (2012)
10. LSI Corporation: PowerPCTM processor (476FP) embedded core product brief (January 2010), <http://www.lsi.com/DistributionSystem/AssetDocument/PPC476FP-PB-v7.pdf>
11. Halfhill, T.R.: Netlogic broadens XLP family. Microprocessor Report 24, 1–11 (2010)
12. Byrne, J.: Freescale drops quad-core threshold. Microprocessor Report 26, 10–12 (2012)
13. Austin, T., Burger, D.: The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342. University of Wisconsin Madison (1997)
14. Muralimanohar, N., Balasubramonian, R., Jouppi, N.: Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0. In: Proc. of the 40th Ann. IEEE/ACM Int. Symp. on Microarchitecture, pp. 3–14 (2007)
15. Henning, J.L.: SPEC CPU2006 benchmark descriptions. SIGARCH Comput. Archit. News 34, 1–17 (2006)
16. Hamerly, G., Perelman, E., Lau, J., Calder, B.: SimPoint 3.0: Faster and more flexible program analysis. Journal of Instruction Level Parallelism (2005)
17. Suárez, D., Monreal, T., Viñals, V.: A comparison of cache hierarchies for SMT processors. In: Proc. of the 22nd Jornadas de Paralelismo (2011)
18. Wackerly, D., Mendenhall, W., Scheaffer, R.L.: Mathematical Statistics with Applications, 7th edn. Brooks/Cole Cengage Learning (2008)
19. Gabor, R., Weiss, S., Mendelson, A.: Fairness and throughput in switch on event multithreading. In: Proc. of the 39th Ann. IEEE/ACM Int. Symp. on Microarchitecture, pp. 149–160 (2006)
20. Li, Y., Brooks, D., Hu, Z., Skadron, K., Bose, P.: Understanding the energy efficiency of simultaneous multithreading. In: Proc. of the 2004 Int. Symp. on Low Power Electronics and Design, pp. 44–49 (2004)
21. Yang, C.L., Lee, C.H.: Hotspot cache: joint temporal and spatial locality exploitation for i-cache energy reduction. In: Proc. of the 2004 Int. Symp. on Low Power Electronics and Design, pp. 114–119 (2004)
22. Albonesi, D.H.: Selective cache ways: on-demand cache resource allocation. In: Proc. of the 32nd Ann. ACM/IEEE Int. Symp. on Microarchitecture, pp. 248–259 (1999)

Exploiting Data Compression to Mitigate Aging in GPU Register Files

Francisco Candel*, Alejandro Valero†, Salvador Petit*, Darío Suárez-Gracia‡, and Julio Sahuquillo*

Abstract— Nowadays, GPUs sit at the forefront of high-performance computing thanks to their massive computational capabilities. Internally, thousands of functional units, architected to be fed by large register files, fuel such a performance.

At nanometer technologies, the SRAM cells that implement register files suffer the Negative Bias Temperature Instability (NBTI) effect, which degrades the transistor threshold voltage V_{th} and, in turn, can make cells faulty unreliable when they hold the same logic value for long periods of time.

Fortunately, the GPU single-thread multiple-data execution model writes data in recognizable patterns. This work proposes mechanisms to detect those patterns, and to compress and shuffle the data, so that compressed register file entries can be safely powered off, mitigating NBTI aging.

Experimental results show that a conventional GPU register file experiences the worst case for NBTI, since maintains cells with a single logic value during the entire application execution (i.e., a 100% ‘0’ and ‘1’ duty cycle distributions). On average, the proposal reduces these distributions by 61% and 72%, respectively, which translates into V_{th} degradation savings by 57% and 64%, respectively.

I. INTRODUCTION

The role GPUs play in high-performance computing is growing in importance due to their excellent performance to energy ratio compared to conventional processors. For instance, the most energy-efficient supercomputers in the world, ranked in the Green500 list [2], include GPU devices.

GPUs are designed for improving system throughput, and their design is aimed at exploiting Thread Level Parallelism (TLP) by supporting the concurrent execution of a massive number of threads. The number of threads that a GPU can simultaneously execute exceeds, in several orders of magnitude, the number of hardware contexts supported by advanced processors like the IBM Power8 [22] or the Intel Knights Landing [23]. This feature is especially important for the execution of parallel scientific applications that rely on a high number of threads.

On the other hand, technology advances are allowing the semiconductor industry to implement fabrication nodes whose size is so small that process variations threaten system reliability. Process variations make transistors less reliable in low-power modes and intensify transistor aging phenomena, which affects any modern computing device, especially those that implement a large number of transistors, such as GPUs. In this regard, GPUs have dramatically evolved during the

last years to support massive numbers of threads. This implies that GPUs must incorporate huge register files to feed the computation performed by these threads. For example, the NVIDIA Tesla P100 (Pascal GP100) GPU features a 14MB register file, 3.5 times larger than its shared L2 cache (4MB) and several orders of magnitude bigger than CPU register files or CPU private L1 and L2 caches.

This work focuses on attacking aging in those transistors implementing the SRAM memory cells of GPU register files. In particular, the effect that most accelerates aging in SRAM cells is known as Negative Bias Temperature Instability (NBTI). NBTI degrades the threshold voltage V_{th} of the PMOS transistors that are on (i.e., their gate is connected to a logic ‘0’). In an SRAM cell, this happens when a logic value is stored for a relatively long period of time, known as duty cycle. In turn, V_{th} degradation slows down transistor switching time. Since NBTI can affect several transistors along the critical path of a digital circuit, it can cause operation faults if the critical path delay exceeds the clock cycle.

To avoid timing faults due to critical path delay growth, designers include guardbands (i.e., lower operating frequencies) [26], [15]. As a consequence, the maximum frequency for a given device decreases with time [6]. For instance, for a 45nm technology node, NBTI can cause a 25% performance degradation after 3 years [8]. Moreover, as technology scales down and aging intensifies, NBTI is becoming the principal source of transistor degradation [13], [24], compromising not only performance, but power consumption and area due to the transistor design margins required to compensate for NBTI [7].

A straightforward strategy for coping with transistor aging is powering off memory cells. Fortunately, when turned off, transistors not only stop degrading, but they partially recover from the NBTI effect [12]. Therefore, some works have attacked NBTI in GPUs by switching off those memory structures that are not used by GPU applications (kernels) [17], [8] or have severely aged [13]. Other works propose kernel compilation techniques based on aging information [15].

Unlike previous works, we exploit data redundancy in the GPU register file to mitigate the NBTI effect. This work makes two main contributions:

- A data compression/decompression mechanism, inspired by the BDI algorithm [19], which allows to switch off whole registers.
- A mechanism that rotates physical register addresses with the aim of evenly distribute switch-off cycles among all the registers.

These contributions can be applied to any modern GPU architecture either from NVIDIA or AMD, since i) both ven-

dynamically power off whole compute units [8], [13]. On the contrary, our fine-grain proposals only depend on register compression capabilities, which allow powering off registers in spite of being actually used by the kernel.

Lofit *et al.* [15] propose software techniques to recompile kernels from the point of view of the aging impact of each instruction type and the process variation characteristics of each hardware structure. This way, the most stressful instructions are distributed among those compute units less sensitive to degradation. Tan *et al.* [24] propose a technique that estimates at runtime the register latencies taking into account the additional delay induced by NBTL, and consequently rename register banks to extend their lifetime.

The compression algorithm proposed in this work is based on the BDI algorithm [19] for GPU caches. BDI calculates the arithmetic differences, Δ_s , between the first word and the rest within a cache line, storing in the same line the first word and all the obtained Δ_s . Contrary to this algorithm, our proposal calculates the differences between consecutive components of the register and only compresses those registers whose Δ_s are equal, resulting in a higher compression factor as just a single base component and a single Δ_s are required. BDI has been recently used to compress GPU register files [14] and caches [5] for energy saving purposes exclusively. These approaches store the base and all Δ_s in the same entry, while the unused cells are power gated. However, the remaining cells always stay on, so they are affected by NBTL throughout the kernel execution.

VII. CONCLUSIONS

Negative Bias Temperature Instability (NBTL) is the main deleterious effect that accelerates transistor aging over the lifetime of GPU memory structures. NBTL manifests when memory cells store a given logic value for an extended period of time, which in turn degrades transistor threshold voltage V_{th} and could eventually result in faulty operation. This work has enhanced the GPU register file design to reduce transistor aging caused by NBTL.

Data patterns showing similarities between the different register components have been analyzed. According to these observations, a data compression mechanism has been proposed to enable switching off entire registers, which induces a partial recovery from NBTL. In order to ensure an even aging among registers, a second mechanism that periodically rotates register addresses has been proposed, which distributes switch-off cycles along all registers.

Experimental results have shown that a conventional implementation of the register file maintains memory cells storing a given logic value during the entire execution of the applications, which implies 100% ‘0’ and ‘1’ duty cycle distributions. In contrast, the proposed mechanisms are able to reduce the longest ‘0’ and ‘1’ duty cycle distributions by up to 61% and 72%, respectively. These reductions cut by more than half the average transistor V_{th} degradation.

ACKNOWLEDGMENTS

This work was supported by the *Generalitat Valenciana* under Grant AICO/2016/059, by the *Gobierno de Aragón*

and the European ESF (gaZ-T48 research group), and by the *Ministerio de Economía y Competitividad* (MINECO) and AEI/FEDER (EU) funds under Grants TIN2016-75344-R and TIN2015-66972-C5-1-R.

REFERENCES

- [1] AMD Accelerated Parallel Processing (APP) Software Development Kit (SDK).
- [2] TOP500 Supercomputer Sites, available online at <http://www.top500.org/>.
- [3] Advanced Micro Devices, Inc. AMD Graphics Cores Next (GCN) Architecture, 2012.
- [4] Advanced Micro Devices, Inc. OpenCL Optimization Guide, 2013.
- [5] E. Atoufian. Compressed L1 Data Cache and L2 Cache in GPGPUs. In ASAP'16, pages 1–8, 2016.
- [6] K. Bernstein et al. High-performance CMOS Variability in the 65-nm Regime and Beyond. *IBM J. Res. Dev.*, 50(4/5):433–449, 2006.
- [7] B. J. Campbell et al. Leakage and NBTL Reduction Technique for Memory. US Patent No. 8395954 B2, 2013.
- [8] X. Chen et al. Run-time Techniques for Simultaneous Aging and Power Gating in GPUs. In DAC'14, pages 1–6, 2014.
- [9] N. Gong et al. Hybrid-Cell Register File Design for Improving NBTL Reliability. *Elsevier Microelec. Reliab.*, 52(9–10):1865–1869, 2012.
- [10] Intel Corporation. Intel® FPGA SDK for OpenCL. Best Practices Guide, 2017.
- [11] S. Kaxiras et al. Cache Decay: Exploiting Generational Behavior to Reduce Cache Leaking Power. In ISCA'28, pages 240–251, 2001.
- [12] N. Khoshan et al. Applicability of Power-Gating Strategies for Aging Mitigation of CMOS Logic Paths. In MWSCAS'13, pages 929–932, 2014.
- [13] H. Lee et al. Low-overhead Aging-aware Resource Management on Embedded GPUs. In DAC'14, pages 1–6, 2014.
- [14] S. Lee et al. Warped-Compression: Enabling Power Efficient GPUs through Register Compression. In ISCA'42, pages 502–514, 2015.
- [15] A. Lofit *et al.* Aging-Aware Compilation for GP-GPUs. *ACM TACO*, 1(2):1–20, 2015.
- [16] E. M. O’Farrell et al. Workload-Dependent NBTL and PBTI Analysis for a sub-45nm Commercial Microprocessor. In ICPDS, pages 1–6, 2013.
- [17] M. Namaki-Shoushtari et al. ARGO: Aging-aware GPGPU Register File Allocation. In CODES+ISS, pages 1–9, 2013.
- [18] NVIDIA Corporation. NVIDIA OpenCL Best Practices Guide, 2009.
- [19] G. Pekhimenko et al. Base-delta-immediate Compression: Practical Data Compression for On-chip Caches. In PACT'21, pages 377–388, 2012.
- [20] M. Powell et al. Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-submicron Cache Memories. In ISLPED, pages 90–95, 2000.
- [21] T. Siddiqua and S. Gurumurthi. Enhancing NBTL Recovery in SRAM Arrays Through Recovery Boosting. *IEEE TVLSI*, 20(4):616–629, 2012.
- [22] B. Sinharoy et al. IBM POWER8 Processor Core Microarchitecture. *IBM J. Res. Dev.*, 59(1):2:1–2:21, 2015.
- [23] A. Sodani et al. Knight Landing: Second-Generation Intel Xeon Phi Product. *IEEE Micro*, 36(2):34–46, 2016.
- [24] J. Tan et al. Mitigating Negative Bias Temperature Instability for GPGPUs. In IEEE TPDS, 27(11):3263–3277, 2016.
- [25] S. Thorayos et al. CACTI 5.1. HP Development Company, Palo Alto, CA, USA. Tech. Rep. HPL-2008-20, 2008.
- [26] A. Tiwari and J. Torrellas. Facelift: Hiding and Slowing Down Aging in Multicores. In MICRO-41, pages 129–140, 2008.
- [27] A. T. Tran and B. M. Baas. Design of an Energy-Efficient 32-Bit Adder Operating at Subthreshold Voltages in 45-nm CMOS. In ICCE-3, pages 87–90, 2006.
- [28] S. Ubal et al. Multi2Sim: A Simulation Framework for CPU-GPU Co-Design. In PACT'21, pages 335–344, 2012.
- [29] A. Valero et al. On Microarchitectural Mechanisms for Cache Wearout Reduction. *IEEE TVLSI*, 25(3):857–871, 2017.
- [30] R. Vattikonda et al. Modeling and Minimization of PMOS NBTL Effect for Robust Nanometer Design. In DAC'43, pages 1047–1052, 2006.
- [31] S. Wang et al. Low Power Aging-Aware Register File Design by Duty Cycle Balancing. In DATE, pages 53–59, 2012.
- [32] M. Zeng et al. Experimental Evaluation of Vector Instructions for GPGPU Performance, Energy Efficiency, and Opportunistic Reliability Enhancement. In ICS'27, pages 433–442, 2013.

Abstract Representation of Shared Data for Heterogeneous Computing

Tushar Kumar¹, Aravind Natarajan¹, Wenjia Ruan¹, Mario Badr²,
Dario Suarez Gracia³, and Calin Cascaval⁴

¹ Qualcomm Research: {tushark,narvind,wenjiar}@ti.qualcomm.com

² University of Toronto: mario.badr@mail.utoronto.ca

³ Universidad de Zaragoza: dario@unizar.es

⁴ Barefoot Networks: cascaval@acm.org

Abstract. Data management across address spaces in heterogeneous platforms represents a significant performance bottleneck and energy cost for applications, particularly on mobile System-on-Chip (SoC). We propose a light-weight middleware layer to regulate concurrent access to shared data in a Heterogeneous SoC. Our approach uses acquire-release semantics to provide the following benefits: *i)* enable high-level heterogeneous programming frameworks to easily maintain consistent non-device, non-platform-specific data abstractions for programmers, and *ii)* provide an abstract memory interface with strong analyzable properties about the correctness and performance of the synchronization operations across memory-types. These benefits are achieved while retaining the ability to plug-in arbitrary types of heterogeneous memory frameworks and to enable platform-specific and inter-framework synchronization optimizations. We demonstrate that our approach avoids paying the “abstraction cost” and compares favorably with manually optimized OpenCL, while providing a simpler and understandable API.

Keywords: heterogeneous System-on-Chip, memory synchronization, memory concurrency, data sharing

1 Introduction

Heterogeneous computing systems allow programmers to match parts of an application to the strengths of the different devices available [14]. The ultimate goal of heterogeneous computing is to obtain higher performance at lower power by judiciously balancing the computation. Prior work has focused on partitioning applications across heterogeneous devices. For example, the Fast Multipole Method has been shown to work better on a CPU-GPU architecture [10]. However, heterogeneous systems are not limited to CPUs and GPUs. As we scale to a more diverse set of accelerators, a major impediment to programmers becomes moving data across devices. Mobile Systems-on-Chip (SoCs) typically share data across devices using a contiguously-allocated block of memory (i.e., a buffer) that is modifiable by one device at a time. Without advanced hardware support [2]

- SLAMBench, a performance and accuracy benchmarking methodology for slam. In *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pages 5783–5790, May 2015.
16. R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohi, J. Shotton, S. Hodges, and A. Fitzgibbon. KinectFusion: Real-time dense surface mapping and tracking. In *Mixed and Augmented Reality (ISMAR), 2011 10th IEEE International Symposium on*, pages 127–136, Oct 2011.
 17. Norman P. Jouppi *et al.* In-datacenter performance analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA ’17*, pages 1–12, New York, NY, USA, 2017. ACM.
 18. OpenCL, the open standard for parallel programming of heterogeneous systems. <http://www.khronos.org/opencl>.
 19. The OpenMP API specification for parallel programming. <http://www.openmp.org/>.
 20. Prasanna Pandit and R Govindarajan. Fluidic kernels: Cooperative execution of opencl programs on multiple heterogeneous devices. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 273. ACM, 2014.
 21. Qualcomm Snapdragon. Qualcomm Technologies Inc. <https://www.qualcomm.com/products/snapdragon>.
 22. Dave Shreiner and The Khronos OpenGL ARB Working Group. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1*. Addison-Wesley Professional, 7th edition, 2009.
 23. Heterogeneous computing made simpler with the Symphony SDK. <https://developer.qualcomm.com/blog/heterogeneous-computing-made-simpler-symphony-sdk>.
 24. Qualcomm Symphony System Manager SDK. <https://developer.qualcomm.com/software/symphony-system-manager-sdk>.

A Proposal to Introduce Power and Energy Notions in Computer Architecture Laboratories

Alicia Asín Pérez^{*}
Libelum Comunicaciones Distribuidas
Ctra. María de Luna 11
E-50018 Zaragoza, Spain
a.asin@libelum.com

Darío Suárez Gracia, Víctor Viñals
Yúfera
gaZ - DIIS - I3A - Universidad de Zaragoza
Ctra. María de Luna 1
E-50018 Zaragoza, Spain
(dario, victor)@unizar.es

ABSTRACT

Power has emerged as a major concern in the microprocessor industry. From embedded to high-performance processors, all designs employ several power optimizations at the circuit and the architectural levels.

While introductory computer architecture books and courses are starting to cover power concepts, proposals to offer students a practical experience with power issues are still scarce. To do so, we advocate the inclusion of energy and power concepts in computer architecture courses by means of laboratory experiments. These experiments build upon concepts presented in preceding physics and/or electronics courses.

This paper outlines our experience with the development of such hardware-based energy laboratories. We propose experiments on a simple, yet powerful hardware-software platform capable of live energy measurements in a desktop computer processor. The proposed laboratory setup can help to teach students the basics of power-aware computer architectures. The performed experiments demonstrate the viability of our approach. For example, our experiments show that students can deduce the dynamic and static power dissipation of the Intel Pentium 4. Information that is not documented in the processor's datasheet.

Categories and Subject Descriptors

K.3.2 [Computer and Information Science Education]: Computer science education

General Terms

Experimentation, Measurement, Performance

^{*}This work was performed while Alicia Asín Pérez was at the University of Zaragoza.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

WCAE'07 San Diego, California USA

Copyright 2007 ACM 978-1-59593-797-1/07/0006 ...\$5.00.

Keywords

Power, Energy, Processor, Platform, Measurements, Computer Architecture, Education

1. INTRODUCTION

Technology scaling has permitted an outstanding growth pace in microprocessor performance. Smaller technologies make the transistors both faster and smaller allowing more of them to be integrated on the same die area. The integration of millions of transistors in a few square centimeters results into a significant increase in power density. This power has to be dissipated [16]. To address this challenge, energy and power have to be considered from the very beginning of the microprocessor design. Accordingly, it is necessary for undergraduate education to expose our students to these concepts. However, undergraduate students are often unaware of energy and power concepts after completing their computer architecture courses. Neither the joint ACM & IEEE Computing Curricula 2001, Computer Science, nor the Computer Engineering 2004 Curriculum guidelines include any reference to energy and power in computer architecture courses [11, 12]. Even many of our students have forgotten the physics concept of current or voltage when they arrive to the computer architecture design courses.

A typical computer architecture course syllabus is quite large. Instead of removing existing concepts from the curriculum to introduce energy and power, we propose to do so in a laboratory setting. Via experiments students can gain insight on the trade-offs amongst processor architecture, compilation, and energy consumption. Students could understand the impact that power can have on a design and basic concepts relating to power and energy. For example, consider two different processors for a battery operated device. Even if one of them consumes less power than the other, i.e., smaller average power, it may not be the best choice. If it spends more time to perform the same task, it may consume more energy reducing the operation time of the device.

These trade-offs could be easily understood with real-life experiments where students carry out energy measurements. Also, laboratory experiments can allow students to understand the energy-optimizations incorporated in modern processors, such as the Intel Core DUO whose microarchitecture implements many power-saving techniques [10].

Traditionally, CMOS transistors expended energy primarily only while switching. The power loss while idle was negligi-

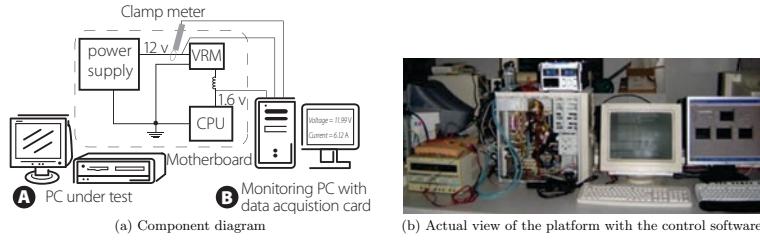


Figure 1: Hardware Platform with a liquid cooling system.

ble. For feature sizes below 100 nm this is no longer true. Transistors waste considerable energy even when idle. Static consumption increases for two reasons. First, there are many more transistors on the same area. Second, to keep dynamic power dissipation at bay, the voltage supply is typically reduced leading to subthreshold leakage. Often large portions of the on-die transistors are idle. For example, if there is not enough parallelism to exploit some functional units remain idle only wasting static power. Large, higher-level caches are also another source of static consumption because they are seldom accessed and they can not be easily turned off without losing their content. Accordingly, the introduction of static power in the laboratories was another important issue for us. Therefore, one of the proposed experiments computes the dynamic and static consumption breakdown of an Intel Pentium 4 with the method suggested by Sinha *et. al.* [15].

The main contributions of this paper are:

- We describe a simple platform to perform energy-related laboratory experiments for computer architecture courses.
- We suggest a set of challenging experiments that allow students to understand the energy impact of several factors, such as processor frequency, program data types or compiler optimization level.
- We present results of our analysis about the static consumption of a contemporary high-performance microprocessor.

Most part of this work was developed as part of Alicia Asín Pérez's Master Thesis on Computer Engineering at the University of Zaragoza.

Section 2 comments on previous work. Section 3 describes the experimental setup. Section 4 describes the set of laboratories we propose. Section 5 summarizes the work.

2. RELATED WORK

Many studies have measured voltage and current for high-performance processors, mostly for validating their processor energy characterization. For instance, Isci *et. al.* developed a methodology and its corresponding platform to obtain live power measurements [6]. Hu *et. al.* describe an infrastructure to characterize program power behavior

and validate it against physical measurements on an Intel Pentium 4 [4].

Sinha *et. al.* proposed a method to isolate the switching and leakage energy components [15]. Their method is based on the observation that the dynamic energy consumed by a program is independent of the execution frequency. They computed the leakage current of an ARM embedded processor and studied the energy consumed by each instruction and by the operating system.

But, to the best of our knowledge no previous work has been concerned with setting up a measurement platform in order to allow students to acquire practical insights on energy and power concepts.

3. PLATFORM DESCRIPTION

This section describes the platform developed to test the feasibility of energy-related laboratories for computer architecture courses. The proposed platform is simple and robust. Also, it is built using commodity materials that are typically available to any Computer Science or Engineering department.

3.1 Hardware

We chose an Intel Pentium 4 because of the Intel SpeedStep technology which allows us to vary the processor frequency. As Section 4.1 explains, this capability was very useful during the experiments because it allows us to execute the same program at different frequencies without rebooting the machine. Moreover, the Pentium 4 includes a large set of hardware performance counters, useful both for verifying the results and for obtaining indirect energy measurements, such as Watts per instruction. We access the performance counters via the Performance Application Programming Interface, PAPI, which provides a consistent access interface [2]. Accordingly, we believe that the software component of our laboratories can be re-used with future processors.

Most motherboards for the Intel Pentium 4 processors use a direct and isolated power line between the power supply and the voltage regulation circuit which feeds the processor. So, the product of the voltage of these wires times their current is the power consumed by the processor assuming that the consumption of the voltage regulation manager is negligible compared to the processor one [5, 1].

Figure 1 shows the energy measurement platform. Figure

1(a) shows a simplified connection schematic. Two PCs, A and B, are used to get the measurements. A executes the program under test and B visualizes and saves the acquired data. A clamp meter measures the current drawn by the processor. This ammeter reads the current in the two 12 Volts lines which go from the power supply to the Voltage Regulation Manager (VRM) [5]. Also, the voltage on these wires is sent to the data acquisition card on the second PC, B. With these two values it is possible to get the power consumed by the processor. We also monitor the voltage in the coils at the exit of the VRM, in order to get the exact operational voltage of the processor and to check that variations in frequency do not imply lower supply voltages.

In all the experiments, we collect 500 samples per second, which is the maximum rate of our low-cost data acquisition card (Adlink PCI-9112).

Sub-threshold leakage current depends on the temperature [18, 7]. In order to reduce the temperature variability we used a liquid refrigeration system. With the Thermaltake Aquarius III, A1681, the CPU temperature ranges between 29 and 33 C among the tested frequencies, making the impact of temperature in the results negligible. Figure 1(b) shows the actual view of the platform with both computers and the clamp meter located over the table on the bottom of the PC under test.

3.2 Software

The selected operating system was GNU/Linux for several reasons. Linux is supported by PAPI. By only installing a few kernel modules it is possible to vary the frequency without rebooting the machine with utilities such as `cpufrequtils`. To support software frequency scaling, in a system with an Intel Pentium 4, the following modules are required: `speedstep/lib`, `freq_table`, `p4_clockmod`, and `cpufreq_userspace`.

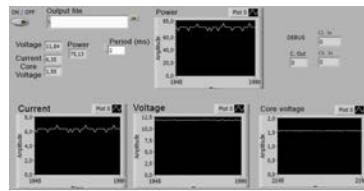


Figure 2: Control software screenshot

In order to reduce the load on the computer under test, A, all non-necessary services like window server or printing services have been removed. During the experiments the measured program uses the CPU for more than 99.9% of the time.

The acquired data is monitored in real time with a LabView application. LabView is a well-known software for visual control and data analysis [9]. Figure 2 shows a screenshot of the application. The interface is clean and intuitive. In the upper part, it shows the instant values for current and voltage at both, the VRM input and at the processor. Also, it shows the sampling period which can be dynamically changed up to the minimum value of 2 ms. The medium

top graph plots the live evolution of the power consumed by the processor and makes it easy to detect phases in program execution. In the lower part, the different plots show the evolution of the three measured quantities, voltage and current for the VRM and only voltage for the CPU, core voltage.

The application also allows us to save the acquired results in order to analyze them offline.

In summary, the proposed platform enables to acquire and store the power measurement consumed by a desktop computer during large intervals of time.

4. EXPERIMENTS

To demonstrate the capabilities of the platform, we performed two different laboratory experiments. We estimated that it would take less than 2 hours for the students to complete them. The first experiment, which is the most complex one, describes how to isolate the dynamic and the static energy components. The second shows an introduction to the interaction between compilation and energy consumption.

4.1 Dynamic and Static energy

Energy consumption in sub-micron technologies comprises two main components, dynamic and static [18]. The former, is dissipated from the charge and discharge of capacitances when the transistors switch state. Static power is dissipated from multiple leaky sources, especially sub-threshold and gate leakage [7]. It wastes power while the device is on even if idle. For example, L2 and L3 on-chip caches are components that waste a lot of static power because they are dense, they contains a lot of transistors, and depending on the executed coded they are rarely accessed.

Processor datasheets like the ones from Intel Pentium 4 or AMD Opteron do not distinguish between these two components and normally only show the maximum power consumption. Based on the previous work of Sinha et. al. [15], we propose a laboratory to compute the dynamic and static consumption of a desktop microprocessor. We make the simplifying assumption that there are no static bias currents in the microprocessor. The total power consumption of a program is given by

$$P_{tot} = P_{dyn} + P_{sta} = C_L V_{dd}^2 f + V_{dd} I_{leak} \quad (1)$$

where P_{tot} is the total sum of the dynamic and static power. The dynamic power, P_{dyn} , is the product of the average capacitance switched per cycle, C_L , times the square of the supply voltage, V_{dd} , times the frequency, f . The static power is the product of the supply voltage times all the leakage currents, I_{leak} . If the total execution time of a program is Δt , the energy consumed by the program is

$$E_{tot} = P_{tot} \Delta t = C_{tot} V_{dd}^2 + V_{dd} I_{leak} \Delta t \quad (2)$$

where C_{tot} is the total capacitance that has been switched across all execution cycles.

From (2), it follows that the dynamic component is independent of the execution time. Therefore, if we execute the same program multiple times at different frequencies keeping the same supply voltage, the dynamic energy will remain constant. The number of executed instructions and transistor switchings should be the same. But the static component will vary linearly with regards to the execution time. With a few executions at different frequencies an approximation of the leakage current, I_{leak} , can be estimated.

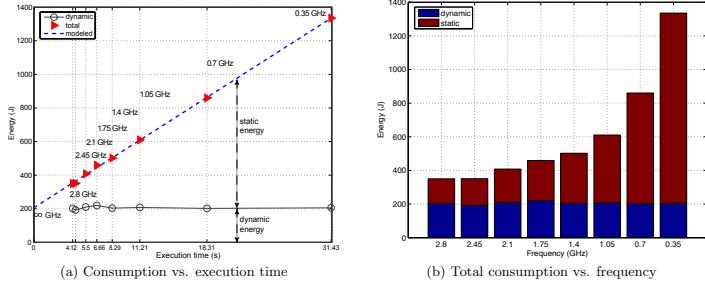


Figure 3: Static and dynamic energy breakdown

The execution time of a program, Δt does not only depend on the frequency. The execution time of the program is

$$\Delta t = Instr \times CPI \times T_c \quad (3)$$

Where, $Instr$ represents the number of executed instructions, CPI , the average number of cycles per instruction and T_c , the cycle time which is the inverse of the frequency, f . So, it is mandatory that CPI does not vary for ensuring that execution time only depends on processor frequency.

Since Linux is a multi-tasked Operating System and the Pentium 4 includes multiple cache levels, we use the performance counters to verify this assumption. Also, the benchmark choice is very important. If we ensure that code and data are small enough to fit in the first level caches, we reduces the chance of variations in the CPI due to accesses to main memory. We tested a classical LU factorization in which more than 95% of the execution time comes from one loop. For these experiments, the matrixes have a size of 20×20 , and the number of iterations was fixed to one million. The LU factorization and all the benchmarks used in this paper were compiled with gcc 3.4.3 at the maximum optimization level (-O3).

The behavior of the first level data cache is an interesting prerequisite for the laboratory. The student should prove that the working set does produce neither capacity nor conflict misses. Otherwise, the measurements could be invalid. After program execution, they can verify the validity of their results with the hardware counters. In our case, we read them to check cache misses, tlb misses, total cycles, and retired instructions. For the results shown in this paper, we also repeat several times the experiment to check that the variation in the execution time and CPI is very small.

Figure 3 shows the measurements and the computed static and dynamic energy. We ran the program for all the available frequencies, from 350 MHz to 2.8 GHz. The execution time varies between 31 s and 4.12 s, for 350 MHz and 2.8 GHz, respectively. In Figure 3(a), the triangular marks are the measurements of the consumed total energy at different frequencies. A linear regression fits well the experimental data. From Equation (2) we know that the slope of this line represents the static power, 35.94 W. Multiplying this value

times the execution time we get the static energy consumption for each execution. By subtracting this static component from the total energy, we get the dynamic component (line with circles). The resulting values also fit with the constant term of the linear regression, around 200 J.

Since P_{sta} and V_{dd} are 35.94 W and 1.6 V, respectively, the leakage current, I_{leak} , is 22.46 A for our 130 nm Intel Pentium 4.

From Figure 3(b), at maximum frequency, the static consumption represents a 42.3 % of the total, a reasonable value. This can be partially explained from the fact that Intel used very short channel length transistors, 60 nm for 130 nm technology, and the sub-threshold current is inversely proportional to the channel length [17, 13]. When the frequency decreases to 350 MHz, static consumption represents more than 80% of the total.

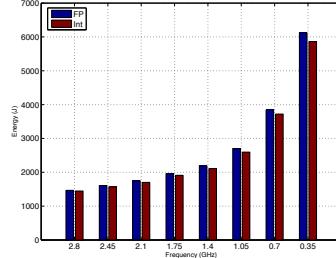


Figure 4: Total energy for the LU factorization with floating point and integer data.

Another interesting experiment entails changing the data type definitions of the LU matrixes from 32-bit floats to 32-bit integers, trying to do a similar work but with different

execution units. Figures 4 and 5 show the original experiment (left bars), and the new one (right bars). The total energy values change because the number of iterations increase to five millions.

Figure 4 shows the total energy consumed. The floating point version wastes more energy because floating point arithmetic units are more complex and slower than integer ones. The first factor raises both dynamic and static consumption. The larger number of transistors results into more number of transitions and leaky devices. The longer execution time affects the static consumption. Figure 5 shows the average power for the same experiment. In this case, at higher frequencies, the integer version has a higher average power. This larger average power comes from the Pentium 4's tight core, consisting of the integer ALUs, their bypass paths, and the schedulers, which run at double the frequency compared to the rest of the core [14]. The difference decreases when we reduce the frequency because at lower frequencies the static power dominates. As shown in Figure 3(b), the static consumption represents a 84.6% of the total at 350 MHz.

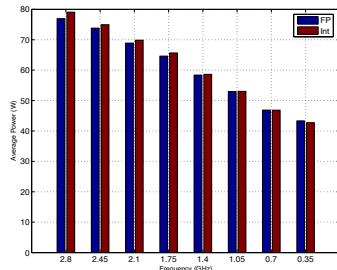


Figure 5: Average power for the LU factorization with floating point and integer values.

This short experiment could help the students to understand the differences between power and energy. In this case and for frequencies range between 1.4 and 2.8 GHz, the lower power version, floating point, wastes more energy than the higher power one, integer. Following this example, we can devise a family of experiments aimed at comparing the energy/power required to perform the same amount of program work. These include changing the data type precision, 32 bits vs. 64 bits, using packed arithmetic on SSE units, or using library emulation to perform floating point computation on integer units.

4.2 Effects of compilation in energy and power

Studying the interaction between the compiler and the energy/power consumed by the processor is another appealing application of the platform.

A simple experiment is to compile the same program at different compiler optimization levels and see how this affects energy and power. We present results for two SPEC 2K benchmarks, mcf and wupwise, which are integer and

floating point, respectively [3]. Since the laboratory time is limited and the reference data set takes a long time to execute, we have used the MinneSPEC input set [8].

In this experiment, the processor frequency remains constant at 2.8 GHz. We focus on total energy and average power. Figure, 6(a), shows the total energy. For both programs, the more aggressive the compilation, the lower energy. In mcf, the difference is smaller than in wupwise where going from -O1 to -O2 achieves a 20% energy reduction.

Figure 6(b) shows the power results. For mcf the average power is reduced when the compilation level is increased but in wupwise it remains the same. Mcf maximum power, 86W, is achieved at -O2 compilation. The average power does not vary significantly amongst the three different optimization levels in wupwise. But the maximum power increases to 91 W with -O3.

5. CONCLUSIONS AND FUTURE WORK

This paper summarizes our experience in studying the feasibility of introducing energy-related laboratories in computer architecture courses. It describes a platform to obtain live measurements of voltage and current of the processors and to visualize them in real time. The platform is able to save the acquired data to realize more complex subsequent analysis.

We describe the experimental setup and several simple experiments to show some of the platform applications. Many other possibilities exists, including more specialized studies on the efficiency and role of the cooling device. Breaking down the dynamic and static components of the total consumption is a very interesting experiment that discloses a fact not included in the public datasheets. The results shows that for the LU matrix computation, the static energy represents the 42.3 % of the total at the maximum frequency, 2.8 GHz, for a 130 nm Intel Pentium 4.

Since power is a real concern in computer architecture, we consider that its gradual inclusion in computer architecture courses will be beneficial. After finishing this project, we believe that an energy laboratory within the computer architecture courses taught at the University of Zaragoza will be useful for the students and we will try to introduce it during the following years.

Acknowledgements

We gratefully acknowledge Patrick Akl, Andreas Moshovos, and Jason Zebchuk for their helpful suggestions and comments on a prior version of this paper. Also, we would like to thank the anonymous reviewers for their comments.

This work has been supported by the Diputación General de Aragón grant "Grupo Consolidado de Investigación" (BOA 20/04/2005), the Spanish Ministry of Education and Science grant TIN2004-07739-C02-01/02, and the European Union Network of Excellence HiPEAC (High-Performance Embedded Architectures and Compilers, FP6-IST- 004408).

6. REFERENCES

- [1] Analog Devices. *ADP3180, 6-Bit Programmable 2-, 3-, 4-Phase Synchronous Buck Controller*. Analog Devices, 2003.
- [2] J. Dongarra, L. London, S. Moore, P. Mucci, and D. Terpstra. Using papi for hardware performance

Processor Energy and Temperature in Computer Architecture Courses: a hands-on approach

Sergio Gutiérrez-Verde Octavio Benedí-Sánchez

Darío Suárez-Gracia José María Marín-Herrero[†] Víctor Viñals-Yúfera

gaZ. Dpto. de Informática e Ingeniería de Sistemas

† Gitse. Dpto. de Ingeniería Mecánica

I3A–Universidad de Zaragoza

C\ Marfa de Luna 1. E-50018 Zaragoza, Spain

<http://webdiis.unizar.es/gaz/>

Abstract

Performance has driven the microprocessor industry for more than thirty years. Its effort has enabled to multiply by several orders of magnitude the computational power; e.g., the Intel 8080 was able to execute 0.64 MIPS and the newest Core i7 can execute 6400 MIPS. The cost of this fabulous improvement has been a large rise in energy consumption. Nowadays, we have reached a point where one of the most limiting factor for improving performance is energy dissipation.

In order to keep the performance improvement during the next years, it is necessary to study energy and temperature in depth. Nevertheless, most current computer architecture curricula include neither energy nor temperature.

The lack of adequate experimental platforms contributes to the difficulty in teaching these topics. In this paper we propose a possible solution: to instrument a commodity PC for measuring the processor power and temperature during the execution of real programs. The platform is devised for teaching, but it can be used to support research experiments as well. For example, we describe an interesting undergraduate laboratory that analyzes the interaction between compiler optimizations and energy. With this laboratory, students can learn that performance optimizations usually reduce energy but may increase power.

1 Introduction

Recently, designing energy-efficient computers or reducing energy consumption is going beyond marketing strategies or personal experiences to turn into a collective goal for governments, societies, or companies. For instance, Green Computing advocates for an environmentally sustainable

computing and communication, with minimal or no impact on the environment. Together with the concepts of total cost of ownership, including the cost of disposal and recycling, the economics of energy efficiency is a key point of Green Computing. So we think that computer engineers should be aware of these issues.

Energy-efficient computers are not only important from a Green Computing perspective, but also from a pure performance point of view. On one hand, in the embedded domain, lowering the energy consumed by the processor increments the device uptime. On the other hand, in the commodity segment, the cooling system affects performance when it is not able to dissipate all the generated heat and forces a processor frequency/voltage reduction.

While the study of many design constrains, such as performance or programmability, may be done by means of white boards or simulators, evaluation of energy and temperature appeals for hands-on laboratories—where students deal with real—for many reasons such as: 1) This approach reinforces their physics background and establish a clear connection between computer architecture and its implementation; 2) They will quickly learn the importance of energy dissipation and temperature by watching for example how fast a processor shutdowns when its fan stops; 3) Energy and temperature simulations require sophisticated environments for being accurate, and since energy depends on both the instructions and their data, the simulation time can be very high and unfordable in two/three hour lab sessions.

The main barrier this hands-on approach faces is the lack of well established platforms for carry on the measurements. Many authors have performed processor power measurements either research oriented such Isci *et al.* or academic oriented like Asfn *et al.* [3]. Others such us Mesa-Martínez *et al.* have measured temperature in commodity PCs [17]. But up to our knowledge there is not an ade-

quate platform able to simultaneously measure both magnitudes. The present work extends the Asín *et al.* platform adding temperature monitoring support and automatic synchronization of the sampling process. The resulting platform improves measurement accuracy and data logging capabilities, and at the same time its academic capabilities such ease of use or cost are reinforced.

Platform features are presented by means of a laboratory intended use case for last year undergraduate or master courses. Our final goal is to use this platform with students from both Computer Engineering (Computer Architecture courses) and Mechanical Engineering (Heat Transfer courses) degrees in our institution to make them working together in a common problem. As a session suitable for both kind of students we present a lab dealing with the interaction between compiler optimizations and energy.

Summarizing, the contributions of this work are the following: we improve an existing platform for measuring energy and temperature in commercial processors extending its logging capabilities and improving the sampling accuracy. We present the potential of the platform with an interesting laboratory in which the relation of power and temperature and the impact of compiler optimization in energy and power are analyzed.

This paper is organized as follows. Section 2 comments on the related work. Section 3 describes the measurement platform in detail. Section 4 explains some test for validating the platform. Section 5 describes the example laboratory. Section 6 concludes and present some possible future work lines.

2 Related Work

Energy and temperature have aroused the interest in both industry and academia. In the industrial side, SPEC has introduced SPECpower_ssj2008 focusing on server computer consumption [6], and EEMBC has defined EnergyBench establishing a framework for adding energy to the metrics of the EEMBC's performance benchmarks [5].

Many studies have been conducted in the academic side. Regarding energy, Isci and Martonosi describes a methodology for obtaining per-unit power estimations combining real power measurements with performance counters [14]. Other authors have proposed infrastructures based on an Intel Pentium 4 for characterizing program phases, evaluating compiler optimizations, or studying energy [9, 21, 3].

Temperature measurements have been performed with more sophisticated setups; e.g., Mesa-Martinez *et al.* have presented some power and temperature estimations using an expensive IR thermal imaging equipment [16].

While most previous work focuses on energy and temperature from a research perspective, our work also takes

into consideration academia requirements such as simplicity or affordable cost.

3 Platform description

The measurement platform is based in our previous work and consists of two commodity PCs [3]. One, named computer under test (CUT), is monitored, and another, named data acquisition and storage computer (DASC), acquires and saves all the power and temperature samples gathered from the CUT. Both computers are shown in Figure 1a, the CUT in the left and the DASC in the right.

The CUT runs a GNU/Linux system with a 2.6.25 kernel in which all non-required modules and services (X-Windows, printing, USB, ...) have been removed to minimize the energy consumed by the operating system tasks. The processor and the motherboard are a 2.8 Ghz Intel Pentium 4 *Northwood* and an ASUS P4 P8000, respectively. This motherboard employs a dedicated power line between the power supply and the processor voltage regulator manager; thus, it removes the need of hacking the motherboard and simplifies the monitoring of the processor consumption because the product of the voltage of the VRM power line times its current is the power drawn by the processor—assuming negligible the VRM consumption [2]. The above described power line is present on most current PCs, so this technique can be used with other hardware configurations.

The current is measured with a Tektronix TPC-312 clamp ammeter [23]. The output of the clamp ammeter along with the voltage are logged with an Adlink PCI-9112 [10] data acquisition card sampling at 2 Kilosamples/second per channel, 1000x more than the previous version of the platform. At this sampling rate, we are able to observe the main program execution phases, and power traces remain in reasonable sizes, lower than 1 GiByte. All samples are stored in the DASC in order to allow off-line analysis. The DASC system also runs GNU/Linux and the previous LabView software has been replaced by C based code and some perl scripts because they allowed much higher sampling rates and we observed that the real-time visualization of LabView was seldom used. In fact, real-time visualization is useful for debugging the platform, but for that purpose an oscilloscope is preferable. The use of the new programs is straightforward with a small learning time as it was with the LabView based software.

Current processors require large heat sinks with powerful fans for cooling. Cold air flows towards the processor pushed by the fan and gets warmer. The hot air is expelled through the sides of the head sink as shown in Figure 1b. Since the air (a fluid) flows through a solid (each of the narrow channels in between the parallel fins), the whole processor cooling package could be modeled according to a forced-convection thermal model. If some conditions are

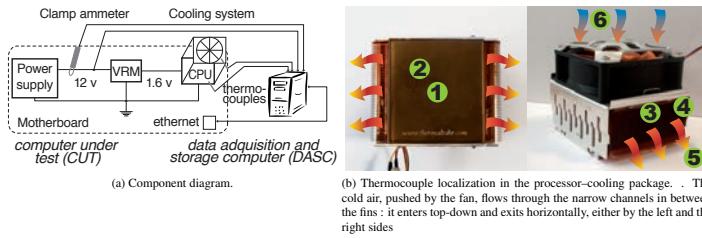


Figure 1: Overview of the platform with its main components

meet, and forced convection holds, heat transfer, q , becomes proportional to dissipation area, A , and gradient temperature, ΔT :

$$q = h \times A \times \Delta T$$

Being the constant h an (experimental) number depending mainly on thermal conductivity, speed of the flow, and channel geometry [11]. Acquiring temperature at multiple points will help us to determine the model goodness. Measurements are carried with K-type thermocouples optimized for the temperature range of 0-100 °C that are located at 6 positions: 1) drilled in middle of the heat sink contacting with the processor, 2) drilled in the border of the heat sink—Intel provides some guidelines for the placement at these locations [13], 3) in the lateral edge of a fin placed in the middle of the heat sink, 4) in the lateral edge of a fin placed in a corner of the heat sink, 5) in the free path of the output hot air flow without touching the heat sink, and 6) in the free path of the input cold air flow.

The six measurement points ease the verification of the forced convection model because from this model we know that the temperature of the hot air flow should be much bigger than that of the cold air flow. Also, the temperature should rise as we approach close to the processor; therefore, in the real measures we have to observe that $Temp(5) >> Temp(6)$ and $Temp(T1) > Temp(T2)$.

The acquisition of temperature samples is done with a Picootech TC-08 converter that is connected to a USB port of the DASC [22]. The conversion frequency depends on the number of attached thermocouples. In our case, 6 thermocouples, the data acquisition rate is 0.73 samples/second, so that any individual thermocouple gets sampled every 4.4 s. This rate is much smaller than that of power, but it is enough because the change rate of temperature is much lower than that of power as we will see in Section 4.

Since the platform uses two computers, it is required to synchronize the beginning and the end of the sampling pro-

cess. The synchronization is accomplished by sending two low-latency Ethernet packets, one just at the beginning of the execution of the program under test and the other just after its end. This synchronization schema is done by a wrapper on the executables that avoids any complexity to the students, even for those without a good shell knowledge. The platform is able to monitor any program independently of its execution time as long as the hard disk drive has space left.

Summarizing, the platform is able to measure the temperature and the energy drawn by the execution of any program in an Intel Pentium 4 processor with high precision and without interfering the computer under test. All the platform software is freely available upon request.

4. Platform Validation

Most changes in the hardware of the platform with regards to the previous version were motivated for increasing the sampling accuracy and for logging power and temperature simultaneously. The objective was to detect power phases during program execution, and to see how changes in energy consumption affected temperature.

As a prove of the accuracy of the platform, Figure 2 shows the temporal evolution of power and temperature for the complete run of `473.astar` (SPEC CFP2006) compiled at the maximum level of optimizations with Intel C compiler¹.

The left Figure, 2a, shows the instant power and temperature at the center of the heat sink (thermocouple 1 in Figure 1b). Note that with this easy experiment students can see how changes in the phases of programs also affects to its energy consumption, and how temperature reacts slowly to the changes in power—justifying the choice of a much lower sample rate for temperature than for power. Besides,

¹For more methodology details please read Section 5.

this plot also shows how the processor–heatsink–fan system tends towards their thermodynamic equilibrium when power is almost constant after roughly 130 s (this can be noticed in both the 150–300 and 600–800 time windows). Our software package includes a PID controller able to stabilize the processor consumption, or alternatively the temperature, at a given value for performing these kind of experiments in a controlled way.

In order to employ the forced convection model of the processor–cooling package we have to take several steps. The first one is to verify relations among the measured temperatures. As shown in the right Figure, 2b, the output air temperature (T_5) is warmer than the input one (T_6), and the difference in temperature increases as the processor activity rises. Once the initiation phase is completed, the temperature difference between the processor–heatsink package and the input air ($T_2 - T_6$) is large (a maximum of almost 30 °C) while the difference between the processor–heatsink package and the output air ($T_2 - T_5$) is small (less than 5 °C). These differences between both values indicate that the air is absorbing heat from the heatsink and spreads it out of the processor–heatsink package. Also the temperature in the middle of the heat sink (T_1) is bigger than that of the border of the heat sink (T_2). All these relations match with the model expectations.

The second step involves considering also the fin temperatures (T_3 and T_4), determine which gradient temperature has to be computed (ΔT), and tune the experimental constant h . We have some preliminary numbers, allowing us to approximate the package temperature from the power drawn by the processor, but we do not show the numbers because the model is not accurate enough; the h constant does not completely match with the handbook data normally used in thermal engineering.

5 Example Laboratory

This section describes a laboratory to get some insights between compiler optimizations and energy/power and then comments some other challenging experiments using the thermal measurement abilities of the platform.

5.1 Interaction between Compiler Optimization and Energy/Power

One possible application of the platform in academia is its use in computer architecture laboratories. For example, it easily allows to study the interaction between compiler optimizations and energy/power.

The lab would be introduced by explaining the basic relationships among time, energy, and power paying attention to what changes should be expected when the optimization level rises. An outline of such introduction follows.

In a processor without Dynamic Voltage Frequency Scaling (DVFS), the execution time T_{ex} of a program can be expressed as

$$T_{ex} = N_{inst} \times CPI \times T_{cycle} \quad (1)$$

where N_{inst} , CPI , and T_{cycle} represents the total number of instructions, the average number of cycles per instruction, and the cycle time, respectively. For minimizing T_{ex} , compilers focus on reducing the total number of cycles, $N_{inst} \times CPI$. But which are the effects of this reduction on power and energy?

Assuming the simplifying assumption that static bias current does not flow in a microprocessor [20], its total power consumption is given by

$$P_{tot} = P_{dyn} + P_{sta} = C_L V_{dd}^2 f + V_{dd} I_{leak} \quad (2)$$

where P_{tot} is the total sum of the dynamic and static power. The dynamic power, P_{dyn} , is the product of the average capacitance switched per cycle (processor activity), C_L , times the square of the supply voltage, V_{dd} , times the frequency, f . The static power is the product of the supply voltage times leakage current, I_{leak} [19].

From equations (1) and (2) we observe that compiler optimizations only affect power indirectly. Regarding dynamic power, P_{dyn} , on one hand, it is difficult to establish a relationship between N_{inst} and C_L because executing more, less, or different instructions may or may not change the performed activity per cycle. On the other hand, CPI seems to impact more the dynamic power (C_L) because optimizations that rise/reduce Instruction Level Parallelism (ILP), such as instruction scheduling or dead-code elimination, can increase/decrease activity per cycle, C_L .

Static power is less affected by compiler optimizations since it depends mostly on technological parameters; however, they can affect static power when the optimizations increase/decrease the processor activity and this results in a variation of processor temperature because leakage current depends on temperature [4]. The most straightforward path for reducing static power from compilation is to add special instructions in the code for switching off processor parts as suggested by Zhang *et al.* [26]. These proposals will become more and more important in the future because as technology scales, the percentage of static power is rising [15].

The product of P_{tot} times T_{ex} is the energy consumed by a program

$$\begin{aligned} E_{tot} &= P_{tot} \times T_{ex} = E_{dyn} + E_{sta} \\ &= C_{tot} V_{dd}^2 + V_{dd} I_{leak} \times T_{ex} \end{aligned} \quad (3)$$

where C_{tot} is the total capacitance that has been switched across all execution cycles.

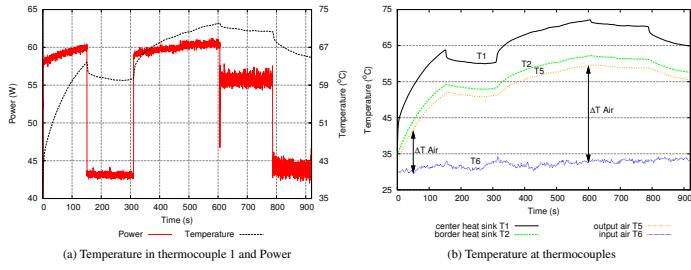


Figure 2: Temperature and Power temporal evolution during the full execution of 473.astar compiled with `iO3prf` options.

Recalling equations (1) and (3), E_{dyn} is independent of the frequency and

$$C_{tot} = C_L \times N_{inst} \times CPI \quad (4)$$

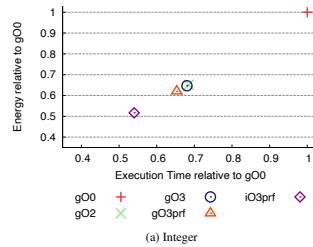
Thus, execution-time optimization saves energy when they reduce the total number of cycles, $N_{inst} \times CPI$, because we do not expect that compiler optimizations increase significantly C_L . In deep-pipelined processors with complex decoding such as the Intel Pentium 4, this is specially true because the energy consumed in the execution stage is smaller than the energy consumed in the rest.

Table 1: Compiler optimization impact summary. ↓, ?, and ↑ means decrement, undetermined, and increment, respectively.

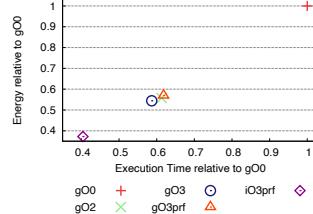
Power	$N_{inst} \downarrow$	$CPI \downarrow$
dynamic (P_{dyn})	?	↑
static (P_{sta})	?	?
Energy	$N_{inst} \downarrow$	$CPI \downarrow$
dynamic (E_{dyn})	↓	↓
static (E_{sta})	↓	?

Table 1 summarizes all previous relations and derives the effect of decreasing either N_{inst} or CPI , assuming constant the other factor. As it can be seen, performance-oriented compiler optimizations (focused on reducing $N_{inst} \times CPI$) are beneficial for energy, and may not be power-efficient when their target is to reduce only the CPI because dynamic power can increase. Asking the students to complete this table before the laboratory session is a good assignment for ensuring that students understand the underneath theory.

5.1.1 Experimental Results



(a) Integer



(b) Floating Point

Figure 3: Average Energy and Execution time relative to g00.

Table 2: Tested SPEC CPU2006 benchmarks.

Integer	Input	Floating Point	Input
400.perlbench	-l_71ib checkspam.pl 2500 5 25 11 150 1 1 1 1	436.cactusADM	benchADM.par
462.libquantum	1397 8	437.leslie3d	-l leslie3d.in
473.astar	rovers.cfg	447.dealII	23
483.xalancbmk	-v t5.xml xalanc.xsl	453.povray	SPEC-benchmark-ref.ini
		454.calculix	-l hyperviscoplastic
		470.lbm	3000 reference.dat 0 0 100,100,130,ldc.of

Table 3: Compiler configurations with their respective optimization flags

	Compiler	Flags
g00	gcc	-O0
g02	gcc	-O2 -mtune=pentium4 -march=pentium4
g03	gcc	-O3 -mtune=pentium4 -march=pentium4 -mfpmath=sse,387 -msse2
g03prf	gcc	-O3 -mtune=pentium4 -march=pentium4 -mfpmath=sse,387 -msse2 -fprofile-generate/use
i03prf	icc	-O3 -xN -ipo -no-prec-div -prof-gen/use

The previous relations can be verified with the proposed platform by executing multiple programs with different compiler optimizations and acquiring the energy and power measurements. For the sake of brevity, we only show the results for the relation between energy and execution time.

As a benchmark we can choose any program not spending most of the time in I/O to ensure that the impact of compiler optimization is significant in energy and power. Due to its widespread use in industry and academia SPEC CPU2006 has been our choice [8]. In order to reduce the measurement time we select the representative subset proposed by Phansalkar *et al.* [18]. The input sets for each program used in this paper are shown in Table 2. Other events of interest such as fetch stalls or instruction count can be measured with Intel Performance Tuning Utility (PTU); e.g., to compute the energy per instruction value [1].

To check the impact of compiler optimizations in energy and power we suggest to test multiple configurations of the GNU C compiler 4.1.2 (gcc) [7] and one configuration of the Intel C compiler 10.1 (icc) [12], all listed in Table 3. As a baseline, we use a configuration without optimizations, g00. We also checked a production-level configuration tuned for our processor, g02. Finally, we encourage using more aggressive gcc configurations: -O3 without and with profiling, and icc at its maximum level of optimizations with profiling (i03prf).

In integer, the more optimizations are applied, the better the results are. The best gcc configuration, g03prf saves 34.7% of execution time and 38% of energy. i03prf increases the gains saving 46% and 48.4% of execution time and energy, respectively. In floating point, optimizations are more effective; i.e., g02 (the best gcc configuration) saves 41.3% and 45.6% of execution time and energy, respectively. Again, i03prf performs better with 59.6% and 62.8% reductions in execution time and energy.

Gains in execution time and energy are very close suggesting a strong correlation. To support this claim, Figure 4 plots execution time and energy for each benchmark. As can be seen the correlation is strong, which is in line with previous work [24, 21]. We believe that the correlation is due to the fact that the clock net, static consumption, and fetch, decoding, and control parts of the processor consume more than functional units [25]; hence, it seems that reducing the number of executed instructions is more important than its kind for improving energy consumption.

Regarding execution time, icc beats gcc in all but one benchmark, 447.dealII. Besides, icc consumes less energy in all programs but 470.lbm. To conclude, both gcc and icc reduce notably the number of executed instructions (50% and 75% on average for integer and floating point, respectively) and increase the CPI (rising also the Energy per instruction) but icc does it in a lower quantity.

Summarizing, the main assignments for this lab can be: to perform the measurements for the program, to verify that the table they have completed before the lab is correct, and to finish extracting the conclusions of the previous paragraphs.

5.2 Other Experiments

The platform can be used with a more research-oriented focus such as master dissertations. For example, an outgoing work in our lab wants to obtain a power/temperature profile of individual instructions.

Since the processors' manual does not document the consumption of the instructions, we can get an estimation with the platform. For example, we have observed that stack operations rise power consumption and heat more the processor, which makes sense because stack instructions require a read/write in the cache and one increment/decrement in the stack pointer register in the same cycle.

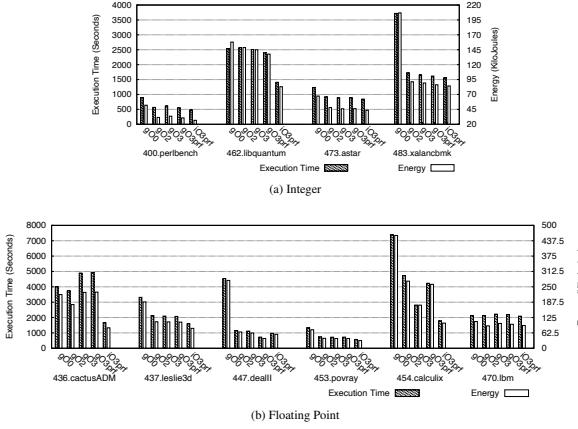


Figure 4: Execution time and Energy per benchmark.

6. Conclusions and Future Work

This paper presents a platform for measuring energy/power and temperature in commodity PCs with an academic focus. In this work, measures are carried out in an Intel Pentium 4, but the platform can be easily ported to any commodity PCs. The acquired data can be stored to perform off-line analysis, and its accuracy enables to detect power and temperature phases.

With the platform students can, for example, study the interaction between compiler optimizations and energy/power. This laboratory enables the student to learn that on average performance optimizations are energy-efficient.

Nowadays, the platform is used and extended by a small group of students. Our next main step is to set up a whole laboratory for using it as a regular laboratory session in our Computer Architecture and Heat Transfer courses. Our ongoing work is to obtain a simple linear equation relating measured power, fan speed, and dissipating surface to compute output air temperature for using it during the introduction of the laboratories.

Our future work will try to extend the platform reducing the granularity of the sampling process. Now, the platform does not know at which code fragment or function each sample belongs. We believe that this ability will help us finding the most heat-producing instruction sequences to continue our studies on per instruction energy estimations.

Acknowledgements

The authors would like to thank the anonymous reviewers for their suggestions on this paper. Dario Suárez Gracia and Víctor Viñals Yúfera were supported in part by the Gobierno de Aragón grant gaZ: Grupo Consolidado de Investigación, the Spanish Ministry of Education and Science under contracts TIN2007-66423, TIN2007-68023-C02-01, and Consolider CSD2007-00050, and the European Union Network of Excellence HiPEAC-2 (FP7/ICT 217068).

References

- [1] Intel Performance Tuning Utility 3.1 Update 3. <http://software.intel.com/en-us/articles/intel-performance-tuning-utility-31-update-3>, 2007 edition.
- [2] Analog Devices. *ADP3180, 6-Bit Programmable 2-, 3-, 4-Phase Synchronous Buck Controller*. Analog Devices, 2003.
- [3] A. Asín Pérez, D. Suárez Gracia, and V. Viñals Yúfera. A proposal to introduce power and energy notions in computer architecture laboratories. In *WCAE '07: Proceedings of the 2007 workshop on Computer architecture education*, pages 52–57, New York, NY, USA, 2007. ACM.
- [4] D. Brooks, R. P. Dick, R. Joseph, and L. Shang. Power, thermal, and reliability modeling in nanometer-scale microprocessors. *IEEE Micro*, 27(3):49–62, May-June 2007.

- [5] E. T. E. M. B. Consortium. EnergyBench™ version 1.0 power/energy benchmarks. http://www.eembc.org/benchmark/power_sl.php, 2008.
- [6] S. P. E. Corporation. SPECpower_ssj2008 benchmark suite. http://www.spec.org/power_ssj2008/, 2008.
- [7] Gcc team. *GCC 4.1.2 Manual*. <http://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/>. Free Software Foundation, February 2008.
- [8] J. L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, 2006.
- [9] C. Hu, J. McCabe, D. A. Jiménez, and U. Kremer. Infrequent basic-block-based program phase classification and power behavior characterization. In *Proceedings of The 10th IEEE Annual Workshop on Interaction between Compilers and Computer Architectures*. ACM Press, 2006.
- [10] A. T. Inc. Adlink pci-9112 data acquisition card. http://www.adlinktech.com/PD/web/PD_detail.php?cKind=&pid=29&seq=&id=&sid=2008.
- [11] F. P. Incropera, D. P. DeWitt, T. L. Bergman, and A. S. Lavine. *Fundamentals of Heat and Mass Transfer*. Wiley, 6th edition, 2007.
- [12] Intel. *Intel C++ Compiler 10.1 Professional edition*. <http://www.intel.com/cd/software/products/asmo-na/eng/277618.htm>, 2007 edition.
- [13] Intel. *Intel® Pentium® 4 Processor in the 478-Pin Package Thermal Design Guidelines*. Intel Corporation, 1st edition, May 2002.
- [14] C. Isci and M. Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 93, Washington, DC, USA, 2003. IEEE Computer Society.
- [15] S. Kaxiras and M. Martonosi. *Computer Architecture Techniques for Power-Efficiency*. Number 4 in Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2008.
- [16] F. J. Mesa-Martinez, M. Brown, J. Nayach-Battilana, and J. Renau. Measuring performance, power, and temperature from real processors. In *ExpsCS '07: Proceedings of the 2007 workshop on Experimental computer science*, page 16, New York, NY, USA, 2007. ACM.
- [17] F. J. Mesa-Martinez, J. Nayach-Battilana, and J. Renau. Power model validation through thermal measurements. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 302–311, New York, NY, USA, 2007. ACM.
- [18] A. Phansalkar, A. Joshi, and L. K. John. Analysis of redundancy and application balance in the spec cpu2006 benchmark suite. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 412–423, New York, NY, USA, 2007. ACM.
- [19] J. Rabaey. *Low Power Design Essentials*. Springer, 2009.
- [20] J. M. Rabaey, A. Chandrakasan, and B. Nikolic. *Digital Integrated Circuits. A design perspective*. Prentice Hall Electronics and VLSI series. Prentice Hall, second edition, 2003.
- [21] J. S. Seng and D. M. Tullsen. The effect of compiler optimizations on pentium 4 power consumption. In *Seventh Annual Workshop on Interaction between Compilers and Computer Architectures (INTERACT'03)*, page 51, 2003.
- [22] P. Technologies. *USB TC-08 Temperature Logger User's Guide*. Pico Technologies Limited, 2007.
- [23] Tektronix. Tektronix tpc-312 current probe. <http://www2.tek.com/cmswpt/psdetails.lotr?ct=PSci=13540&cs=psu&lc=EN>, 2008.
- [24] M. Valluri and L. John. Is compiling for performance == compiling for power? In *Fifth Annual Workshop on Interaction between Compilers and Computer Architectures (INTERACT'00)*, page 51, 2001.
- [25] W. Wu, L. Jin, J. Yang, P. Liu, and S. X.-D. Tan. A systematic method for functional unit power estimation in microprocessors. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*, pages 554–557, New York, NY, USA, 2006. ACM.
- [26] W. Zhang, J. S. Hu, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Compiler-directed instruction cache leakage optimization. In *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture*, page 208. IEEE Computer Society, 2002.

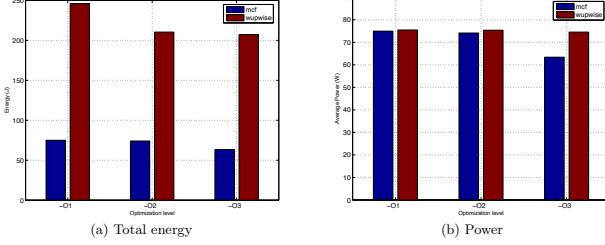


Figure 6: Mcf and wupwise varying the compilation optimization level

- monitoring on linux systems. In *Proceedings of the Conference on Linux Clusters: The HPC Revolution*, 2001.
- [3] J. L. Henning. Spec cpu2000: Measuring cpu performance in the new millennium. *Computer*, 33(7):28–35, 2000.
 - [4] C. Hu, J. McCabe, D. A. Jiménez, and U. Kremer. Infrequent basic block-based program phase classification and power behavior characterization. In *Proceedings of The 10th IEEE Annual Workshop on Interaction between Compilers and Computer Architectures*. ACM Press, 2006.
 - [5] Intel. *Voltage Regulator-Down (VRD) 10.0 Design Guide For Desktop Socket 478*. Intel Corporation, February 2004.
 - [6] C. Isci and M. Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *Proceedings of the Int. Symp. on Microarchitecture, MICRO'03*, page 93, Los Alamitos, CA, USA, 2003. IEEE Computer Society.
 - [7] N. S. Kim, T. Austin, D. Blaauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, and V. Narayanan. Leakage current: Moore's law meets static power. *Computer*, 36(12):68–75, 2003.
 - [8] A. J. KleinOsowski and D. J. Lilja. Minnespec: A new spec benchmark workload for simulation-based computer architecture research. *IEEE Comput. Archit. Lett.*, 1(1):7, 2006.
 - [9] National Instruments. Labview, <http://www.ni.com/labview/>.
 - [10] A. Naveh, E. Rotem, A. Mendelson, S. Gochman, R. Chabukswar, K. Krishnan, and A. Kumar. Power and thermal management in the Intel® Core™ Duo processor. *Intel Technology Journal*, 10:109–122, 2006.
 - [11] T. J. T. F. on Computing Curricula IEEE Computer Society Association for Computing Machinery. *Computing Curricula 2001 Computer Science*. Association for Computing Machinery, 2001.
 - [12] T. J. T. F. on Computing Curricula IEEE Computer Society Association for Computing Machinery. *Computer Engineering 2004. Curriculum Guidelines for Undergraduate Degree Programs in Computer*
 - [13] J. M. Rabaey, A. Chandrakasan, and B. Nikolić. *Digital Integrated Circuits. A design perspective*. Prentice Hall Electronics and VLSI series. Prentice Hall, 2nd edition, 2003.
 - [14] D. Sager, G. Hinton, M. Upton, T. Chappell, T. Fletcher, S. Samaan, and R. Murray. A 0.18 μ m CMOS IA32 microprocessor with a 4 GHz integer execution unit. In *Proceedings of the Solid-State Circuits Conference. Digest of Technical Papers*, pages 324–325, 461, 2001.
 - [15] A. Sinha, N. Ickes, and A. P. Chandrakasan. Instruction level and operating system profiling for energy exposed software. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 11(6):1044–1057, December 2003.
 - [16] D. Sylvester and K. Kentzer. Impact of small process geometries on microarchitectures in systems on a chip. *Proceedings of the IEEE*, 89(4):467 – 489, April 2001.
 - [17] S. Thompson, M. Alavi, M. Hussein, P. Jacob, C. Kenyon, P. Moon, M. Prince, S. Sivakumar, S. Tyagi, and M. Bohr. 130nm logic technology featuring 60nm transistors, low-k dielectrics, and eu interconnects. *Intel Technology Journal*, 6(2):5–13, May 2002.
 - [18] N. H. E. Weste and D. Harris. *CMOS VLSI Design. A Circuits and Systems Perspective*. Addison Wesley, 3rd edition, 2005.

6.3 Gestión de I+D+i y participación en comités científicos

6.3.1 Comités científicos, técnicos y/o asesores



D. Mº JESÚS TEJEDOR BAÑ, JEFE UNIDAD DE GESTIÓN ECONOMICA, PERSONAL Y CONTRATACIÓN DE LA ENTIDAD PÚBLICA ARAGONESA DE SERVICIOS TELEMÁTICOS CON CIF Q5000455E.

CERTIFICA

Que D. Darío Suárez Gracia, con D.N.I nº 18.443.694-V participó como asesor externo en las comisiones de selección constituidas en los procesos de selección relativos a la plaza del Grupo B Responsable de Infraestructuras y a la plaza del Grupo A Responsable de Explotación, realizados por esta entidad durante el año 2008.

Lo que certifico para su conocimiento y a los efectos oportunos.

Zaragoza, 16 de enero de 2012.

A handwritten signature in black ink, appearing to read "D. Darío Suárez Gracia". Below the signature is a circular official stamp.

6.3.2 Organización de actividades de I+D+i

6.3.3 Gestión de I+D+i



Departamento de
Informática e Ingeniería
de Sistemas
Universidad Zaragoza

D. Gonzalo López Nicolás, Presidente de la Comisión Académica del Programa de Doctorado
del Departamento de Informática e Ingeniería de Sistemas, de la Universidad de Zaragoza

HACE CONSTAR:

Que D. Darío Suárez Gracia, Profesor Ayudante Doctor, adscrito al Departamento de Informática e Ingeniería de Sistemas de la Universidad de Zaragoza, pertenece a la Comisión Académica del Programa de Doctorado de Ingeniería de Sistemas e Informática desde el día 1 de septiembre de 2016 hasta la actualidad.

Lo que comunica a petición de la interesada y a los efectos oportunos, en Zaragoza, a 13 de septiembre de 2017.

Fdo. Gonzalo López Nicolás

Pte. Comisión Académica Doctorado



Programa de Doctorado
Ingeniería de Sistemas
e Informática
Universidad Zaragoza

Maria de Luna, 1 - 50018 ZARAGOZA (ESPAÑA)
Teléfonos: (+34) 976 761949 - (+34) 976 762406
Fax: (+34) 976 762406
e-mail: secinf@unizar.es
<http://dis.unizar.es/>

unizar.es

6.3.4 Evaluación y revisión de proyectos y artículos de I+D+i



TESIS DOCTORALES - TESEO

Título: NETWORKS-ON-CHIP: FROM THE OPTIMIZATION OF TRADITIONAL ELECTRONIC NOCS TO THE DESIGN OF EMERGING OPTICAL NOCS

Nombre: Ortín Obón, Marta

Universidad: Universidad de Zaragoza

Departamento: Informática e ingeniería de sistemas

Fecha de lectura: 04/02/2016

Mención a doctor europeo: concedido

Programa de doctorado: Programa Oficial de Doctorado en Ingeniería de Sistemas e Informática

Dirección:

> **Director:** VÍCTOR VIÑALS YÚFERA

> **Director:** MARÍA VILLARROYA GAUDÓ

Tribunal:

> **presidente:** Julio Ramon Beivide Palacio

> **secretario:** Dario Suárez Gracia

> **vocal:** Sandro Bartolini

Descriptores:

> ARQUITECTURA DE ORDENADORES

El fichero de tesis no ha sido incorporado al sistema.

Resumen: As technology improves, memories and processors become faster, smaller, cheaper, and more energy-efficient, enabling computer architects to include more of them in a single chip. Now that Moore's Law is reaching its limit, the replication of simple cores is being used to continue improving performance while minimizing fabrication costs. As a consequence, performance does not only face a bottleneck on computing power and memory access any more, but also on the communication of the chip elements. In this context, interconnection networks have emerged to replace buses as the prevailing solution to provide fast, cost-effective, and scalable communications. They are the key for the success of future digital systems, both chip multiprocessors composed of tens of identical cores and heterogeneous systems-on-chip. In the last decade, there has been an extensive research effort towards optimizing the networks-on-chip (NoCs) from low-level physical aspects all the way up to system-level and application-related issues, and NoCs have now reached a mature level of development with their integration as a fundamental component in many successful commercial products.

In this thesis, we start by analysing the state-of-the-art of electronic networks-on-chip and detect that, even though the fundamental purpose of the interconnect is to exchange information among processors and memories, it is often designed and optimized without taking those essential components into consideration. We revisit the comparison of several well-known topologies from a comprehensive point of view: running realistic applications on a detailed model of the processors, the caches, and the interconnect. This study identifies the dominant impact of the network latency and designates the concentrated mesh as the most cost-effective



Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza



DOÑA ESMERALDA MAINAR MAZA, Secretaria de la Escuela de Ingeniería y Arquitectura de la Universidad de Zaragoza,

CERTIFICA:

Que, según los antecedentes obrantes en la Secretaría de este Centro, Don **DARIO SUÁREZ GRACIA**, como profesor adscrito al área de Arquitectura y Tecnología de Computadores de el departamento de Informática e Ingeniería de Sistemas fue nombrado miembro de los siguientes tribunales de evaluación de Trabajos Fin de Grado, Trabajos Fin de Máster y/o Proyectos Fin de Carrera, durante los cursos académicos que se indican:

CURSO	TITULACIÓN Y ESPECIALIDAD	CARGO
2015-2016	INGENIERO EN INFORMÁTICA	SUPLENTE
2016-2017	GRADO EN INGENIERÍA INFORMÁTICA	Nº 3 TITULAR
	GRADO INGENIERÍA INFORMÁTICA 1	SECRETARIO

Y, para que conste, a petición de la persona interesada, expido la presente certificación con el VºBº del Ilmo. Sr. Director y sello del Centro, en Zaragoza, a día 12 de septiembre de 2017.

VºBº
El Director,

Edo.: José Ángel Castellanos Gómez Universidad de Zaragoza Mainar Maza

La Secretaria,

Fdo.: David Melús Andrés

El Funcionario,

Escuela de Ingeniería y Arquitectura
Maria de Luna, 3 Edificio Torres Quevedo Campus Rio Ebro 50018-ZARAGOZA.



Association for
Computing Machinery

Advancing Computing as a Science & Profession

June 27–29, 2011
Darmstadt, Germany



ITiCSE'11

Proceedings of the 16th Annual Conference on
Innovation and Technology in Computer Science

Sponsored by:

ACM SIGCSE

Supported by:

**TU Darmstadt, Intel, BlackBerry, GK E-Learning, SoftwareAG,
Carlo und Karin Giersch-Stiftung, and Sparkasse Darmstadt**

ITiCSE 2011 Reviewers

- Raman Adaikalavan, *Indiana University South Bend*
Elizabeth S. Adams, *James Madison University*
Rajeev Agrawal, *North Carolina A&T State University*
Tuukka Ahoniemi, *Tampere University of Technology and Digiia Plc*
Carl Alphonse, *University at Buffalo, The State University of New York*
Ruth Anderson, *University of Washington*
Barbara Anthony, *Southwestern University*
Michal Armoni, *Weizmann Institute of Science*
John Aycock, *University of Calgary*
David Barnes, *University of Kent*
Lewis Barnett, *University of Richmond*
John Barr, *Ithaca College*
Valerie Barr, *Union College*
Tim Bell, *University of Canterbury*
Mordechai Ben-Ari, *Weizmann Institute of Science*
Mikael Berndtsson, *University of Skovde*
Stefan Brandle, *Taylor University*
Torsten Brinda, *University of Erlangen-Nuremberg*
Kim Bruce, *Pomona College*
Dennis Brylow, *Marquette University*
David Bunde, *Knox College*
Barry Burd, *Drew University*
Andre Paul Calitz, *Nelson Mandela Metropolitan University*
Daniel Canas, *Wake Forest University*
Lillian N. Cassel, *Villanova University*
Maiga Chang, *Athabasca University*
Mahesh Chaudhari, *Arizona State University*
Peng-Wen Chen, *Oriental Institute of Technology*
Ping Chen, *University of Houston-Downtown*
Li-hsiang Cheo, *William Paterson University of New Jersey*
Jayan Kurian Chirayath, *Royal Melbourne Institute of Technology*
Radhouane Chouchane, *Columbus State University*
Vincen Cicirello, *Richard Stockton College*
John Cigas, *Park University*
Peter Clarke, *Florida International University*
Tony Clear, *Auckland University of Technology*
Joe Clifton, *University of Wisconsin, Platteville*
Stephen Cooper, *Purdue University*
Michelle Craig, *University of Toronto*
Mark Crouch, *Angelo State University*
Joyce Blair Crowell, *Belmont University*
Jose Cunha, *New University of Lisbon*
Steve Cunningham, *Stanford University*
Quintin Cutts, *University of Glasgow*
Nell Dale, *University of Texas at Austin*
Andrew Dalton, *Western Carolina University*
Douglas Dankel, *University of Florida*
Lawrence D'Antonio, *Ramapo College of New Jersey*
Stephen Davies, *University of Mary Washington*
Renzo Davoli, *University of Bologna*
Adrienne Decker, *University at Buffalo SUNY*
Katherine Deibel, *University of Washington-Seattle*
Rafael del Vado, *Universidad Complutense de Madrid*
Dorothy Deremer, *Montclair State University*
Molisa Derk, *Dickinson State University*
Kamyar Dezghosha, *University of Illinois at Springfield*
Michele Di Santo, *University of Sannio*
Suzanne W. Dietrich, *Arizona State University*
William Doane, *Bennington College*
Zachary Dodds, *Harvey Mudd College*
John Dooley, *Knox College*
Brian Dorn, *University of Hartford*
Maureen Doyle, *Northern Kentucky University*
Peter Drexel, *Plymouth State University*
J. Philip East, *University of Northern Iowa*
Mary Anne Egan, *Siena College*
Joseph Ekstrom, *Brigham Young University*
Stephanie Elzer, *Millersville University*
Barbara Ericson, *Georgia Tech*
Daniel Ernst, *The University of Wisconsin - Eau Claire*
Henry Etlinger, *Rochester Institute of Technology*
Alan Fekete, *University of Sydney*
Maria Feldgen, *Universidad de Buenos Aires*
Ernest Ferguson, *Northwest Missouri State University*
Samantha Foley, *Oak Ridge National Laboratory*
Edward Fox, *Virginia Tech*
Alessio Gaspar, *University of South Florida Polytechnic*
Paul Gestwicki, *Ball State University*
Michael Goldweber, *Xavier University*
Jean Goulet, *Universite de Sherbrooke*
Mary Granger, *George Washington University*

Simon Olberding, *TU Darmstadt*
Lawrence Osborne, *Lamar University*
Barbara Owens, *Southwestern University*
Katherine Panciera, *University of Minnesota*
Iraklis Paraskakis, *City College*
Abelardo Pardo, *Carlos III University of Madrid*
David Parker, *Salisbury University*
James Paterson, *Glasgow Caledonian University*
Laurie Patterson, *University of North Carolina - Wilmington*
Arnold Pears, *Uppsala University*
Teresa Peterman, *Grand Valley State University*
Andrew Petersen, *University of Toronto Mississauga*
Vreda Pieterse, *University of Pretoria*
Paul Piwowarski, *University of Kentucky*
Wayne Pollock, *Hillsborough Community College*
Irene Polycarpou, *Colorado School of Mines*
John-Paul Pretti, *University of Waterloo*
Philip Prins, *Seattle Pacific University*
Atanas Radenski, *Chapman University*
John Rager, *Amherst College*
Noa Ragonis, *Beit Berl College*
Bina Ramamurthy, *University at Buffalo*
Samuel Rebelsky, *Grimmell College*
Charles Riedesel, *University of Nebraska - Lincoln*
Suzanne Rivoire, *Sonoma State University*
Christian Roberson, *Plymouth State University*
Eric Roberts, *Stanford University*
Stefan Robila, *Montclair State University*
Susan H. Rodger, *Duke University*
Guido Roessling, *Technische Universität Darmstadt*
Constantine Roussos, *Lynchburg College*
Ingrid Russell, *University of Hartford*
Adrian Rusu, *Rowan University*
Mihaela Sabin, *University of New Hampshire*
Mehran Sahami, *Stanford University*
Ian Sanders, *University of the Witwatersrand*
Otto Seppala, *Helsinki University of Technology*
Behrooz Seyed-Abbassi, *University of North Florida*
Judy Sheard, *Monash University*
Ching-Kuang Shene, *Michigan Technological University*
Mark Sherriff, *University of Virginia*
Yasuto Shirai, *Shizuoka University*
Charles Shub, *University of Colorado at Colorado Springs*

Peter Smith, *California State University - Channel Islands*
Raja Sooriamurthi, *Carnegie Mellon University*
Barry Soroka, *Cal Poly Pomona*
Christian Spannagel, *PH Heidelberg*
Carol Spradling, *Northwest Missouri State University*
Jeffrey Stone, *Pennsylvania State University*
Fred Strickland, *South University*
Dario Suarez Gracia, *Universidad de Zaragoza*
Jorma Tarhio, *Helsinki University of Technology*
James Teresco, *Siena College*
Allison Elliott Tew, *Georgia Institute of Technology*
William Thacker, *Winthrop University*
Megan Thomas, *California State University Stanislaus*
Rebecca Thomas, *Bard College*
Errol Thompson, *University of Birmingham*
John Thompson, *Buffalo State College*
Deborah Trytten, *University of Oklahoma*
Shengru Tu, *University of New Orleans*
William Turner, *Wabash College*
Guenther Tusch, *Grand Valley State University*
Hakan Tuzun, *Hacettepe University*
Suleyman Uludag, *The University of Michigan - Flint*
Jaime Urquiza-Fuentes, *Rey Juan Carlos University/Lecturer*
Ian Utting, *University of Kent at Canterbury*
Jan Vahrenhold, *Technische Universität Dortmund*
Tammy VanDeGrift, *University of Portland*
Troy Vasiga, *University of Waterloo*
Steven Vegdahl, *University of Portland*
J. Angel Velazquez-Iturbide, *Universidad Rey Juan Carlos*
Tamar Vilner, *The Open University of Israel*
David Voorhees, *Le Moyne College*
Sally Wahba, *Clemson University*
Gursimran Walia, *North Dakota State University*
Henry Walker, *Grimmell College*
Thomas Way, *Villanova University*
Howard Whitston, *University of South Alabama*
Linda Wilkens, *Providence College*
Craig Wills, *Worcester Polytechnic Institute*
Gary Ka Wai Wong, *the Community College at Lingnan University*
Arthur Yanushka, *Christian Brothers University*
Juang Yih-Ruey, *Jinwen University of Science & Technology*



Association for
Computing Machinery

Advancing Computing as a Science & Profession

February 29–March 3, 2012
Raleigh, North Carolina, USA



SIGCSE'12

Proceedings of the 43rd ACM Technical
Symposium on Computer Science Education

Sponsored by:
ACM SIGCSE

SIGCSE 2012 Reviewers

Safurah Abdul Jalil, <i>University of Auckland</i>	Isabel Azevedo, <i>ISEP</i>	Michael Blumenstein, <i>Griffith University</i>
Adel Abunawass, <i>University of West Georgia</i>	Serhat Azgur, <i>Bilkent University</i>	Steven Bogart, <i>Wittenberg University</i>
Ernest Ackermann, <i>University of Mary Washington</i>	Dinabandhu Bag, <i>National Institute of Technology</i>	Esmail Bonakdarian, <i>Franklin University</i>
Donald Acton, <i>University of British Columbia</i>	Mark Bailey, <i>Hamilton College</i>	William Booth, <i>Baylor University</i>
Raman Adaikalavan, <i>Indiana University South Bend</i>	Laura Baker, <i>St. Edward's University</i>	James Bowring, <i>College of Charleston</i>
Victor Adamchik, <i>Carnegie Mellon University</i>	Doug Baldwin, <i>SUNY Geneseo</i>	Stefan Brandle, <i>Taylor University</i>
Elizabeth S. Adams, <i>James Madison University</i>	Catherine Barreiss, <i>Olivet Nazarene University</i>	Evelyn Bramnock, <i>George Gwinnett College</i>
Joel Adams, <i>Calvin College</i>	Ian Barland, <i>Radford University</i>	Anna Bretscher, <i>University of Toronto at Scarborough</i>
Robert Adams, <i>Grand Valley State University</i>	David Barnes, <i>University of Kent</i>	Monica Brockmeyer, <i>Wayne State University</i>
Paul Addison, <i>Ivy Tech Community College</i>	Tiffany Barnes, <i>University of North Carolina at Charlotte</i>	Renee Bryce, <i>Utah State University</i>
Rajeev Agrawal, <i>North Carolina A & T State University</i>	Lewis Barnett, <i>University of Richmond</i>	Dennis Brylow, <i>Marquette University</i>
Shakil Akhtar, <i>Clayton State University</i>	N. Dwight Barnette, <i>Virginia Tech</i>	Joel Brynielsson, <i>Royal Institute of Technology</i>
Paul Albee, <i>Central Michigan University</i>	John Barr, <i>Ithaca College</i>	Suzanne Buechele, <i>Southwestern University</i>
Richard Allen, <i>St. Olaf College</i>	Valerie Barr, <i>Union College</i>	Duane Buck, <i>Otterbein University</i>
Vicki Almstrum, <i>University of Texas at Austin</i>	Martin Barrett, <i>East Tennessee State University</i>	Everett Bull, <i>Pomona College</i>
Hana Alnuaim, <i>King Abdulaziz University</i>	Joao Barros, <i>Instituto Politecnico de Beja - ESTIG</i>	David Bunde, <i>Knox College</i>
Carl Alphonce, <i>University at Buffalo, The State University of New York</i>	Phillip Barry, <i>University of Minnesota</i>	Barry Burd, <i>Drew University</i>
Jamal Alsabagh, <i>Grand Valley State University</i>	Carlos Barto, <i>Universidad Nacional de Cordoba</i>	Darci Burge, <i>Nassau Community College</i>
Christine Alvarado, <i>Harvey Mudd College</i>	Rebecca Bates, <i>Minnesota State University Mankato</i>	Kevin Burger, <i>Arizona State University</i>
Peter Anderson, <i>Rochester Institute of Technology</i>	Michael Bauer, <i>University of Hawaii - Leeward Community College</i>	Ben Burrell, <i>Randolph-Macon College</i>
Ruth Anderson, <i>University of Washington</i>	Robert Beck, <i>Villanova University</i>	Vicky Bush, <i>University of Gloucestershire</i>
Peter Andreac, <i>Victoria University of Wellington</i>	Byron Weber Becker, <i>University of Waterloo</i>	Mary Elaine Calif, <i>Illinois State University</i>
Stefan Andrei, <i>Lamar University</i>	Andrew Begel, <i>Microsoft Research</i>	Andre Paul Calitz, <i>Nelson Mandela Metropolitan University</i>
Danner Andrew, <i>Swarthmore College</i>	John Beidler, <i>Univ. of Scranton</i>	Johan Calu, <i>KIIBO</i>
Peter Andrews, <i>Eastern Illinois University</i>	Tim Bell, <i>University of Canterbury</i>	Tracy Camp, <i>Colorado School of Mines</i>
Steven K. Andrianoff, <i>St. Bonaventure University</i>	Mordechai Ben-Ari, <i>Weizmann Institute of Science</i>	Daniel Canas, <i>Wake Forest University</i>
Karen Anewalt, <i>University of Mary Washington</i>	Jens Bennedsen, <i>Aarhus School of Engineering</i>	Roxanne Canosa, <i>Rochester Institute of Technology</i>
Barbara Anthony, <i>Southwestern University</i>	Chris Bennett, <i>University of Maine Farmington</i>	Angela Carbone, <i>Monash University</i>
Florence Appel, <i>Saint Xavier University</i>	Seth Bergmann, <i>Rowan University</i>	James Caristi, <i>Valparaiso University</i>
Antonio Araujo, <i>Universidade do Porto</i>	Mikael Berndtsson, <i>University of Skovde</i>	Andrew Carle, <i>University of California, Berkeley</i>
Michal Armoni, <i>Weizmann Institute of Science</i>	Gian Mario Besana, <i>CTI DePaul</i>	Martin Carlisle, <i>US Air Force Academy</i>
Tom Armstrong, <i>Wheaton College</i>	Anne Beug, <i>California Polytechnic State University</i>	Janet Carter, <i>University of Kent at Canterbury</i>
David Arnov, <i>Brooklyn College</i>	Ivona Bezakova, <i>Rochester Institute of Technology</i>	Lori Carter, <i>Point Loma Nazarene University</i>
Charles Ashbacher, <i>Charles Ashbacher Technologies</i>	Marie Bienkowski, <i>SRI International</i>	Jeffrey Carver, <i>University of Alabama</i>
John Avitable, <i>College of Saint Rose</i>	William Birmingham, <i>Grove City College</i>	Lori Case, <i>University of Waterloo</i>
John Aycock, <i>University of Calgary</i>	Judith M Bishop, <i>Microsoft Research</i>	Steven Case, <i>University of the Virgin Islands</i>
Pavel Azalov, <i>Penn State</i>	M. Brian Blake, <i>University of Notre Dame</i>	Lillian N. Cassel, <i>Villanova University</i>
	Stephen Bloch, <i>Adelphi University</i>	Sheila Castaneda, <i>Clarke University</i>
		Tim Chamillard, <i>University of Colorado at Colorado Springs</i>
		Albert Chan, <i>Fayetteville State University</i>
		Maiga Chang, <i>Athabasca University</i>

Donna Reese, <i>Mississippi State University</i>	Stephen Schaub, <i>Bob Jones University</i>	Evelyn Stiller, <i>Plymouth State University</i>
Susan Reiser, <i>UNC Asheville</i>	Walter Schilling, <i>Milwaukee School of Engineering</i>	Vojislav Stojkovic, <i>Morgan State University</i>
Dan Resler, <i>Virginia Commonwealth University</i>	Diane Schwartz, <i>California State University Northridge</i>	Christopher Stone, <i>Harvey Mudd College</i>
Eugene K. Ressler, <i>United States Military Academy</i>	leslie schwartzman, <i>UIS</i>	Jeffrey Stone, <i>Pennsylvania State University</i>
Loren Rhodes, <i>Juniata College</i>	Dino Schweitzer, <i>United States Air Force Academy</i>	Forrest Stinedahl, <i>Centre College</i>
Catherine Ricardo, <i>Iona College</i>	Otto Seppälä, <i>Aalto University</i>	Fred Strickland, <i>South University</i>
Michael Rieck, <i>Drake University</i>	Amber Settle, <i>DePaul University</i>	Kristina Sriegnitz, <i>Union College</i>
Charles Riedesel, <i>University of Nebraska - Lincoln</i>	Behrooz Seyed-Abassi, <i>University of North Florida</i>	Catherine Stringfellow, <i>Midwestern State University</i>
Suzanne Rivoire, <i>Sonoma State University</i>	Cliff Shaffer, <i>Virginia Tech</i>	Lena Stromback, <i>Department of Computer and Information Science</i>
Steven Robbins, <i>University of Texas at San Antonio</i>	Victor R.L. Shen, <i>National Taipei University</i>	Deborah Sturm, <i>College of Staten Island CUNY</i>
Christian Roberson, <i>Plymouth State University</i>	Ching-Kuang Shene, <i>Michigan Technological University</i>	Dario Suarez Gracia, <i>Universidad de Zaragoza</i>
Eric Roberts, <i>Stanford University</i>	Mark Sherriff, <i>University of Virginia</i>	Jose Such, <i>Technical University of Valencia</i>
Stefan Robila, <i>Montclair State University</i>	Sen Shilar, <i>Macalester College</i>	Leigh Ann Sudol, <i>Carnegie Mellon University</i>
Susan H. Rodger, <i>Duke University</i>	Yasuto Shirai, <i>Shizuoka University</i>	John R. Sullins, <i>Youngstown State University</i>
Ken Rodham, <i>Brigham Young University</i>	Keith Shomper, <i>Cedarville University</i>	Fred Sullivan, <i>Wilkes University</i>
Guido Roessling, <i>Technische Universität Darmstadt</i>	charles shub, <i>University of Colorado at Colorado Springs</i>	Wayne Summers, <i>Columbus State University</i>
Sami Rollins, <i>University of San Francisco</i>	Scott Sigman, <i>Drury University</i>	Valerie Summet, <i>Emory University</i>
Joel Ross, <i>University of California, Irvine</i>	Gavin Sim, <i>Uni of Central Lancashire</i>	Weiqing Sun, <i>University of Toledo</i>
John Ross, <i>Indiana University Kokomo</i>	María Simi, <i>Universita di Pisa</i>	Ken Surendran, <i>Southeast Missouri State University</i>
Jerry Roth, <i>Vanderbilt University</i>	Beth Simon, <i>University of California, San Diego</i>	Jerry Talton, <i>Stanford University</i>
Krishnendu Roy, <i>Valdosta State University</i>	Oberta Slotterbeck, <i>Hiram College</i>	Joo Tan, <i>Kutztown University</i>
Martin Ruckert, <i>Munich University of Applied Sciences</i>	William Slough, <i>Eastern Illinois University</i>	Stewart Tansley, <i>Microsoft Research</i>
Anthony Ruocco, <i>Roger Williams University</i>	Joslyn Smith, <i>Florida International University</i>	Blair Taylor, <i>Towson University</i>
Paulo Rapino da Cunha, <i>University of Coimbra</i>	Peter Smith, <i>California State University - Channel Islands</i>	James Teresco, <i>Siena College</i>
Adrian Rusu, <i>Rowan University</i>	Robert Snapp, <i>University of Vermont</i>	Allison Elliott Tew, <i>Georgia Institute of Technology</i>
Amalia Rusu, <i>Fairfield University</i>	Peter Sommerlad, <i>HSR Rapperswil</i>	William Thacker, <i>Winthrop University</i>
Rebecca Rutherford, <i>Southern Polytechnic State University</i>	Joel Sommers, <i>Colgate University</i>	Lynda Thomas, <i>Aberystwyth University</i>
Mihala Sabin, <i>University of New Hampshire</i>	Raja Sooriyamurthi, <i>Carnegie Mellon University</i>	Megan Thomas, <i>California State University Stanislaus</i>
Roberta Evans Sabin, <i>Loyola College</i>	Jonathan Sorenson, <i>Butler University</i>	Rebecca Thomas, <i>Bard College</i>
Shazia Sadiq, <i>The University of Queensland</i>	Barry Soroka, <i>Cal Poly Pomona</i>	Stan Thomas, <i>Wake Forest University</i>
Mehran Sahami, <i>Stanford University</i>	Jaime Spacco, <i>Knox College</i>	Alfred Thompson, <i>Microsoft</i>
Samuel Sambasivam, <i>Azusa Pacific University</i>	Greg Speegle, <i>Baylor University</i>	Jodi Tims, <i>Baldwin-Wallace College</i>
Ian Sanders, <i>University of the Witwatersrand</i>	David Spooner, <i>Rensselaer Polytechnic Institute</i>	Fergus Toolan, <i>University College Dublin</i>
William Sanders, <i>University of Hartford</i>	Carol Spradling, <i>Northwest Missouri State University</i>	Gloria Childress Townsend, <i>DePauw University</i>
Pete Sanderson, <i>Otterbein University</i>	Nathan Sprague, <i>James Madison University</i>	Christian Trefftz, <i>Grand Valley State University</i>
André Santos, <i>Lisbon University Institute</i>	Denbigh Starkey, <i>Montana State University</i>	Deborah Trytten, <i>University of Oklahoma</i>
Vijayalakshmi Saravanan, <i>Ryerson University, Canada & VIT University</i>	Mark Stehlik, <i>Carnegie Mellon University</i>	Shengru Tu, <i>University of New Orleans</i>
Amit Sawant, <i>NetApp</i>	Josh Steinhurst, <i>Bucknell University</i>	David S. Tucker, <i>Midwestern State Univ.</i>
Suzanne Schaefer, <i>University of California, Los Angeles</i>	Ben Stephenson, <i>University of Calgary</i>	William Turner, <i>Wabash College</i>
christelle scharff, <i>Pace University</i>	Daniel Stevenson, <i>University of Wisconsin - Eau Claire</i>	Sharon M. Tuttle, <i>Humboldt State University</i>
	Khadja Stewart, <i>DePauw University</i>	Hakan Tuzun, <i>Hacettepe University</i>



Association for
Computing Machinery

Advancing Computing as a Science & Profession

March 9-12, 2011
Dallas, Texas, USA



SIGCSE'11

Proceedings of the 42nd ACM Technical Symposium on
Computer Science Education

Sponsored by:

ACM SIGCSE

Editors:

Thomas J. Cortina, Carnegie Mellon University

Ellen L. Walker, Hiram College

Laurie Smith King, College of the Holy Cross

David R. Musicant, Carleton College

Lester I. McCann, The University of Arizona

SIGCSE 2011 Reviewers

- Adel Abunawass, *University of West Georgia*
 Ernest Ackermann, *University of Mary Washington*
 Donald Acton, *University of British Columbia*
 Raman Adaikkalavan, *Indiana University South Bend*
 Victor Adamchik, *Carnegie Mellon University*
 Elizabeth S. Adams, *James Madison University*
 Evans Adams, *Fort Lewis College*
 Joel Adams, *Calvin College*
 Robert Adams, *Grand Valley State University*
 Paul Addison, *Ivy Tech Community College*
 Iwan Adhicandira, *University of Pisa*
 Rajeev Agrawal, *North Carolina A&T State University*
 Shakil Akhtar, *Clayton State University*
 Paul Albee, *Central Michigan University*
 Vicki Almstrum, *Univ. of Texas at Austin*
 Carl Alphonce, *University at Buffalo, The State University of New York*
 Jamal Alsabbagh, *Grand Valley State University*
 Ruth Anderson, *University of Washington*
 Stefan Andrei, *Lamar University*
 Peter Andrews, *Eastern Illinois University*
 Steven K. Andrianoff, *St. Bonaventure University*
 Karen Anewalt, *University of Mary Washington*
 Barbara Anthony, *Southwestern University*
 Antonio Araujo, *Faculdade de Engenharia da Universidade do Porto*
 Michal Armoni, *Weizmann Institute of Science*
 David Arnov, *Brooklyn College*
 Charles Ashbacher, *Charles Ashbacher Technologies*
 Owen Astrachan, *Duke University*
 Stephanie August, *Loyola Marymount University*
 John Aycock, *University of Calgary*
 Pavel Azalov, *Penn State*
 Shiri Azenkot, *University of Washington*
 Laura Baker, *St. Edward's University*
 Doug Baldwin, *SUNY Geneseo*
 Jie Bao, *University of Minnesota*
 Catherine Bareiss, *Olivet Nazarene University*
 Ian Barland, *Radford University*
 David Barnes, *University of Kent*
- Tiffany Barnes, *University of North Carolina at Charlotte*
 Lewis Barnett, *University of Richmond*
 N. Dwight Barnett, *Virginia Tech*
 John Barr, *Ithaca College*
 Valerie Barr, *Union College*
 Joao Barros, *Instituto Politecnico de Beja - ESTIG*
 Philip Barry, *University of Minnesota*
 Rebecca Bates, *Minnesota State University Mankato*
 Michael Bauer, *University of Hawaii - Leeward Community College*
 Robert Beck, *Villanova University*
 Byron Weber Becker, *University of Waterloo*
 Andrew Begel, *Microsoft Research*
 John Beidler, *University of Scranton*
 Tim Bell, *University of Canterbury*
 Mordechai Ben-Ari, *Weizmann Institute of Science*
 Jens Bennedsen, *Aarhus School of Engineering*
 Chris Bennett, *University of Maine Farmington*
 Susan Bergin, *National University of Ireland, Maynooth*
 Seth Bergmann, *Rowan University*
 Mikael Berndtsson, *University of Skövde*
 Sanjukta Bhowmick, *University of Nebraska at Omaha*
 Devi Prasad Bhukya, *Osmania University*
 Robert Biddle, *Carleton University*
 William Birmingham, *Grove City College*
 M. Brian Blake, *Georgetown University*
 Michael Blumentstein, *Griffith University*
 Esmail Bonakdarian, *Franklin University*
 William Booth, *Baylor University*
 Matthew Boutell, *Rose-Hulman Institute of Technology*
 Dennis Bouvier, *Southern Illinois University-Edwardsville*
 Roger Boyle, *University of Leeds*
 Stefan Brandle, *Taylor University*
 Anna Bretscher, *University of Toronto at Scarborough*
 Josef Breutzmann, *Wartburg College*
 Monica Brockmeyer, *Wayne State University*
 Kim Bruce, *Pomona College*
 Renee Bryce, *Utah State University*
 Dennis Brylow, *Marquette University*
 Joel Brynielsson, *Royal Institute of Technology*
 Suzanne Buchele, *Southwestern University*
 Duane Buck, *Otterbein University*
- Everett Bull, *Pomona College*
 David Bunde, *Knox College*
 Barry Burd, *Drew University*
 Darci Burge, *Nassau Community College*
 Kevin Burger, *Arizona State University*
 Gerald Burgess, *Western New Mexico University*
 Vicky Bush, *University of Gloucestershire*
 Caldeira Caldeira, *University of Evora*
 Mary Elaine Califff, *Illinois State University*
 Andre Paul Calitz, *Nelson Mandela Metropolitan University*
 Debra Calliss, *Arizona State University*
 Johan Calu, *KHBO*
 Mario Camilleri, *University of Malta*
 Tracy Camp, *Colorado School of Mines*
 Daniel Canas, *Wake Forest University*
 Angela Carbone, *Monash University*
 James Caristi, *Valparaiso University*
 Stephen Carl, *Seawee: The University of the South*
 Martin Carlisle, *US Air Force Academy*
 Janet Carter, *University of Kent at Canterbury*
 Lori Carter, *Point Loma Nazarene University*
 Jeffrey Carver, *University of Alabama*
 Lori Case, *University of Waterloo*
 Steven Case, *University of the Virgin Islands*
 Lillian N. Cassel, *Villanova University*
 Mano Chad, *Utah State University*
 Tim Chamillard, *University of Colorado at Colorado Springs*
 Tat Chan, *Methodist University*
 Maiga Chang, *Athabasca University*
 Mahesh Chaudhari, *Arizona State University*
 Ranjan Chaudhuri, *Eastern Michigan University*
 Tom Cheatham, *Middle Tennessee State University*
 Tzu-Yi Chen, *Pomona College*
 Wei Kian Chen, *Champlain College*
 Jimmy Chen, *Salt Lake Community College*
 Peng-Wen Chen, *Oriental Institute of Technology*
 Ping Chen, *University of Houston-Downtown*
 Li-hsiang Cheo, *William Paterson University of New Jersey*
 Chia-Chu Chiang, *University of Arkansas at Little Rock*
 Donald Chinn, *University of Washington, Tacoma*

Jaime Spacco, <i>Knox College</i>	Shengru Tu, <i>University of New Orleans</i>	Laurie White, <i>Mercer University</i>
Greg Speegle, <i>Baylor University</i>	David S. Tucker, <i>Midwestern State University</i>	Elizabeth White, <i>George Mason University</i>
David Spooner, <i>Rensselaer Polytechnic Institute</i>	William Turner, <i>Wabash College</i>	Howard Whitston, <i>University of South Alabama</i>
Carol Spradling, <i>Northwest Missouri State University</i>	Guenther Tusch, <i>Grand Valley State University</i>	Paul Wiedemeier, <i>The University of Louisiana at Monroe</i>
Denbigh Starkey, <i>Montana State University</i>	Sharon M. Tuttle, <i>Humboldt State University</i>	Sheila Wiggins, <i>Morgan State University</i>
Mark Stehlík, <i>Carnegie Mellon University</i>	Hakan Tuzun, <i>Hacettepe University</i>	Linda Wilkens, <i>Providence College</i>
Josh Steinhurst, <i>Bucknell University</i>	Paul Tymann, <i>Rochester Institute of Technology</i>	Judith Williams, <i>William Penn University</i>
Daniel Stevenson, <i>University of Wisconsin - Eau Claire</i>	Suleyman Uludag, <i>The University of Michigan - Flint</i>	Craig Wills, <i>Worcester Polytechnic Institute</i>
Khadija Stewart, <i>DePauw University</i>	Susan D. Urban, <i>Texas Tech University</i>	Linda Wilson, <i>Texas Lutheran University</i>
Evelyn Stiller, <i>Plymouth State University</i>	Timothy Urness, <i>Drake University</i>	Brent Wilson, <i>George Fox University</i>
Vojislav Stojkovic, <i>Morgan State Univ.</i>	Jaime Urquiza-Fuentes, <i>Rey Juan Carlos University</i>	Michael Wirth, <i>University of Guelph</i>
Jeffrey Stone, <i>Pennsylvania State University</i>	Ian Utting, <i>University of Kent at Canterbury</i>	Lee Wittenberg, <i>GW Software</i>
Christopher Stone, <i>Harvey Mudd College</i>	Jan Vahrenhold, <i>Technische Universität Dortmund</i>	Walter Wolf, <i>Rochester Institute of Technology</i>
Fred Strickland, <i>South University</i>	Ignatios Vakalis, <i>California Polytechnic State University</i>	Rosalee Wolfe, <i>DePaul University</i>
Catherine Stringfellow, <i>Midwestern State University</i>	David Valentine, <i>Slippery Rock University</i>	Christine Wolfe, <i>Ohio University</i>
Lena Stromback, <i>Lingopoint University</i>	Robert Van Camp, <i>Marietta College</i>	David Wolff, <i>Pacific Lutheran University</i>
Deborah Sturm, <i>College of Staten Island CUNY</i>	Tammy VanDeGraft, <i>University of Portland</i>	Greg Wolfe, <i>Grand Valley State University</i>
Dario Suarez Gracia, <i>Universidad de Zaragoza</i>	Jorge Vasconcelos, <i>Johns Hopkins University</i>	Steven Wolfman, <i>University of British Columbia</i>
Jose Such, <i>Technical University of Valencia</i>	Troy Vasiga, <i>University of Waterloo</i>	Ursula Wolz, <i>The College of New Jersey</i>
Leigh Ann Sudol, <i>Carnegie Mellon University</i>	Steven Vegdahl, <i>University of Portland</i>	Gary Ka Wai Wong, <i>The Community College at Lingnan University</i>
Wayne Summers, <i>Columbus State University</i>	J. Angel Velazquez-Iturbe, <i>Universidad Rey Juan Carlos</i>	David Womacot, <i>Haverford College</i>
Ken Surendran, <i>Southeast Missouri State University</i>	Gabriela Vilanova, <i>National Patagonia Austral University</i>	Karl Wurst, <i>Worcester State University</i>
Hengky Susanto, <i>University of Massachusetts Lowell</i>	Tamar Vilner, <i>The Open University of Israel</i>	T. Andrew Yang, <i>University of Houston - Clear Lake</i>
Jerry Talton, <i>Stanford University</i>	Kimberly Voll, <i>Univ. of British Columbia</i>	Arthur Yanushka, <i>Christian Brothers University</i>
Joo Tan, <i>Kutztown University</i>	Ken Vollmar, <i>Missouri State University</i>	Ken Yasuhara, <i>University of Washington</i>
Yonglei Tao, <i>Grand Valley State University</i>	David Voorhees, <i>Le Moyne College</i>	Dowming Yeh, <i>National Kaohsiung Normal University</i>
Blair Taylor, <i>Towson University</i>	Paul Wagner, <i>University of Wisconsin - Eau Claire</i>	Juang Yih-Ruey, <i>Jinwen University of Science & Technology</i>
Carol Taylor, <i>Eastern Washington University</i>	Sally Wahba, <i>Clemson University</i>	Duane Yoder, <i>University of West Georgia</i>
James Teresco, <i>Siena College</i>	Gursimran Walia, <i>North Dakota State University</i>	Alison Young, <i>Christchurch Polytechnic Institute of Technology</i>
Allison Elliott Tew, <i>Georgia Institute of Technology</i>	Henry Walker, <i>Grimmell College</i>	Benjamin Yu, <i>UBC</i>
William Thacker, <i>Winthrop University</i>	Xinli Wang, <i>Michigan Technological University</i>	Kwok-Bun Yue, <i>University of Houston - Clear Lake</i>
Soo Than, <i>Virginia Military Institute</i>	Yi Wang, <i>City University of Hong Kong</i>	Timothy Yuen, <i>UTSA</i>
Rebecca Thomas, <i>Bard College</i>	Stan Warford, <i>Pepperdine University</i>	Marsha Zaidman, <i>University of Mary Washington</i>
Megan Thomas, <i>California State University Stanislaus</i>	Richard Wasniowski, <i>California State University Channel Islands</i>	Carol Zander, <i>University of Washington, Bothell</i>
Stan Thomas, <i>Wake Forest University</i>	Thomas Way, <i>Villanova University</i>	Alan Zaring, <i>Ohio Wesleyan University</i>
Alfred Thompson, <i>Microsoft</i>	Scott Weiss, <i>Mt. St. Mary's University</i>	Julie Zelenski, <i>Stanford University</i>
Jodi Tims, <i>Baldwin-Wallace College</i>	Emily Wenk, <i>Penn State York</i>	Dean Zeller, <i>Art Institute of Jacksonville</i>
Fergus Toolan, <i>University College Dublin</i>	Linda Werner, <i>University of California Santa Cruz</i>	Jinghua Zhang, <i>Winston-Salem State Univ.</i>
Gloria Childress Townsend, <i>DePauw University</i>	Roger West, <i>University of Illinois at Springfield</i>	Uta Ziegler, <i>Western Kentucky University</i>
Christian Treffitz, <i>Grand Valley State University</i>	Suzanne Westbrook, <i>The University of Arizona</i>	Jill Zimmerman, <i>Goucher College</i>
Deborah Trytten, <i>University of Oklahoma</i>	Curt White, <i>DePaul University</i>	Guy Zimmerman, <i>Bowling Green State University</i>
George Tsiknis, <i>University of British Columbia</i>		Daniel Zingaro, <i>University of Toronto</i>
		Fani Zlatarova, <i>Elizabethtown College</i>

6.4 Otros méritos

6.4.1 Estancias en centros de I+D+i públicos o privados



UNIVERSIDAD DE ZARAGOZA

Universidad de Zaragoza
Registro General
Salida NE: BEI-08201
Fecha: 19/06/2007

Zaragoza, 18 de junio de 2007

N. Ref.: VICERRECTORADO DE INVESTIGACIÓN / SERVICIO DE GESTIÓN DE LA INVESTIGACIÓN

SUÁREZ GRACIA, DARÍO

INFORMÁTICA E INGENIERÍA DE SISTEMAS

CENTRO POLITÉCNICO SUPERIOR

Asunto: Concesión de Estancia Breve de Investigación del Programa de Formación de Personal Investigador para el año 2007.

En relación con las ayudas complementarias para Estancias Breves de investigación en España y en el extranjero, se ha publicado en el Boletín Oficial del Estado, de 18 de junio de 2007, Resolución de 12 de abril de 2007 de la Secretaría de Estado de Universidades e Investigación, por la que se le concede 152 días de estancia solicitada a:

PAÍS: CANADÁ

CENTRO DE ACOGIDA: UNIVERSITY OF TORONTO

teniendo en cuenta la autorización de su Director de Investigación,

ESTE VICERRECTORADO da su conformidad para que pueda realizar la mencionada estancia.





UNIVERSIDAD DE ZARAGOZA

Zaragoza, 30 de enero de 2008

N. Ref.: VICERRECTORADO DE
INVESTIGACIÓN / SERVICIO DE GESTIÓN DE
LA INVESTIGACIÓN

SUÁREZ GRACIA, DARÍO

INFORMÁTICA E INGENIERÍA DE SISTEMAS

CENTRO POLITÉCNICO SUPERIOR

Universidad de Zaragoza
Registro General
Salida №: GEN-002130
Fecha: 01/02/2008

Asunto: Concesión de Estancia Breve de Investigación del Programa de Formación de Personal Investigador para el año 2008.

En relación con las ayudas complementarias para Estancias Breves de investigación en España y en el extranjero, se ha publicado en el Boletín Oficial del Estado, de 29 de enero de 2008, Resolución de 27 de diciembre de 2007 de la Secretaría de Estado de Universidades e Investigación, por la que se le concede 138 días de estancia solicitada a:

PAIS: ESTADOS UNIDOS

CENTRO DE ACOGIDA: UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

teniendo en cuenta la autorización de su Director de Investigación,

ESTE VICERRECTORADO da su conformidad para que pueda realizar la mencionada estancia.

EL VICERRECTOR DE INVESTIGACIÓN, DESARROLLO
E INNOVACIÓN

José Ángel Villar Rivacoba



HiPEAC
EU ICT-217068



Collaboration Grant Report

Between:

	Institution	Name
The student	Universidad de Zaragoza	Dario Suárez Gracia
The student's advisor/manager	Universidad de Zaragoza	Víctor Viñals Yúfera
The host	FORTH	Manolis Katevenis
The HiPEAC Coordinator	UGent	Koen De Bosschere

Actual start date of the collaboration: 12/07/2009

Actual end date of the collaboration: 15/10/2009

Title of the collaboration:

This report summarizes the work and results of the collaboration grant.
The host hereby confirms that the student was present at the host's sites in the above mentioned period, and has completed the work as agreed upon. He confirms that this report is true, actual and written by the student.

The report should include, but is not limited to, the following sections:

- An overview of the activities done at the host's site
- Summary of research results, publications, project proposals,... that were produced in relation to the collaboration grant
- Future plans for collaboration if any
- Copies of travel tickets as proof of the above mentioned start and end dates.

For agreement,

The student,

(Date+signature)

The host

M. Katevenis
11 Nov. 2009

The student

The host



Qualcomm Incorporated
3165 Kifer Road
Santa Clara, CA 95051

January 16, 2012

To whom it may concern:

This is to confirm that Dario Suárez Gracia was a research intern at Qualcomm's Bay Area Research and Development Center (BARD) from June 26 to December 16, 2011. During this time he worked under my guidance on runtime libraries, system software, and hardware-software interfaces to eliminate the overhead of parallel execution and reduce power consumption through optimizations that exploit runtime behavior of mobile applications.

If you need any additional information do not hesitate to contact me at +1 408 533 9472.

Sincerely,

Calin Cascaval
Director, Engineering
Qualcomm, Inc.



969 Commercial St
Palo Alto, CA 94303
T: (650) 681-9257

October 4th, 2017

Re: Mr Dario Suárez Gracia

To whom it may concern:

This is to confirm that Mr Dario Suárez Gracia has been a visiting scientist at Eonite Perception Inc. during the following periods:

- From June 16, 2016 to August 30, 2016
- From July 14, 2017 to August 14, 2017

During his visit, Dario was collaborating with the Computer Vision and Robotics Department at our headquarters in Palo Alto, California.

Should you need further information, please do not hesitate to contact the undersigned.

Sincerely,

A handwritten signature in black ink, appearing to read "Yousri Helmy".

Mr Youssri Helmy, CEO

6.4.2 Resumen de otros méritos



MINISTERIO
DE EDUCACIÓN
Y CIENCIA

SECRETARÍA DE ESTADO DE UNIVERSIDADES E INVESTIGACIÓN
DIRECCIÓN GENERAL DE INVESTIGACIÓN

PROGRAMA DE BECAS PREDOCCTORALES
DE FORMACIÓN DE PERSONAL INVESTIGADOR
(FPI)

CREDENCIAL DE BECARIO

Por Resolución de 29 de julio de 2005, del Secretario de Estado de Universidades e Investigación del Ministerio de Educación y Ciencia, le ha sido adjudicada una beca Predoctoral de Formación de Personal Investigador asociada al proyecto de investigación TIN2004-07739-C02-02, con efectos económicos y administrativos desde el 01/08/2005 hasta el 31/07/2009, con una dotación económica de 1.100 euros brutos mensuales y demás prestaciones establecidas en la Orden ECI/4484/2004, de 29 de diciembre de 2004 (B.O.E. de 14 de enero de 2005).

Esta convocatoria de becas está cofinanciada por el Fondo Social Europeo.

Madrid, 1 de agosto de 2005

EL Subdirector General de
Formación y Movilidad del Personal Investigador
P.A. El Jefe del Servicio de Becas



Fdo.: Eloy Morón Román

SUÁREZ GRACIA, DARIO
Referencia de la beca:BES-2005-10385

N. 765



EXCMO. SR. RECTOR MAGNÍFICO:

INVESTIGADOR PRINCIPAL
APELLIDOS Y NOMBRE: Viñals Yúfera, Victor

EMPLEO:
Profesor Titular de Universidad

DESTINO CENTRO: CPS
DEPTO.: Informática e Ing. de
Sistemas

EXPONE: que tiene suscrito un proyecto/contrato de investigación con:

Ministerio de Ciencia y Tecnología. Proyecto TIC2001-0995-C02-02 (Cod. UZ 287-43/1)
(Computación de altas prestaciones III. Jerarquía de memoria de altas prestaciones)

para la realización de

Diseño, evaluación e implementación de estructuras de memoria cache para
microporcesadores de altas prestaciones

y precisa para ello

<input checked="" type="checkbox"/> Becario Nº 1	<input type="checkbox"/> Personal laboral Nº _____	En Jornada	
Desde	Desde _____	Hasta _____	Reducida <input type="checkbox"/> Completa <input type="checkbox"/>
16 3 2004	_____	_____	_____
Hasta	_____	_____	_____
27 12 2004	_____	_____	_____
Cantía mensual de la beca:	Descripción del puesto a ocupar: _____		
720 euros	Del grupo _____	En el caso de jornada reducida: Días: _____ Horario: _____	

PARA LO CUAL PROPONE (BECARIO DE COLABORACIÓN) A:

D. Dario Suárez Gracia con D.N.I. 18443694V

Titulación Ingeniero en Informática

Fecha de nacimiento 25/7/80 Lugar de nacimiento Madrid

Dirección Vía Universitas 4 7º2º, Zaragoza Teléfono 676855833 (UZ 2472)

Banco y c/c (indicar los 20 dígitos) Ibercaja (Oficina Principal, Teruel) 2085 3852 1303 0062 9876

Por todo lo expuesto a V.M.E.,

SOLICITA: se efectúen los trámites pertinentes para dar cumplimiento a lo expuesto, minorando el total de los costos de los nombramientos o contratos, del presupuesto asignado para la realización del citado proyecto/contrato. Asimismo manifiesta que actuará como tutor del becario/contratado propuesto, efectuando el seguimiento de la labor realizada.

Zaragoza, 15 de Marzo de 2004

Fdo.: Victor Viñals Yúfera

zación previstos en la Ley 16/2002, de 1 de julio, de preventión y control integrados de la contaminación. El titular de la actividad deberá solicitar la renovación de la Autorización Ambiental Integrada 10 meses antes como mínimo del vencimiento del plazo de vigencia de la actual.

Condicionado 15º

La instalación se ejecutará y la actividad se desarrollará necesariamente de acuerdo a los documentos que obran en el expediente y a lo establecido en la presente Resolución, y en la legislación vigente.

Condicionado 16º

Previo al comienzo de la actividad, se deberá comprobar el cumplimiento del condicionado de la presente Resolución, mediante visita de inspección a las instalaciones, no siendo efectiva la autorización en tanto no se disponga de la correspondiente notificación de efectividad y del número de autorización asignado.

Condicionado 17º

A efectos del cumplimiento de lo establecido en el condicionado anterior, el titular de la instalación podrá notificar la fecha prevista de inicio de la actividad o solicitar la inspección al Órgano Competente Ambiental de la Comunidad Autónoma de Aragón, considerando que el plazo desde la publicación de la autorización y el comienzo de la actividad deberá ser inferior a cuatro años, de otra forma la autorización quedará anulada y sin efecto.

Condicionado 18º

La empresa remitirá previo al inicio de la modificación de las instalaciones de pintura, un Plan de actuación temporal, con especial referencia a los focos de emisión de Compuestos Orgánicos Volátiles (COV's).

Condicionado 19º

Esta Resolución se notificará en la forma prevista en la Ley 30/1992, de 26 de noviembre, de Régimen Jurídico de las Administraciones Públicas y del Procedimiento Administrativo Común, modificada por la Ley 4/1999, de 13 de enero y se publicará en el «Boletín Oficial de Aragón» de acuerdo con lo establecido en el artículo 23.3 de la Ley 16/2002, de 1 de Julio, de prevención y control integrados de la contaminación.

Contra la presente Resolución, que no pone fin a la vía administrativa, y de conformidad con lo establecido en los artículos 107 y 114 de la Ley 30/1992, de 26 de noviembre, de Régimen Jurídico de las Administraciones Públicas y del Procedimiento Administrativo Común, modificada por la Ley 4/1999, de 13 de Enero podrá interponerse recurso de alzada ante el Excmo. Sr. Consejero de Medio Ambiente, en el plazo de un mes desde la recepción de la presente notificación, sin perjuicio de cualquier otro que pudiera interponerse.

Zaragoza a 24 de enero de 2005.

La Directora General de Calidad Ambiental,
MARTA PUENTE ARCOS

DEPARTAMENTO DE CIENCIA, TECNOLOGÍA
Y UNIVERSIDAD

RESOLUCIÓN de 21 de enero de 2005, de la Dirección General de Investigación, Innovación y Desarrollo, por la que se conceden tres ayudas destinadas a la formación de personal Investigador.

Por Orden de 26 de julio de 2004, del Departamento de Ciencia, Tecnología y Universidad se aprueban las bases de la convocatoria de ayudas destinadas a la formación de personal de investigador. Por Resolución de 18 de diciembre de 2004, de la Dirección General de Investigación, Innovación y Desarrollo, se resuelve dicha convocatoria, concediéndose 40 becas predoctorales y nombrándose suplentes para las diferentes áreas de investigación en la que estén designados y en el orden de preferencia establecido.

Habiéndose producido la renuncia de D. David Portolés Rodríguez en el área de tecnología, de D^a Laura Pastor Campo en el área de ciencias experimentales y matemáticas y de D. Eric Jean Philippe Martínez en el área de ciencias biomédicas, procede, en consecuencia, modificar la citada Resolución, dando de baja a D. David Portolés Rodríguez, a D^a Laura Pastor Campo y a D. Eric Jean Philippe Martínez, designando los nuevos beneficiarios de las becas.

En su virtud, y de acuerdo con la Base 4º.2.2. de la Orden de 26 de julio de 2004, resuelvo:

Primero: Declarar la pérdida de eficacia de la concesión de las mencionadas ayudas a D. David Portolés Rodríguez, D^a Laura Pastor Campo y D. Eric Jean Philippe Martínez.

Segundo: Conceder la beca de investigación a D. Darío Suárez Gracia en el área de tecnología, a D^a. Ana Villares Garicóchea en el área de ciencias experimentales y matemáticas y D^a María Bes Félix en el área de biomédicas suplentes de la lista priorizada en las respectivas áreas, financiándose con cargo a la aplicación presupuestaria 17.03.542.3.480.00 B Fomento a la Investigación-Fondo Social Europeo-

Tercero: El periodo de disfrute tendrá efectos desde el día de la aceptación de la misma por parte del beneficiario y finalizará el 31 de diciembre de 2005.

Contra la presente resolución, que no pone fin a la vía administrativa, podrá interponerse recurso de alzada ante el Consejero de Ciencia, Tecnología y Universidad en el plazo de un mes computado a partir del día siguiente al de su publicación en el «Boletín Oficial de Aragón», de conformidad con lo dispuesto en los artículos 107 y 114 de la Ley 30/1992, de 26 de noviembre, de Régimen Jurídico de las Administraciones Públicas y del Procedimiento Administrativo Común, en la redacción dada por Ley 4/1999, de 13 de enero, sin perjuicio de la interposición de cualquier otro que legalmente proceda.

Zaragoza, 21 de enero de 2005.

El Director General de Investigación,
Innovación y Desarrollo,
JESÚS SANTAMARÍA RAMIRO

IV. Administración de Justicia

ANUNCIO de la Audiencia Provincial, relativo a recurso de apelación 494/2004.

El Ilmo. Sr. don Pedro Antonio Pérez García, Presidente de la Sección Quinta de la Audiencia Provincial de Zaragoza, hace saber: Que en esta Sala se sigue recurso de apelación (LECN) 0000494/2004, diramante de Procedimiento Ordinario 0000236/2003 seguido ante el Jdo. Primera Instancia número 12 de Zaragoza, entre la comunidad de Propietarios calle Esculor Moreto, número 23 de Zaragoza, representado por el procurador Sr. Galón Carrillo, contra don José María Gabasa Cabello y otros; y contra Iberlatre Royal, S. L., no comparecido en esta segunda instancia en el mencionado rollo, con fecha 24 de enero de 2005 recayó sentencia cuya parte dispositiva dice:

«Que estimando parcialmente el recurso de apelación interpuesto por don José María Gabasa Cabello contra la Sentencia dictada por el Juzgado de Primera Instancia número 12 de Zaragoza, y recaída en el juicio declarativo ordinario número 236/2003, con revocación parcial de la misma se absuelve al recurrente de la condena a reparar la fisuración aparecida en el antepiecho del hueco de fachada.

No se hace una especial imposición de las costas causadas en ninguno de las dos instancias».

Y para que sirva de notificación a la apelada rebelde «Iberlatre Royal, S. L.» libro el presente edicto que se insertará en el

European Network of Excellence on High Performance and Embedded Architecture and Compilation



Home | Contact | Help | Log in

HiPEAC

- # About
- # Calendar
- # Conference
- # HiPEAC1 research
- # Clusters
- # HiPEAC2 research
- # Clusters
- # Journal
- # Mailing List
- # Members
- # Newsletter
- # Phd students
- # Platforms
- # Related Projects
- # Roadmap
- # Summer School
- # Web Seminars

User login

Username: *

Password: *

Create new account
 Request new password

Home

Adaptive data caches

Description:

Processor performance closely follows the performance of its memory hierarchy. Inside the hierarchy, the first level data cache plays a critical role because the data they supply can lie in the critical computation path. Cache size and latency are constant irrespective of the working set it tries to store, and this is the point we want to study in the proposed collaboration between the University of Zaragoza and the University of Toronto.

A per-program tuned cache could be desirable for each program, or even better, for each execution phase. For example, when comparing three L1 data caches: 8 KB 1-cycle, 16 KB 2-cycle and 32 KB 3-cycle using SPEC CPU 2000, there is not a clear winner. If we focus on L2 caches we can not reach a single good configuration for any application at all phase, either. Assuming a uniform memory access pattern, a 16 KB 3-cycle cache, a 1 MB 14-cycle L2, and a uniform load access pattern, then in one third of accesses we will have to look up L2 for just 16 KBytes. So, it is clear that the balance between latency and cache size is very difficult to achieve. Of course, a large cache with a small latency is infeasible due to technological limitations, mostly energy consumption.

Such an adaptive cache should provide a fine-grain size-latency steps with a reduced energy consumption and a sustainable bandwidth. In order to meet these requirements a partitioned organization could be employed, each partition being set with a latency as small as possible. To save energy partitions could be switched on and off, since the rise of static consumption due to scaling makes essential the ability of turning completely off unused processor parts.

There are very good proposals of adaptive, reconfigurable, caches but most of them are very complex heuristics like that by (Balasubramanian et al., 2003) or (Zhang et al., 2003) in a different context wire delay dominated cache NCUA (Korpi et al., 2003) propose some techniques that may be employed for our purposes like block migration within the cache.

One student from Zaragoza, Dario Suárez Gracia, will stay within Prof. Moshovos's group during 2007. The main goal of this collaboration is the proposal of a cache microarchitecture which follows the previously described principles.

Nature:
3 months collaboration grant
Total funding:
Requested: € 5000
Including fellowship funding:
Requested: € 0
Description of how the funding will be used:
Travel and living in Toronto from January 2007 until June 2007.
Travel from Zaragoza to Toronto: 1500 €
Rent a room in Toronto: 6'400 = 2400 €
Maintenance: 6'350 = 2100 €
Duration of the funding:
Requested: 6 month(s)
Participating members:
SUÁREZ GRACIA Dario (**University of Zaragoza**) (-phd student-)
MONREAL Teresa (**University of Zaragoza**) (-colleague-) VIFALS Victor (**University of Zaragoza**) (-member-)
Other people collaborating:
Andreas Moshovos, Assistant Professor, University of Toronto.

By admin at 17/01/2008 - 14:37

Announcements

- # Current calls
 - # HiPEAC 2009 Conference
 - # 8th HiPEAC Industrial Workshop
 - # Computing Systems Week Barcelona
 - # Phd internships
- # Conferences
- # News
- # Seeking/Opening Positions




<http://www.hipeac.net/node/2135>

05/05/2008



Published on *HiPEAC* (<http://www.hipeac.net>)

[Your funding requests](#) > [VLSI implementation of Light NUCAs](#) > [VLSI implementation of Light NUCAs](#)

VLSI implementation of Light NUCAs

By [dario](#)

Created 18/05/2009 - 17:27

Submitted by [dario](#) [1] on Mon, 18/05/2009 - 17:27

Applicant's name

Darío Suárez Gracia

[Request Pre-financing](#) [2]

Applicant's institution

University of Zaragoza

Host institution

FORTH

Host researcher

[Manolis Katevenis](#) [3]

Estimated start date

2009-07-01

Research proposal

My PhD thesis focuses on memory hierarchies, namely on the cache levels close to the processor. In order to improve their performance, we have proposed a novel organization named Light NUCA (L-NUCA) [2]. L-NUCA leverages the distributed organization and the block migration techniques from the NUCA design [1], but completely redesigns the interconnection network, replacing the 2-D mesh by three specialized networks tailored to the three key cache operations: lookup, line transport, and replacement. The use of these Network-On-Cache mechanisms, such us headerless messages or implicit routing, reduces network latency to the bare minimum and enables to fit within a single cycle a small-sized cache access and one-hop routing.

We have developed a very accurate micro-architectural simulation environment of L-NUCAs including the flow control, congestion, arbitration, etc. Delay and energy

modelling have been performed with Cacti and Orion, and even some HSPICE simulations have been done for estimating the delay of the most critical network components. Nevertheless, due to the lack of expertise in RTL implementation, we have not built the RTL model of L-NUCAs that will provide more accurate estimations of area, delay and energy.

The Computer Architecture and VLSI Systems (CARV) Laboratory at FORTH is well-known for its expertise in both interconnection networks and VLSI design; hence, this Laboratory is a very good place for an intern aimed at the VLSI implementation of the network mechanisms, the cache tiles, the processor interface, and the overall control.

The expected benefits from this stay are twofold. On one hand, we expect to publish some interesting results from the accurate modelling with a novel low-complexity topology for L-NUCA. On the other hand, the discussion of L-NUCA ideas with FORTH members will open new opportunities to study and increase the interaction between both institutions (FORTH and the Computer Architecture Group at the University of Zaragoza).

Summarizing, the goal of the intern will be the implementation of an RTL model of the L-NUCAs. The work will be done within the frameworks of the Synopsys toolset for the hardware description and with Cadence SoC Encounter for placement and routing. The implementation will include a new topology for transporting lines from the cache tiles to the processor and a new block placement policy designed to reduce the energy consumption. The new topology reduces the complexity of the network by decreasing the number of output links in the nodes and the policy saves energy by restricting the tiles in which a block can be present. Neither proposals have been published yet but are already implemented in our simulation infrastructure.

References:

- [1] An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. Changkyu Kim, Doug Burger, and Stephen W. Keckler. 10th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2002.
- [2] Light NUCA: A Proposal For Bridging The Inter-Cache Latency Gap. Dario Suárez, Teresa Monreal, Fernando Vallejo, Ramón Beivide and Víctor Viñals. 12th Design, Automation & Test Europe in Europe (DATE), 2009.

Rationale for the collaboration grant

VLSI models of microarchitectural proposals can generate added-value because they provide very accurate area, delay, and power estimations. In our particular case, VLSI modeling of an L-NUCA, we assume that the former holds and the implementation will also serve as a proof-of-concept of the single cycle integration of a cache access plus one-hop routing. We think 3 months are enough for the planned work. The knowledge learned at FORTH and the subsequent collaboration, with the support of the Spanish National Center of Microelectronics -whom we are already collaborating with-provide potential for success to this project in the short and long term.

Evaluation

Status

Accepted

Decision

Congratulations. You will be contacted in August for the administrative details.

Attachment **Size**

[2009-05-dsuarez.pdf](#) [4] 66.7 KB

For remarks about the website, contact webmaster[at]hipeac[dot]net.

Source URL: <http://www.hipeac.net/node/2854>

Links:

- [1] <http://www.hipeac.net/user/77>
- [2] http://www.hipeac.net/collaboration/process_reimbursing/2854
- [3] <http://www.hipeac.net/user/10>
- [4] <http://www.hipeac.net/system/files/2009-05-dsuarez.pdf>



Published on *HiPEAC* (<http://www.hipeac.net>)

Your funding requests > Memory Hierarchies in complex SoCs: the Snapdragon case > Memory Hierarchies in complex SoCs: the Snapdragon case

Memory Hierarchies in complex SoCs: the Snapdragon case

By [dario](#)

Created 04/10/2010 - 17:03

Submitted by [dario](#) [1] on Mon, 04/10/2010 - 17:03

Proposed by

[Claim reimbursement](#) [2]

[Darío Suárez Gracia](#) [1]

Cluster

Multi-core architecture

Period

June 2010 to November 2011

Motivation

1) The period of your collaboration

June 2011 - November 2011. The delay for obtaining an US VISA and Darío Suárez Gracia's teaching duties from February to June 2011 make it impossible to schedule the internship for an earlier date.

2) Work to be performed

The Qualcomm Snapdragon is one of the most successful System-on-a-Chip (SoC) in the market. The Snapdragon SoC combines a high-performance, proprietary application processor (Scorpius) with state-of-the-art communication, 3D graphics, video and sound processors [1]. One key reason for this success is its very low power consumption, around .35 mW/MHz.

SoCs such as the Snapdragon are very complicated heterogeneous, multicore systems. They include many processors, and multiple memories disseminated all over the SoC. For example, the applications processor in the QSD8660 has two Scorpius cores with several levels of cache. A high-bandwidth AXI crossbar connects the Scorpius cores with a memory subsystem composed either by stacked, on-chip, or off-chip low power DRAMs. The ARM-based modem processor, also connected to the AXI crossbar, is the main component of the communication subsystem, which also

include GPS, Wi-Fi and Bluetooth chips. Many other coprocessors connect to the crossbar: cryptography, graphics, audio and video, etc. Finally, a slower bus connects the rest of the devices, such as the touch screen, memory cards, etc.

The aim of this internship is to characterize the performance and the energy consumption of the Snapdragon's complex memory hierarchy in the context of modern mobile applications. The heterogeneity of the devices and the patterns exhibited by some applications require fast and multiple transfers among them. For example, if we get lost in a city during a conference, we use our smartphone to learn our location and the directions to the conference hotel. This simple operation requires data transfers from the GPS (location information) and the 3G modem (maps sent by an online provider) to the application processor, which will calculate the shortest route and will send it to the graphic subsystem. In addition, the application processor will generate voice commands and will send them to the audio processor. While all of this occurs, the communications subsystem must be available in case there is an incoming call.

Clearly, the organization and management policies of the interconnection networks and the memory is very complex, and plays a vital role in the performance and the energy consumption of the SoC. Our work will consist on analyzing different application patterns such as the one described above. We will gain insights about the memory hierarchy and the interconnections and will propose alternative designs that either improve performance, reduce power, or both. For example, the interconnection network could be optimized for common use cases. A key part of our research will include the analysis of L2-L3 cache sharing policies across processors in the SoC and the control of the separated power plane of the CPUs and caches. Using separate power planes would help minimizing power consumption when large amounts of data are transferred between devices and the application processors are stalled.

Our group at the University of Zaragoza has done significant research on Non-Uniform Cache Architectures (NUCA) [2, 3]. Thus, we will study whether NUCA is a viable alternative for SoC design. To the best of our knowledge, there is little prior work on this area [4], especially with the constraints imposed by these environments are taken into consideration (quality of service, power).

We expect several benefits from this collaboration grant. First, the HiPEAC participants from the University of Zaragoza will get an industrial perspective of research with the market leader in mobile technology. Second, we expect to publish some results from the internship, and in the long term, our group will have a deeper understanding of what are the problems to be faced in the memory hierarchies of high-end System-On-Chips. And third, we would start a collaboration between our group in Zaragoza and Qualcomm. We expect that this collaboration will continue after the internship.

References

- [1] Two-Headed Snapdragon Takes Flight. L. Gwennap. Microprocessor Report. July 2010. Pages 23-27.
- [2] Light NUCA: A Proposal For Bridging The Inter-Cache Latency Gap. D. Suárez, T. Monreal, F. Vallejo, R. Beivide, and V. Viñals. 12th Design, Automation & Test Europe in Europe (DATE). 2009.
- [3] LP-NUCA: a Low-Power Implementation of Medium-Sized Caches Leveraging Networks-in-Cache. D. Suárez, G. Dimitrakopoulos, T. Monreal, M. Katevenis, and V. Viñals. Submitted to a Journal.
- [4] No Cache-Coherence: A Single-Cycle Ring Interconnection for Multi-Core L1-NUCA Sharing on 3D Chips. Shu-Hsuan Chou, Chien-Chih Chen, Chi-Neng Wen, Yi-Chao Chan, Tien-Fu Chen, Chao-Ching Wang and Jinn-Shyan Wang. 46th Annual

Design Automation Conference (DAC). 2009.

3) A statement of why the collaboration is useful

This collaboration between the Qualcomm Bay Area Research Center (BARD) and the University of Zaragoza will provide Mr. Dario Suárez-Gracia with the opportunity to learn how modern SoC are designed. Such knowledge will allow our research group in Zaragoza to enter new research areas. In addition, it will create new links between the HiPEAC community and one of the main players in the embedded market. Finally, the quality of the host group and their very successful intern program suggest that the collaboration will be fruitful.

4) Participants:

From Qualcomm Bay Area Corporate Research and Development (BARD):
Călin Cașcaval (Director of Engineering)
Pablo Montesinos-Ortego (Engineer, Senior)

From the University of Zaragoza
Dario Suárez-Gracia
Víctor Viñals-Yúfera

Estimated budget (in Euro)

15000

Fund use

Travel expenses (trains + flight from Zaragoza, Spain to San Jose, USA): 2000 €
Housing + Car rental + living expenses: 12000 € Health Insurance: 700 € VISA cost
(including trip to Madrid for the interview at the Embassy): 300 € Total: 15000 €

Participants

- Dario Suárez Gracia [1] (Progress indicator) [3]
- Víctor Viñals-Yúfera [4] (Progress indicator) [5]

Evaluation**Status**

Accepted

Decision

Congratulations

Approved budget (in Euro)

15000

Decision made by[HiPEAC Administrator](#) [6]**Reimbursement Status****Current status of reimbursement:**

Registered, payment pending

Actual cost to HiPEAC

2404.21 EUR

The actual cost for the HiPEAC network and the calculated amount above may differ due to exchange rates.

For remarks about the website, contact webmaster[at]hipeac[dot]net.

Source URL: <http://www.hipeac.net/node/3770>**Links:**

- [1] <http://www.hipeac.net/user/77>
- [2] http://www.hipeac.net/reimbursing/process_reimbursing/3770/77
- [3] http://www.hipeac.net/members_overview/members_detail/77
- [4] <http://www.hipeac.net/user/1016>
- [5] http://www.hipeac.net/members_overview/members_detail/1016
- [6] <http://www.hipeac.net/user/1>



Departamento de Ciencia, Tecnología y
Universidad

SUAREZ GRACIA, Darío
CL Rosario 26 bis
44003 Teruel

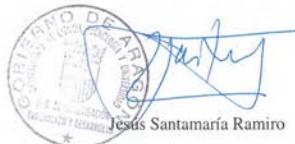
Para su conocimiento, le comunico que por Resolución de 27 de noviembre de 2003, del Director General de Investigación, Innovación y Desarrollo, se le ha concedido una ayuda, destinada a sufragar los gastos de matrícula en estudios de doctorado para la obtención de la suficiencia investigadora con las condiciones establecidas en el punto sexto de la convocatoria realizada por Orden de 18 de septiembre de 2003, del Departamento de Ciencia, Tecnología y Universidad.

Contra dicha resolución, que no agota la vía administrativa, podrá interponerse recurso de alzada ante la Excmo. Sra. Consejera de Ciencia, Tecnología y Universidad en el plazo de un mes contado a partir del día siguiente a la publicación de esta Resolución en el Boletín Oficial de Aragón (B.O.A. de 5 de diciembre de 2003).

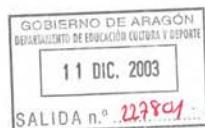
Para iniciar desde el Servicio de Investigación de esta Dirección General los trámites de pago de la ayuda concedida deberá remitir (Avda. Gómez Laguna 25, 2^a planta, 50009-Zaragoza), debidamente cumplimentado el impreso que se le adjunta antes del 30 de diciembre del presente año. Asimismo, debe ponerse en contacto con Tercer Ciclo de la Universidad de Zaragoza para la formalización de su matrícula para el curso 2003-2004.

Zaragoza, 9 de diciembre de 2003

EL DIRECTOR GENERAL DE INVESTIGACIÓN,
INNOVACIÓN Y DESARROLLO



Jesús Santamaría Ramiro





SUAREZ GRACIA, Dario
CL Rosario 26 bis
44003 Teruel

Habiendo informado favorablemente la Universidad de Zaragoza, conforme a lo dispuesto en la base 6º.3 de la convocatoria de ayudas para la obtención del "Diploma de Estudios Avanzados" del año 2003, destinadas a sufragar los gastos de matrícula en estudios de doctorado, realizada a través de la Orden de 18 de septiembre de 2003, del Departamento de Ciencia, Tecnología y Universidad (B.O.A. del 8 de octubre de 2003), del cumplimiento de los requisitos establecidos en la base 11º.2 de la citada convocatoria: en el segundo año (periodo investigador) será necesario haber superado la totalidad de los créditos en los que el beneficiario estuvo matriculado durante el primer año (Periodo Docente) utilizando la ayuda,

la Dirección General de Investigación, Innovación y Desarrollo ha resuelto:

Mantener durante el curso 2004-2005 la ayuda "Diploma de Estudios Avanzados"(Periodo Investigador)

Que se le concedió por Resolución de 27 de noviembre de 2003, de la Dirección General de Investigación, Innovación y Desarrollo (B.O.A. del 5 de diciembre de 2003).

Zaragoza, 17 de septiembre de 2004



Celia Martín Robledo

Workshop on Computer Architecture Education
BEST PAPER AWARD
2007

Presented to

Alicia Asín Pérez, Dario Suárez Gracia
and
Victor Viñals Yúfera

For their paper entitled

A Proposal to Introduce Power and Energy Notions
in Computer Architecture Laboratories

HiPEAC 2014

January 20-22, 2014 | Vienna Austria



HiPEAC '12 - Student Poster awards decided

The following students have won a student poster award:

Amira Mensi
Mihai Pricopi
Ricardo Nobre
Michael Lyons
Ye Gao
Alexandra Ferreron
Ali Bayrak
Akshatha Mulki Bhat



Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza



D. ANTONIO MUÑOZ PORCAR, Presidente de la Comisión de Evaluación y Control de la Docencia de la Escuela de Ingeniería y Arquitectura* de la Universidad de Zaragoza,

HACE CONSTAR:

Que, según los antecedentes obrantes en la Secretaría de este Centro, Don DARIO SUÁREZ GRACIA ha obtenido los siguientes resultados globales en la evaluación de su actividad docente, como profesor adscrito al área de Arquitectura y Tecnología de Computadores del departamento de Informática e Ingeniería de Sistemas, durante los períodos que se detallan a continuación:

CURSO	evaluación	CURSO	evaluación
2015-16	POSITIVA DESTACADA	2011-12	NO EVALUAR
2010-11	POSITIVA DESTACADA	2009-10	POSITIVA DESTACADA
2008-09	POSITIVA	2007-08	POSITIVA

-El procedimiento para la obtención de esta valoración se inició en 1988, cuando la Universidad de Zaragoza estableció un procedimiento general de Evaluación de la Docencia basado en una encuesta realizada por los estudiantes en la que se valoraban aspectos relacionados con el cumplimiento de las obligaciones docentes del profesor, el desarrollo de su actividad docente, la calidad de dicha docencia y las relaciones profesor-estudiante. A partir de esta información, las Comisiones de Control y Evaluación de la Docencia de cada centro emitían una valoración para la cual podían solicitar información adicional al profesor, al departamento y a los estudiantes.

La Comisión de Control y Evaluación de la Docencia de la Universidad, a la vista de la propuesta del centro, así como de toda la documentación recogida, emitía una valoración final (Positiva, Negativa, No-Negativa, Sin evaluación), de la cual informaba al centro.

-A partir del curso 2006-07 se produjo la renovación del proceso de evaluación para consolidar y mejorar este programa específico de evaluación de la calidad de la actividad docente y para su adaptación a la sistemática del Programa DOCENTIA, en el que participa la Universidad de Zaragoza, que supuso asimismo una leve corrección de la valoración final (Positiva destacada, Positiva o Negativa).

-Por último, el Acuerdo de Consejo de Gobierno de 23 de febrero de 2016, de aplicación a partir de la evaluación de la actividad docente del curso 2016/17, adapta este proceso a los nuevos procedimientos de evaluación y acreditación de las enseñanzas por parte de la Agencia de la Calidad y Prospectiva de Universidad de Aragón (ACPUA). En virtud del mismo, le corresponde a la Comisión de Calidad de la Actividad Docente de la Universidad (CCAD), la coordinación y seguimiento del proceso de evaluación de la actividad docente del profesorado a partir de los informes elaborados por las comisiones técnicas de evaluación.

Y, para que conste, a petición del interesado, y a los efectos oportunos, expido la presente certificación en Zaragoza, a día 08 de septiembre de 2017.

EL PRESIDENTE DE LA COMISIÓN DE CONTROL
Y EVALUACIÓN DE LA DOCENCIA DE LA EINA.

Fdo.: Antonio Muñoz Porcar.

Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

*Con fecha 15 de abril de 2011 se creó la Escuela de Ingeniería y Arquitectura (EINA), como resultado de la integración del Centro Politécnico Superior y la Escuela Universitaria de Ingeniería Técnica Industrial de Zaragoza.

Maria de Luna, 3 Edificio Torres Quevedo Campus Rio Ebro 50018-ZARAGOZA,



GERARDO SANZ SÁIZ, VICERRECTOR DE POLÍTICA ACADÉMICA y, por delegación del RECTOR (Resolución de 19 de mayo de 2016, BOA N°100 de 26 de mayo de 2016), PRESIDENTE DE LA COMISIÓN DE CALIDAD DE LA ACTIVIDAD DOCENTE,

HACE CONSTAR

Que, en cumplimiento de lo dispuesto en el Artículo 109 de los Estatutos de la Universidad de Zaragoza y en aplicación de la Resolución del Rector de la Universidad de Zaragoza de 5 de abril de 2017, por la que se aprueba el texto refundido de la normativa básica sobre el procedimiento y los criterios de valoración de la actividad docente del profesorado por parte de los estudiantes, la Comisión de Calidad de la Actividad Docente ha valorado los resultados de las encuestas de evaluación de la actividad docente cumplimentadas por los estudiantes, así como, en su caso, las alegaciones presentadas a dichos resultados y, de acuerdo con esta información, el resultado de la evaluación ha sido la siguiente calificación:

Darío Suárez Gracia

Curso 2017/2018 Positiva Destacada

Y para que conste y surta los efectos oportunos, expide el presente documento en Zaragoza, a fecha de la firma.

El Presidente de la Comisión de Calidad de la Actividad Docente

Gerardo Sanz Sáiz

(firmado electrónicamente, art. 3 L59/2003 y art. 26.2.e) L39/2015)



Copia auténtica de documento firmado digitalmente. Puede verificar su autenticidad en <http://verificad.unizar.es/service/digitalSignature/dc9e0371d797341ada3bc6d267db11b8>

CSV: dc9e0371d797341ada3bc6d267db11b8	Organismo: Universidad de Zaragoza	Página: 1 / 1
Firmado electrónicamente por	Cargo o Rol	Fecha
GERARDO SANZ SAIZ	EL PRESIDENTE DE LA COMISIÓN DE CALIDAD DE LA ACTIVIDAD DOCENTE	19/08/2019 11:13:00



Universidad
Zaragoza

**D.JUAN GARCÍA BLASCO, SECRETARIO GENERAL DE LA UNIVERSIDAD
DE ZARAGOZA,**

CERTIFICA:

Que, Don Darío Suárez Gracia con NIF: 18443694V, ha participado en la Convocatoria de Innovación Docente 2015-2016 de la Universidad de Zaragoza, dentro de la modalidad Programa de Incentivación de la Innovación Docente en la UZ (PIIDUZ) en calidad de participante del equipo del proyecto :

"iClickers para mejorar interactividad, atención y "feedback" en las clases" con identificador PIIDUZ_15_334 .

Y para que conste, una vez finalizado y cumpliendo los requisitos marcados, se expide en Zaragoza, a 9 de noviembre de 2016.

**Universidad**
Zaragoza

VIB^a
El Vicerrector de Política Académica

Fdo.:Gerardo Sanz Salz



D.JUAN GARCÍA BLASCO, SECRETARIO GENERAL DE LA UNIVERSIDAD
DE ZARAGOZA,

CERTIFICA:

Que, Don Dario Suárez Gracia con NIF: 18443694V, ha participado en la Convocatoria de Innovación Docente 2016-2017 de la Universidad de Zaragoza, dentro de la modalidad Programa de Incentivación de la Innovación Docente en la UZ (PIIDUZ) en calidad de coordinador principal del equipo del proyecto :

"Estudio y diseño de una plataforma común de trabajo para la mejora del aprendizaje en el Grado en Ingeniería Informática
" con identificador PIIDUZ_16_270 .

Y para que conste, una vez finalizado y cumpliendo los requisitos marcados, se expide en Zaragoza, a 28 de agosto de 2017.

A handwritten signature in black ink.



VºBº
El Vicerrector de Política Académica

A handwritten signature in black ink.

Fdo.:Gerardo Sanz Saiz



D. JUAN FRANCISCO HERRERO PEREZAGUA, SECRETARIO
GENERAL DE LA UNIVERSIDAD DE ZARAGOZA,

C E R T I F I C A:

Que, D. DARÍO SUÁREZ GRACIA, con NIF 18443694V, ha participado en la Convocatoria de Innovación Docente 2008-2009 de la Universidad de Zaragoza, Programa de Incentivación de la Innovación Docente, Línea 2: Proyectos de implantación de actividades de aprendizaje innovadoras en el ámbito de la docencia de una materia o asignatura específica; en calidad de miembro integrante del equipo del proyecto:

'REALIZACIÓN DE PROYECTOS CON UN ENTORNO BASADO EN EL PROCESADOR ARM COMO MÉTODO DE APRENDIZAJE EN LA ASIGNATURA DE LABORATORIO DE COMPUTADORES', con identificador PIIDUZ_08_2_246.

Y para que conste, cumplidos los requisitos marcados en dicha Convocatoria, se expide en Zaragoza, a 18 de julio de 2009.

Vº Bº
El Adjunto al Rector
para Innovación Docente



A blue ink signature of Javier Paricio Royo is written over the official stamp of the University of Zaragoza.

Fdo.: Javier Paricio Royo



Universidad
Zaragoza

**D.JUAN GARCÍA BLASCO, SECRETARIO GENERAL DE LA UNIVERSIDAD
DE ZARAGOZA,**

CERTIFICA:

Que, Don Dario Suárez Gracia con NIF: 18443694V, ha participado en la Convocatoria de Innovación Docente 2016-2017 de la Universidad de Zaragoza, dentro de la modalidad Programa de Incentivación de la Innovación Docente en la UZ (PIIDUZ) en calidad de coordinador principal del equipo del proyecto :

"Estudio y diseño de una plataforma común de trabajo para la mejora del aprendizaje en el Grado en Ingeniería Informática " con identificador PIIDUZ_16_270 .

Y para que conste, una vez finalizado y cumpliendo los requisitos marcados, se expide en Zaragoza, a 28 de agosto de 2017.

Universidad
Zaragoza

VºBº
El Vicerrector de Política Académica

Fdo.: Gerardo Sanz Saiz



D. JUAN FRANCISCO HERRERO PEREZAGUA, SECRETARIO GENERAL DE
LA UNIVERSIDAD DE ZARAGOZA,

C E R T I F I C A:

Que el profesor D. DARÍO SUÁREZ GRACIA con NIF 18443694V ha participado en el *Anillo Digital Docente* (<http://add.unizar.es>), campus virtual de la Universidad de Zaragoza, en el curso académico 2008-2009 con el curso:

"ARQUITECTURA DE COMPUTADORES (INGENIERÍA INFORMÁTICA)" (plataforma MOODLE)

Y para que conste, se expide la presente en Zaragoza, a 29 de julio de 2009.



Vº Bº
EL ADJUNTO AL RECTOR
PARA INNOVACIÓN DOCENTE

A blue ink signature of the name "Javier Paricio Royo".

Fdo.: Javier Paricio Royo



**D.JUAN FRANCISCO HERRERO PEREZAGUA, SECRETARIO GENERAL
DE LA UNIVERSIDAD DE ZARAGOZA,**

CERTIFICA:

Que, Don Darío Suárez Gracia con NIF: 18443694V, ha participado en el Anillo Digital Docente (<http://add.unizar.es>), campus virtual de la Universidad de Zaragoza, en el curso académico 2010-2011 con los cursos :

- "Arquitectura y Organización de Computadores 1" (plataforma Moodle).
- "Arquitectura de Computadores (Ingeniería Informática)" (plataforma Moodle).
- "Laboratorio de computadores" (plataforma Moodle).

Y para que conste, se expide este certificado en Zaragoza, a 15 de junio de 2011.

A handwritten signature in black ink, appearing to read "Juan Francisco Herrero".
The logo of the University of Zaragoza, featuring a stylized building icon above the text "Universidad Zaragoza" and the year "1542" below it.

Vººº
El Adjunto al Rector
para Innovación docente

Fdo.:Javier Paricio Royo

From: Kristin Blomquist kblomquist@nvidia.com 
Subject: RE: NVIDIA Hardware Request: Dario Suarez Gracia
Date: March 8, 2016 at 9:04 PM
To: dario@unizar.es
Cc: Chandra Chejji cchejji@nvidia.com, Kristin Blomquist kblomquist@nvidia.com

KB

Hi Dario,
We have reviewed your NVIDIA Hardware Grant Request and we are happy support your work with the donation of (1) Titan X to support your research. We will ship to the address listed i

Respond within 48 hours if you are unable to house the GPU or the address, email or phone number you have provided is incorrect. If you are unable to house the GPU or the shipment is u

you will receive a courtesy email notice from our shipping team with the tracking number.

When we ship we have to ship with a commercial invoice. The price on the commercial invoice is the donation value not the price you would purchase from a vendor at MSRP price. Since \

want at MSRP price. We also ship so we pay all fees if the country will allow it.

Please ensure that the GPU is powered properly as per the information found here: <http://developer.nvidia.com/hw-grant-supplemental-info>

If this hardware donation results in any publications or reports, we kindly ask that you acknowledge "NVIDIA Corporation". This is one way we track and justify our programs and adding "N

trust your judgment in how you acknowledge NVIDIA Corporation, however here are a couple samples: "The Tesla K40 used for this research was donated by the NVIDIA Corporation." or "

of the Tesla K40 GPU used for this research." You are also encouraged to acknowledge NVIDIA Corporation in any contacts with the news media or in general articles.

This equipment donation will come as an unrestricted gift to support your research, however it is not for resale. You may see an invoice in the shipping box. The price on the invoice is for :

reflect the actual price of the donated equipment. In addition, as part of the Hardware Donation Program, you will receive occasional newsletters and requests to complete surveys. While

the infrequent surveys. These are important for tracking and justifying our academic programs, and cover topics such as updates on your CUDA Courses and GPU-related research progress.

Kristin

Kristin Blomquist | Academic Programs Coordinator | NVIDIA Corporation | developer.nvidia.com/academics |



From: dario@unizar.es [mailto:dario@unizar.es]

Sent: Sunday, February 28, 2016 8:32 AM

To: UniversityPartnership

Subject: NVIDIA Hardware Request: Dario Suarez Gracia

A New Academic Hardware Request Form has been submitted.

First Name:	Dario
Last Name:	Suarez Gracia
Title:	Assistant Professor
University:	University of Zaragoza
Address Line 1:	ED. Ada Byron
Address Line 2:	C. Maria de Luna 1
Address Line 3:	
City:	Zaragoza
Region:	EMEA1 (Europe, Middle East, Africa, India)
Country:	Spain
State/Province:	Zaragoza
Other State:	
Zip/Postal Code:	50018
Phone Number:	003467655549
Email Address:	dario@unizar.es
Are you currently teaching CUDA course or planning to in the next year?:	Yes
Which libraries do you use or interested in using?:	cuBLAS Complete BLAS Library,cuRAND Random Number Generation (RNG) Library,NPP Perform
Research Domain/Field of Interest:	Computer Vision & Machine Vision,Development Tools & Libraries,Embedded,Mobile,Programm
Programming Interfaces/Languages solutions used for GPU acceleration:	CUDA,C,CUDA C++,C++ AMP,OpenACC,OpenCL
Equipment Requested:	Titan X
Do you have equipment to house the donated GPUs?:	Yes
Have you submitted an NVIDIA Hardware Request in the last 12 months?:	No
Has anyone from your group submitted an NVIDIA Hardware request for the same project in the last 12 months?:	No
Is this for a conference sponsorship?:	Yes

This email message is for the sole use of the intended recipient(s) and may contain confidential information. Any unauthorized review, use, disclosure or distribution is prohibited. If you are not the intended recipient, please contact the sender by reply email and destroy all co

From: University university@altera.com
Subject: Altera University Program Enrollment Request Ticket [ER4441]: Approved
Date: 10 October 2016 at 04:06
To: dario@unizar.es



Dear Prof. Dario Suarez Gracia,

Your application to join the Altera University Program has been approved. For future reference, your Altera ID is 1074187.

The Altera University Program provides hardware, software, and teaching materials to help introduce students to digital technology. Hardware support is in the form of Development and Education boards (DE0, DE1-SoC, DE4, and DE5), specially designed for use in teaching and research laboratories. Software support consists of the Quartus Computer Aided-Design software, Nios II soft processor, and simulation tools. Teaching materials comprise tutorials and ready-to-teach laboratory exercises for use in digital logic and computer organization courses.

The University Program recommends Quartus Lite Edition for academic purposes. This software can be downloaded free of charge from the [Altera Download Center](#) and does not require a license.

For further information please visit the [Altera University Program website](#).

Best Regards,

Altera University Program

Confidentiality Notice.
This message may contain information that is confidential or otherwise protected from disclosure. If you are not the intended recipient, you are hereby notified that any use, disclosure, dissemination, distribution, or copying of this message, or any attachments, is strictly prohibited. If you have received this message in error, please advise the sender by reply e-mail, and delete the message and any attachments. Thank you.



Instituto de
Ciencias de la Educación
Universidad Zaragoza



Copia auténtica de documento firmado digitalmente. Puede verificar su autenticidad en <http://verifica.unizar.es/csv/5589752462027cbf603d23ed50dab0e>

Dña Ma Isabel Ubieto Artur, Profesora Secretaria del Instituto de Ciencias de la Educación de la Universidad de Zaragoza,

CERTIFICA

Que D. Darío SUÁREZ GRACIA, con DNI nº 18443694-V, según consta en la Secretaría de este Instituto, participó como Tutor en el Centro Politécnico Superior de Zaragoza, con una dedicación de 40 horas, dentro de las actividades del Programa Tutor del curso académico 2009-2010.

Y para que así conste a los efectos oportunos, se expide la presente certificación duplicada del original, en Zaragoza, a veinticinco de febrero de dos mil veinte.

Profesora Secretaria del Instituto de Ciencias de la Educación
Fdo. M.º Isabel Ubieto Artur
(Firmado electrónicamente y con autenticidad contrastable según el art.27.3 c) de la Ley 39/2015)

unizar.es

CSV: 5589752462027cbf603d23ed50dab0e	Organismo: Universidad de Zaragoza	Página: 1 / 1
Firmado electrónicamente por	Cargo o Rol	Fecha
MARIA ISABEL UBIETO ARTUR	Profesora Secretaria ICE	25/02/2020 12:32:00





Instituto de
Ciencias de la Educación
Universidad Zaragoza



Copia auténtica de documento firmado digitalmente. Puede verificar su autenticidad en <http://ice.unizar.es/csv/ffe691be56f8b5e80312bd1a2976e005>

Dña Ma Isabel Ubieto Artur, Profesora Secretaria del Instituto de Ciencias de la Educación de la Universidad de Zaragoza,

CERTIFICA

Que D. Darío SUÁREZ GRACIA, con DNI nº 18443694-V, según consta en la Secretaría de este Instituto, participó como Tutor en la Escuela de Ingeniería y Arquitectura, con una dedicación de 40 horas, dentro de las actividades del Programa Tutor del curso académico 2010-2011.

Y para que así conste a los efectos oportunos, se expide la presente certificación duplicada del original, en Zaragoza, a veinticinco de febrero de dos mil veinte.

Profesora Secretaria del Instituto de Ciencias de la Educación
Fdo. M.º Isabel Ubieto Artur
(Firmado electrónicamente y con autenticidad contrastable según el art.27.3 c) de la Ley 39/2015)

unizar.es

CSV: ffe691be56f8b5e80312bd1a2976e005	Organismo: Universidad de Zaragoza	Página: 1 / 1
Firmado electrónicamente por	Cargo o Rol	Fecha
MARIA ISABEL UBIETO ARTUR	Profesora Secretaria ICE	25/02/2020 12:32:00



Dña M^a Isabel Ubieto Artur, Profesora Secretaria del Instituto de Ciencias de la Educación de la Universidad de Zaragoza,

CERTIFICA

Que D. Darío SUÁREZ GRACIA,
con DNI nº 18443694V, según consta en la Secretaría de este Instituto, participó como Tutor del *Plan de Orientación Universitaria de la Universidad de Zaragoza (POUZ)* en el curso 1 del Grado en Ingeniería Informática de la Escuela de Ingeniería y Arquitectura durante el curso académico 2016-2017, con una dedicación de 50 horas.

Y para que así conste a los efectos oportunos, se expide la presente certificación duplicada del original, en Zaragoza, a veinticinco de febrero de dos mil veinte.

Profesora Secretaria del Instituto de Ciencias de la Educación
Fdo. M^a Isabel Ubieto Artur
(Firmado electrónicamente y con autenticidad contrastable según el art.27.3 c) de la Ley 39/2015)



Copia auténtica de documento firmado digitalmente. Puede verificar su autenticidad en <http://www.unizar.es/csv/f4697d5538734963ce6cc1b8629f0a0c>

CSV: f4697d5538734963ce6cc1b8629f0a0c	Organismo: Universidad de Zaragoza	Página: 1 / 1
Firmado electrónicamente por	Cargo o Rol	Fecha
MARIA ISABEL UBIETO ARTUR	Profesora Secretaria ICE	25/02/2020 12:31:00





Instituto de Ciencias de la Educación
UNIVERSIDAD DE ZARAGOZA

FERNANDO BLANCO LORENTE, Profesor y Secretario en funciones del Instituto de Ciencias de la Educación de la Universidad de Zaragoza,

CERTIFICA:

Que D. Dario SUAREZ GRACIA con D.N.I. nº 18443694V, ha participado en la actividad "Otras formas de trabajar en el aula: metodologías activas y colaborativas", dentro de las actividades dirigidas al profesorado universitario, celebrado en Zaragoza, 5 y 6 de febrero de 2009, con una duración de 8 horas.

Y para que así conste a los efectos oportunos, se expide la presente certificación en Zaragoza, a diez de febrero de dos mil nueve.

Vº Bº
EL DIRECTOR
EN FUNCIONES

Ramón Garcés Campos



LA FUNCIONARIA

R. Cebollada

Rosa Cebollada Langa

CERTIFICACIÓN DEL PROGRAMA

- 1.- El contexto social e institucional de los procesos de enseñanza y aprendizaje.
- 2.- La práctica pedagógica y la importancia de la metodología.
- 3.- La necesidad de una práctica pedagógica diferente.
- 4.- Diferentes escenarios metodológicos para fomentar la actividad y la colaboración en las aulas universitarias.
- 5.- El papel de los docentes y de los estudiantes.



Instituto de Ciencias de la Educación
UNIVERSIDAD DE ZARAGOZA

FERNANDO BLANCO LORENTE, Profesor y Secretario en funciones del Instituto de Ciencias de la Educación de la Universidad de Zaragoza,

CERTIFICA:

Que D. Darío SUAREZ GRACIA con D.N.I. nº 18443694V, ha participado en la actividad "Gestión eficaz del tiempo", dentro de las actividades dirigidas al profesorado universitario, celebrado en Zaragoza, del 30 de marzo al 7 de abril de 2009, con una duración de 12 horas.

Y para que así conste a los efectos oportunos, se expide la presente certificación en Zaragoza, a veinte de abril de dos mil nueve.

Vº Bº
EL DIRECTOR
EN FUNCIONES

Ramón Garcés Campos



LA FUNCIONARIA

Mº Dolores Ferrera Broncano

CERTIFICACIÓN DEL PROGRAMA

- 1.- Los diferentes niveles de organización del trabajo.
- 2.- La planificación del tiempo. (Herramientas de planificación)
- 3.- Diseño de un programa de tiempo. Establecimiento de objetivos de optimización del tiempo y mejora de resultados profesionales.
- 4.- La administración del tiempo (los ladrones del tiempo).
- 5.- Búsqueda de un estilo personal eficiente.
- 6.- Técnicas de autoevaluación de la gestión del tiempo y organización del trabajo.
- 7.- Técnicas de concentración mental y su contribución a la mejora.
- 8.- Ejercicios prácticos de aplicación de los conceptos tratados en los apartados anteriores.



Instituto de Ciencias de la Educación
UNIVERSIDAD DE ZARAGOZA

FERNANDO BLANCO LORENTE. Profesor y Secretario en funciones del Instituto de Ciencias de la Educación de la Universidad de Zaragoza,

CERTIFICA:

Que **D. Darío SUAREZ GRACIA**, con D.N.I. nº 18443694V, ha participado en la actividad "Innovación en la docencia: cómo mejorar las sesiones expositivas", dentro de las actividades dirigidas al profesorado universitario, celebrado en Zaragoza, 7 de mayo de 2009, con una duración de 4 horas.

Y para que así conste a los efectos oportunos, se expide la presente certificación en Zaragoza, a ocho de mayo de dos mil nueve.

Vº Bº
EL DIRECTOR
EN FUNCIONES

Ramón Garcés Campos



LA FUNCIONARIA

Ana Gómez Opla

CERTIFICACIÓN DEL PROGRAMA

- 1.- ¿Profesor/a o profesores? Componentes del docente universitario.
- 2.- Más allá de la presentación.
- 3.- Aspectos fundamentales en la docencia.
- 4.- La comunicación con el alumnado. Elementos a tener en cuenta en la sesión expositiva.
- 5.- Cómo mejorar la sesión magistral. Estrategias docentes para grupos grandes.
- 6.- Adecuación de las estrategias docentes a los contenidos y a los objetivos de la docencia.



INSTITUTO DE CIENCIAS DE LA EDUCACIÓN
UNIVERSIDAD DE ZARAGOZA

CONCEPCIÓN BUENO GARCÍA, Secretaria en funciones
del Instituto de Ciencias de la Educación de la Universidad de
Zaragoza,

CERTIFICA:

Que D. Darío SUAREZ GRACIA
con D.N.I. nº 18443694V, ha participado en la actividad
"Enquiry based learning in engineering", dentro de las
actividades dirigidas al profesorado universitario,
celebrado en Zaragoza, 1 de febrero de 2010, con una
duración de 4 horas.

Y para que así conste a los efectos oportunos, se expide
la presente certificación en Zaragoza, a dos de febrero de dos
mil diez.

Vº Bº
EL SUBDIRECTOR
EN FUNCIONES

Pedro Allueva Torres



LA FUNCIONARIA

Rosa Cebollada Langa

CERTIFICACIÓN DEL PROGRAMA

- 1.- What is EBL?
- 2.- The benefits of EBL for students and teacher.
- 3.- How does it work?
- 4.- How do you know it is working?
- 5.- Some examples of EBL in engineering - what does it look like in practice?
- 6.- Computing database and website design and programming.



INSTITUTO DE CIENCIAS DE LA EDUCACIÓN

UNIVERSIDAD DE ZARAGOZA

CONCEPCIÓN BUENO GARCÍA, Secretaria en funciones
del Instituto de Ciencias de la Educación de la Universidad de
Zaragoza,

CERTIFICA:

Que **D. Darío SUAREZ GRACIA**
con D.N.I. nº 18443694V, ha participado en la actividad
"Guías docentes: contenido y elaboración", dentro de
las actividades dirigidas al profesorado universitario,
celebrado en Zaragoza, del 11 de enero al 11 de marzo
de 2010, con una duración de 20 horas.

Y para que así conste a los efectos oportunos, se expide
la presente certificación en Zaragoza, a doce de abril de dos mil
diez.

Vº Bº
EL SUBDIRECTOR
EN FUNCIONES

Pedro Allueva Torres



LA FUNCIONARIA

R. Cebollada

Rosa Cebollada Langa

CERTIFICACIÓN DEL PROGRAMA

- 1.- Referencias clave para el diseño de las guías docente.
- 2.- Elementos que constituyen la Guía Docente de una materia.
- 3.- Propuestas para el proceso de elaboración.



INSTITUTO DE CIENCIAS DE LA EDUCACIÓN
UNIVERSIDAD DE ZARAGOZA

CONCEPCIÓN BUENO GARCÍA, Secretaria en funciones
del Instituto de Ciencias de la Educación de la Universidad de
Zaragoza,

CERTIFICA:

Que D. Darío SUAREZ GRACIA
con D.N.I. nº 18443694V, ha participado en la actividad
"El portafolio, una herramienta para la evaluación
auténtica", dentro de las actividades dirigidas al
profesorado universitario, celebrado en Zaragoza, del 26
al 28 de mayo de 2010, con una duración de 8 horas.

Y para que así conste a los efectos oportunos, se expide
la presente certificación en Zaragoza, a veintisiete de mayo de
dos mil diez.

Vº Bº
EL SUBDIRECTOR
EN FUNCIONES

Pedro Allueva Torres

LA FUNCIONARIA

Rosa Cebollada Langa

CERTIFICACIÓN DEL PROGRAMA

- 1.- ¿Qué es el portafolio de evaluación?
- 2.- ¿Por qué nos interesa el portafolio de evaluación?
- 3.- ¿Cómo se hace el portafolio de evaluación?



Instituto de
Ciencias de la Educación
UniversidadZaragoza

CONCEPCIÓN BUENO GARCÍA, Profesora Secretaria del
Instituto de Ciencias de la Educación de la Universidad de
Zaragoza,

CERTIFICA:

Que **D. Darío SUAREZ GRACIA**
con D.N.I. nº 18443694V, ha participado en la actividad
"Cómo introducir y evaluar la competencia específica:
aplicar el método científico en los laboratorios de
ciencias y tecnología", dentro de las actividades
dirigidas al profesorado universitario, celebrado en
Zaragoza, 27 y 28 de enero de 2011, con una duración
de 6 horas.

Y para que así conste a los efectos oportunos, se expide
la presente certificación en Zaragoza, a uno de febrero de dos
mil once.

Vº Bº
EL DIRECTOR

Pedro Allueva Torres



Instituto de
Ciencias de la Educación
UniversidadZaragoza

EL FUNCIONARIO

José Carlos Tienda Trillo

unizar.es

CERTIFICACIÓN DEL PROGRAMA

- 1.- La "Guía para la evaluación de competencias en los laboratorios en el ámbito de ciencias y tecnología"
- 2.- La Competencia específica "Aplicar el método científico para la resolución de problemas en los laboratorios de ciencias y tecnología": Componentes competenciales, elementos y niveles.
- 3.- Cambios metodológicos: Pre-lab y post-lab.
- 4.- Diseño de ficha de la asignatura y de las actividades.
- 5.- Rúbricas de evaluación para los diferentes componentes y niveles.
- 6.- Ejemplos de actividades.



**Instituto de
Ciencias de la Educación
Universidad Zaragoza**

MARÍA ISABEL UBIETO ARTUR, Profesora Secretaria del Instituto de Ciencias de la Educación de la Universidad de Zaragoza.

CERTIFICA:

Que **D. Dario Suárez Gracia**, con DNI nº 18443694V ha participado en la actividad “POUZ. Integración de los estudiantes en la Universidad. Binomio Tutor-Mentor”, dentro de las actividades dirigidas al profesorado universitario, celebrado en Zaragoza, el 4 de octubre de 2016, con una duración de 2 horas.

Y para que así conste a los efectos oportunos, se expide la presente certificación en Zaragoza, a veinte de octubre de dos mil dieciséis.

Vº Bº
LA DIRECTORA

Ana Rosa Abadía Valle

**Instituto de
Ciencias de la Educación
Universidad Zaragoza**

LA FUNCIONARIA

Pilar Vicente Alcutén

unizar.es



**Instituto de
Ciencias de la Educación
Universidad Zaragoza**

MARÍA ISABEL UBIETO ARTUR, Profesora Secretaria del Instituto de Ciencias de la Educación de la Universidad de Zaragoza.

CERTIFICA:

Que **D. Dario Suárez Gracia**, con DNI nº 18443694V ha participado en la actividad “POUZ. Características e implementación”, dentro de las actividades dirigidas al profesorado universitario, celebrado en Zaragoza, el 4 de octubre de 2016, con una duración de 2 horas.

Y para que así conste a los efectos oportunos, se expide la presente certificación en Zaragoza, a veinte de octubre de dos mil dieciséis.

Vº Bº
LA DIRECTORA

Ana Rosa Abadía Valle



**Instituto de
Ciencias de la Educación
Universidad Zaragoza**

LA FUNCIONARIA

Pilar Vicente Alcutén

unizar.es