OS-Level Task-Based Mechanism for Lightweight Manycore Processors

João Vicente Souto
UFSC
Florianópolis, Brazil
joao.vicente.souto@grad.ufsc.br

Pedro Henrique Penna
UGA, PUC Minas
Grenbole, France
pedro.penna@sga.pucminas.br

Márcio Castro
UFSC
Florianópolis, Brazil
marcio.castro@ufsc.br

***Abstract*—Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.**

***Index Terms*—HAL, Distributed Operating System, Lightweight Manycore, Kalray MPPA-256**

I. Introduction

A classe de processadores Lightweight Manycore destaca-se pela seu alto nível de paralelismo com baíxo consumo energético [1]. As aplicações projetadas para executar nesses processadores tem a sua disposição centenas de núcleos agrupados e distribuídos em um único chip. Para endereçar a escalabilidade do sistema e prover a eficiência energética, LWs exibem características arquiteturais próprias que os diferem dos demais Manycores. Especificamente, eles integram: (i) thousands of low-power cores with Multiple Instruction Multiple Data (MIMD) capability [2]; (ii) distributed memory architecture and small local memories shared by tightly-coupled groups of cores (aka *clusters*) [3]; (iii) reliable and fast Network-on-Chips (NoCs) for message-passing [3]; and (iv) heterogeneous processing capabilities [4]. Alguns exemplos de LWs são A, B e C.

Tais características introduzem diversos desafios no desenvolvimento de aplicações de baíxo e alto nível. Por exemplo, (i) programação híbrida entre os modelos de memória compartilhada e troca de mensagens devido a natureza dos LWs [5]; (ii) falta de suporte em hardware para coerência de cache [1]; (iii) sistema de memória distribuído e restritivo [6]; e (iv) programação de componentes heterogêneos [7]. Neste contexto, diversos trabalhos propõem soluções em diversos níveis de abstração para amenizar as dificuldades e prover ao usuário final alternativas para o desenvolvimento dessas arquiteturas. Deste runtimes especializados em um tipo de abstração a ambientes de programação mais robustos e genéricos.

Dentre todas as soluções, uma das principais dificuldades é como lidar com a quantidade reduzida de memória local dentro de um cluster. Vale ressaltar que essa memória muitas vezes não é exclusiva para o usuário. A memória local precisa armazenar códigos e estruturas de dados da aplicação e de suas dependências, e.g., OS e/ou bibliotecas. Algo que consome uma quantidade considerável de memória é o suporte a múltiplos fluxos de programação. Por exemplo, alocação de páginas de memória para armazenar o contexto e pilha de execução de uma rotina. Neste contexto, o presente trabalho propõe uma abstração de task em nível de kernel para definir uma unidade genérica de execução. Similar a uma piscina de threads, uma thread do sistema aguarda receber essas unidades para executar. Com este mecanismo, buscamos atingir os seguintes objetivos e contribuíções:

- Projetar um mecanismo de task no nível do kernel para uso interno do OS e para a aplicação cliente.
- Diminuir a alocação de memória do sistema de threads ao reutilizar o contexto e pilha de execução de uma única thread;
- Melhorar a utilização de núcleos ociosos;
- Explorar a localidade dos dados na cache de um único núcleo;
- Habilitar operações assíncrona e/ou periódicas, e.g., envio assíncrono em LWs que não possuem DMA.
- Facilitar a modelagem de funcionalidades internas do kernel.

Este trabalho faz parte do desenvolvimento do Nanvix, um sistema operacional distribuído projetado para endereçar as características dos LWs. O escopo do trabalho se limita ao aperfeiçoamento do um microkernel assimétrico que gerencia os recursos locais de um cluster.

The remainder of this work is organized as follows. In Section II, we discuss related work. In Section III we cover the problem definition. In Section IV, we present our proposal. In Section V, we detail the experimental environment. In Section VI, we discuss our experimental results. In Section II we discuss related works. In Section VII, we draw our conclusions.

We thank CNRS, CNPq, FAPESC, FAPEMIG for supporting this research. This study was financed in part by the Coordenação de Aperfeicoamento de Pessoal de Nıvel Superior - Brasil (CAPES) - Finance Code 001.

II. Related Work

Existem diversas soluções propostas que visam tirar proveito do alto nível paralelismo e baixo consumo energético dos LWs sem introduzir um overhead indesejado ao sistema.

Soluções específicas, que modelam todo o ambiente de execução entorno de um tipo de um paradigma de programação buscam eliminar problemas de concorrência existentes em sistemas multi-processador e preemptivos. Por exemplo, ambientes de execução baseados em tarefas ou corotinas exploram o paralelismo de tarefas que trabalham de forma cooperativa. Esses ambientes oferecem suporte a flexibilidade e um controle de grão fino das tarefas. É possível também mapear pradões de programação paralela, como OPENMP diretamente em conjuntos de tarefas/corotinas nesses sistemas. A cooperação entre as unidades de execução é um importante ponto de benefício desses sistemas, onde a troca de contexto é leve e não ocorrem involuntáriamente, tornando a execução mais preditiva. Entretanto, esses runtimes limitam a classe de aplicações que podem ser portadas para LWs.

Outras soluções, exploram desenvolver padrões e mecanismos no nível do kernel modeladas especificamente para LWs sem perder a compatibilidade com outros sistemas. Entretanto, alguns serviços do SO podem a vir requerer uma quantidade um footprint de memória excessivo, forçando a aplicação cliente a trabalhar com uma quantidade limitada. Nós acreditamos que muitos desses serviços internos do SO podem ser modelados sobre uma perspectiva de tarefas solicitadas, periódicas e/ou assíncronas. Está solução é especialmente vantajosa em microkernel assímetricos por poderem utilizar núcleos ociosos para executar tais tarefas.

III. Problem Definition

O gerenciamento de memória interna de um cluster afeta todos os níveis de abstração no contexto dos LWs. Por exemplos, OSs precisam ser pequenos e leves para deixar a maior quantidade de memória disponível pra aplicação. Por outro lado, aplicações extremamente paralelas e concorrentes têm a necessidade de gerenciar manualmente a coerência dos dados manipulados.

Para amenizar os problemas de coerência de cache existente nessas arquiteturas, Penna ETAL propôs um microkernel assímetrico para concentrar o gerenciamento e manipulação das estruturas do OS em um único núcleo. A Figura X ilustra a execução de uma chamada de sistema dentro do Nanvix. Quando um núcleo escravo deseja realizar uma operação privilegiada, este envia uma solicitação ao núcleo mestre, que por sua vez, realiza e retorna o resultado ao solicitante. Os problemas explorados neste trabalho estão contidos no escopo deste nível de OS.

Além das dificuldades impostas pelo sistema de memória, nossa proposta busca atacar algumas das desvantagens do microkernel e do sistema de comunicação. As descrições a seguir sumarizam os principais problemas que nossa solução se propõe endereçar:

Memory Utilizationg Para cada novo fluxo de execução (thread), o sistema de memória deve reservar dois espaços de memória, um para a pilha de execução do usuário e outra reservada para o kernel. Geralmente, cada pilha possúi o tamanho de uma página de memória, crescendo rapidamente a quantidade de memória necessária para criar novas threads no sistema.

Data Localityg Por causa da falta de suporte em hardware para coerência de cache, a disputa por uma região de memória compartilhada é custosa. Para entrar em uma região crítica, as threads precisam garantir que os endereços na cache estejam invalidados, forçando o acesso ao banco de memória local. Ao sair de uma região crítica, as threads precisam forçar a escrita dos dados modificados para que os outros possam ver a atualização. Por isso, o isolamento dos dados em um único núcleo para explorar a localidade dos dados na cache é tão importante.

Core Utilizationg Pela necessidade de ter um núcleo reservado para a thread de sistema, o microkernel assimétrico perde poder de processamento ao deixar o núcleo mestre ocioso entre solicitações de syscalls.

Asynchronous Operationsg Pela simplificidade e redução de energia, LWs podem não possuir uma DMA dedicada para executar comunicação assíncrona. Deste modo, é responsabilidade da thread realizar pooling dos dados na NoC manualmente. Todas as chamadas de sistemas dentro do microkernel também bloqueiam as threads, mesmo que a mesma pudesse realizar operações independentes enquanto uma syscall, que não é critica, é realizada.

Periodic Operationsg Para permitir a uma cluster seja monitorado ou receba comandos externos, é necessário que uma thread exista para solicitando verificações e aguardando mensagens externa, aumentando a necessidade de memória do OS. Entretanto, a existência dessas operações são essenciais para desenvolver serviços mais complexos, e.g., gerenciamento e migração de processos, invalidação de memória compartilhada e distribuída, e execução de procedimentos remotos.

IV. OS-Level Task-Based Mechanism for Lightweight Manycore Processors

A abstração de Tarefa proposta podem ser vista como um caso especial de corotinas que encapsulam uma subrotina e pode ser executada independemente de quem as criou. Entretanto, diferentemente das corotinas, que possuem sua própria pilha de execução e contexto, a definição de tarefas proposta é desaclopada da necessidade de ter uma thread dedicada por tarefa. Neste ponto, introduzimos uma thread especial, nomeada de Dispatcher, que é um executor genérico de tarefas. Seguindo o modelo Produtor-Consumidor, o Dispatcher consome tarefas de uma fila global de tarefas, onde solicitantes inserem atomicamente tarefas a serem realizadas.

Uma Tarefa é uma estrutura padronizada que guarda informações semânticas e de controle. De um lado, as variáveis semânticas armazenam a função a ser executada, seus argumentos, e um slot para guardar o retorno, caso desejado. Por outro lado, variáveis de controle, referentes a semântica

do Dispatcher, são compostas pelo estado da tarefa, uma lista de tarefas dependêntes, quantidade de tarefas pais ativas, e um controle de sincronização com o solicitante. Os grafos de dependência permitem introduzir comportamentos e gerenciamentos mais sofisticados assíncronos, onde uma tarefa só estará pronta para executar quando todas as tarefas pais tiverem concluído suas execuções.

O Pesudocódigo A sumariza o comportamento de um Dispatcher. Bloqueado em um semáforo, o Dispatcher aguardará novas solicitações de tarefas surgirem. Ao consumir uma tarefa, o Dispatcher atualiza seu estado e entra no escopo da tarefa. Neste ponto é possível notar a reutilização das pilhas de um Dispatcher por inúmeras tarefas independentes, pois quando o mesmo retornar do escopo da tarefa, o mesmo estará em seu estado inicial e apto a iniciar outra tarefa. A tarefa pode ainda sinalizar entre três valores, o estado de conclusão ao Dispatcher. Especificamente:

TASK_RET_SUCCESS O Dispatcher concluirá a tarefa, sinalizando sua conclusão as tarefas dependentes e liberando o solicitante.

TASK_RET_AGAIN A tarefa retornou com um erro mas é recuperável e a tarefa será reescalonada.

TASK_RET_STOP A tarefa retornou com um erro, mas apesar de ser recuperável, ela precisa aguardar outra operação ser concluída.

TASK_RET_ERROR A tarefa retornou um erro e é irrecuperável, o Dispatcher sinalizará o erro ao solicitante e propagará o erro para todas as tarefas dependêntes.

Para a implementação de novos Dispatchers para aumentar o paralelismo e diminuir o tempo de espera para execução de uma tarefa, basta criar novas threads que executem a função proposta que ela já suporta por definição multiplos Dispatchers.

A proposta do mecanismo baseado em tarefas no nível do OS endereça os problemas descritos na seção anterior da seguinte forma:

Memory Utilization A definição de múltiplos fluxos de execução em tarefas isoladas, ou em um grafo de dependência, reduz a utilização de páginas de memória pra threads baseado na quantidade de Dispatchers existentes. Se o kernel e a aplicação conseguirem isolar comportamentos simples sem a necessidade da criação de uma thread dedicada, mais memória estará disponível para armazenar dados úteis.

Data Locality Configurando um Dispatcher a ficar sempre em apenas um núcleo, o mesmo poderá executar tarefas que compartilham as mesma estruturas de dados explorando a localidade dos dados.

Core Utilization Neste ponto entra a importância de se definir o Dispatcher em nível do OS para que seja possível movê-lo para o núcleo mestre e ele compartilhe o tempo de execução com a thread mestre. Em um extremo, poderíamos até definir a própria thread que realiza chamadas de sistema em um Dispatcher.

Asynchronous Operations É possível modelar tarefas para introduzir essa noção de envio assíncrono, deixando a responsabilidade de enviar os dados manualmente para o Dispatcher enquanto as threads continuam sua execução normalmente. A mesma ideia se aplica para as syscalls, onde operações que não são críticas podem ser realizadas de forma assíncrona, liberando as threads solicitantes.

Periodic Operations É possível modelar tarefas que possuam um período ao qual elas são desbloqueadas e executam. Desta forma, eliminamos a necessidade de ter uma thread dedicada que fica esperando alguma condição para relizar operações específicas. Por exemplo, a leitura de uma comunicação da NoC que criará uma nova tarefa para execução de um procedimento remoto.

A Figura A exemplifica interações dentro do Nanvix após a implementação do mecanismo de tarefas. Tarefas pediódicas e criadas pelo próprio kernel permitiram uma interação mais rica intra e inter-clusters. Como a thread mestre responsável por atender uma solicitação de cada vez, mover tal responsabilidade pro Dispatcher não introduzirá um gargalo no sistema além do que já existe. Por fim, tarefas solicitadas por usuários comuns podem e devem se intercalar entre si, o que nos leva a necessidade de um gerenciamento mais fino do que a ordem de inserção numa fila.

```
while !shutdown do
    Waits for a task;
    Consume from global task queue;
    Execute task function;
    switch ret do
        case TASK_RET_SUCCESS do
            Complete the task and schedule children;
        end
        case TASK_RET_AGAIN do
            Reschedule the task;
        end
        case TASK_RET_STOP do
            Insert the task into a waiting queue;
        end
        case TASK_RET_ERROR do
            Propagate the error and release all tasks;
        end
    end
end
```

Algorithm 1: How to write algorithms

V. Experimental Environment

O MPPA é um LW processor desenvolvido pela empresa francesa Kalray. Ele é uma das arquiteturas suportadas pelo Nanvix e será utilizado como caso de estudo neste trabalho. Conforme a Figura X ilustra, o MPPA integra 288 núcleos de propósito geral, agrupados em 16 clusters de computação, destinados a computação útil, e 4 cluster de IO responsáveis pela comunicação com periféricos. De um lado, cada CC integra 16 PEs, 1 RM, 2 MB de SRAM local, duas interfaces NoC e não suportam coerência de cache pelo hardware. Por outro lado, cada IO intergra 4 RMs, 4 MB de SRAM e 8 intefaces

TABLE I
BENCHMARK PARAMETERS FOR EXPERIMENTS.

Benchmark	#Tasks	#Threads	Memory
Single Dispatcher	1 to 29	1	$1 \times 8\ KB = 8\ KB$
Multiple Dispatchers	1 to 29	14	$14 \times 8\ KB = 112\ KB$
Threads	1 to 29	1 to 29	$[1, 29] \times 8\ KB = [8, 232]\ KB$

NoC. Dois desses IO são conectados a um controlador DDR com acesso a 4 GB de DRAM, e os outros dois são conectos a controladores Ethernet. O MPPA também possui duas NoC distintas em uma topologia 2-D Torus interpolada, a CNoC e a DNoC. A CNOC é destinada a trocas de pequenas mensagens de controle e sincronização e a DNOC é destinada a troca de quantidades arbitrárias de dados.

Figura 1.

VI. RESULTS

Para realizar uma avaliação ponderada dos custos de criar, executar e esperar uma tarefa, criamos um benchmark sintético que realiza a escrita nos endereços de uma página de memória, e.g., 4 KB. Para garantir 95% de confiança, foram realizadas 50 replicações, descartando-se as primeiras 10 para ignorar o período de aquecimento.

Figura I.

A Tabela X exibe a aplicação deste benchmark replicado em três cenários diferentes: (i) Um único dispatcher atende todas as tarefas sequencialmente; (ii) Cada núcleo escravo que esteja livre contém um dispatcher diferente atendendo tarefas paralelamente, resultando em 14 dispatchers; (iii) Uma thread para cada tarefa, variando a quantidade de threads entre 1 à 29 threads. A quantidade de Tarefas foi limitada a quantidade possível de threads simultâneas dentro do microkernel para que não fosse necessário replicar o comportamento de tarefas nas threads de usuário. É possível verificar também, além de exibir a quantidade the tarefas e threads envolvidas, a quantidade de memória necessária para execução de cada cenário. Com base nisso, podemos perceber que muito qualquer esforço extra, o usuário pode utilizar a abstração de tarefas com dispatchers e

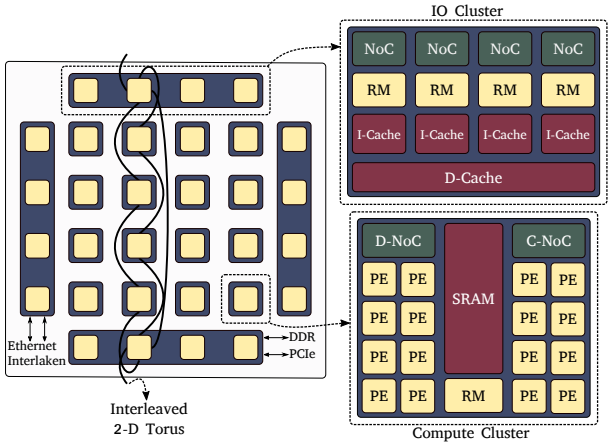

Fig. 1. Kalray MPPA-256 Architecture Overview.

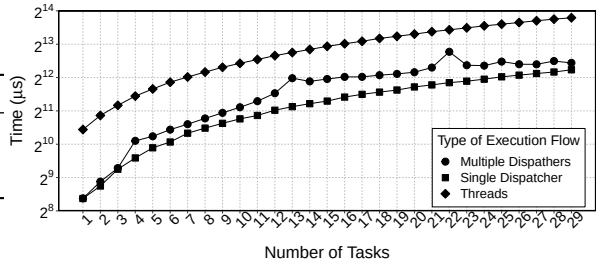

Fig. 2. Runtime Results.

manter a quantidade de memória utilizada limitada. Especialmente, o cenário com um único dispatcher é a situação ideal para que o mesmo possa a proveitar o tempo ocioso do núcleo mestre mas este limita.

A Figura B mostra os tempos de execução de cada cenário. Os tempos coletados são compostos do periodo de Dispatch/Create e Wait/Join de uma Tarefa/Thread.

Figura 2.

VII. CONCLUSIONS

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

trabalhos futuros

Aplicação de algoritmos de escalonamento e ordenação de tarefas, dependência de grafos, etc.

REFERENCES

[1] E. Francesquini, M. Castro, P. H. Penna, F. Dupros, H. Freitas, P. Navaux, and J.-F. Méhaut, "On the Energy Efficiency and Performance of Irregular Application Executions on Multicore, NUMA and Manycore Platforms," *Journal of Parallel and Distributed Computing*, vol. 76, no. C, pp. 32–48, Feb. 2015.

[2] D. Rossi, A. Pullini, I. Loi, M. Gautschi, F. K. Gurkaynak, A. Teman, J. Constantin, A. Burg, I. Miro-Panades, E. Beigne, F. Clermidy, P. Flatresse, and L. Benini, "Energy-efficient near-threshold parallel computing: The pulpv2 cluster," *IEEE Micro*, vol. 37, no. 5, pp. 20–31, sep 2017.

[3] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, B. Liu, A. Tran, E. Adeagbo, and B. Baas, "KiloCore: A 32-nm 1000-Processor Computational Array," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 4, pp. 891–902, 2017.

[4] S. Davidson, S. Xie, C. Torng, K. Al-Hawai, A. Rovinski, T. Ajayi, L. Vega, C. Zhao, R. Zhao, S. Dai, A. Amarnath, B. Veluri, P. Gao, A. Rao, G. Liu, R. K. Gupta, Z. Zhang, R. Dreslinski, C. Batten, and M. B. Taylor, "The Celerity Open-Source 511-Core RISC-V Tiered Accelerator Fabric: Fast Architectures and Design Methodologies for Fast Chips," *IEEE Micro*, vol. 38, no. 2, pp. 30–41, mar 2018.

[5] B. Kelly, W. Gardner, and S. Kyo, "AutoPilot: Message Passing Parallel Programming for a Cache Incoherent Embedded Manycore Processor," in *Proceedings of the 1st International Workshop on Many-core Embedded Systems*, ser. MES '13. Tel-Aviv, Israel: ACM, Jun. 2013, pp. 62–65. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2489068.2491624

[6] M. Castro, E. Francesquini, F. Dupros, H. Aochi, P. O. Navaux, and J.-F. Méhaut, "Seismic wave propagation simulations on low-power and performance-centric manycores," *Parallel Computing*, vol. 54, pp. 108–120, 2016.

[7] A. Barbalace, M. Sadini, S. Ansary, C. Jelesnianski, A. Ravichandran, C. Kendir, A. Murray, and B. Ravindran, "Popcorn: Bridging the Programmability Gap in Heterogeneous-ISA Platforms," in *European Conf. on Computer Systems*, Bordeaux, France, Apr. 2015, pp. 1–16.