Architecture Support for Domain-Specific Accelerator-Rich CMPs

JASON CONG, MOHAMMAD ALI GHODRAT, MICHAEL GILL, BEAYNA GRIGORIAN, and GLENN REINMAN, University of California, Los Angeles

This work discusses hardware architectural support for domain-specific accelerator-rich CMPs. First, we present a hardware resource management scheme for sharing of loosely coupled accelerators and arbitration of multiple requesting cores. Second, we present a mechanism for accelerator virtualization. This allows multiple accelerators to efficiently compose a larger virtual accelerator out of multiple smaller accelerators, as well as to collaborate as multiple copies of a simple accelerator. All of this work is supported by a fully automated simulation tool-chain for both accelerator generation and management. We present the applicability of our approach to four different application domains: medical imaging, commercial, computer vision, and navigation. Our results demonstrate large performance improvements and energy savings over a software implementation. We also show additional improvements that result from enhanced load balancing and simplification of the communication between the core and the arbitration mechanism.

Categories and Subject Descriptors: C.1 [**Processor Architectures**]: Other Architecture Styles—*Heterogeneous (hybrid) systems*

General Terms: Design, Performance

Additional Key Words and Phrases: Chip multiprocessor, hardware accelerators, accelerator virtualization, accelerator sharing

ACM Reference Format:
Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, and Glenn Reinman. 2014. Architecture support for domain-specific accelerator-rich CMPs. ACM Trans. Embedd. Comput. Syst. 13, 4s, Article 131 (March 2014), 26 pages.
DOI: http://dx.doi.org/10.1145/2584664

1. INTRODUCTION

Power efficiency has become one of the primary design goals in the many-core era. While ASIC designs can provide orders of magnitude improvement in power efficiency over general-purpose processors, they lack reusability across different application domains, and significantly increase the overall design time and cost [Schaumont et al. 2003]. On the other hand, general-purpose designs can amortize their cost over many application domains, but can be 1,000 to 1,000,000 times less efficient in terms of performance/power ratio in some cases [Schaumont et al. 2003]. A recent industry trend to address this is the use of on-chip accelerators in many-core designs [Johnson et al. 2007, 2010; Seiler et al. 2009]. According to an ITRS prediction [ITRS 2011], this trend is expected to continue as accelerators become more common and present in greater numbers (close to 1500 by 2022). On-chip accelerators are application-specific

This work is supported by the CDSC funded by the NSF Expedition in Computing Award CCF-0926127, as well as the NSF Graduate Research Fellowship Grant no. DGE-0707424. It is also supported in part by C-FAR, one of six centers of STARnet, an SRC program sponsored by MARCO and DARPA.
Authors' addresses: J. Cong, M. A. Ghodrat (corresponding author), M. Gill, B. Grigorian, and G. Reinman, Computer Science Department, University of California, Los Angeles; email: ghodrat@cs.ucla.edu.

DOI: http://dx.doi.org/10.1145/2584664

implementations that provide power-efficient implementations of a particular functionality, and can range from simple tasks (i.e., a multiply-accumulate operation) to tasks of more moderate complexity (i.e., an FFT or DCT) to even more complex tasks (i.e., complex encryption/decryption or video encoding/decoding algorithms). We believe that future computing servers will improve their performance and power efficiency via extensive use of accelerators.

Accelerator-rich architectures also offer a good solution to overcome the *utilization wall* as articulated in the recent study reported in Venkatesh et al. [2010]. It demonstrated that a 45nm chip filled with 64-bit operators would only have around 6.5% utilization (assuming a power budget of 80W). The remaining *unutilizable* transistors are ideal candidates for accelerator implementations, as we do not expect all the accelerators to be used all the time.

We classify on-chip accelerators into two classes: (1) tightly coupled accelerators where the accelerator is a functional unit that is attached to a particular core (e.g., [Johnson et al. 2007; Jiang et al. 2009]); and (2) loosely coupled accelerators (e.g., [Nallatech 2011]) where the accelerator is a distinct entity attached to the Network-on-Chip (NoC), which can be shared among multiple cores. This article focuses on the efficient use of Loosely Coupled Accelerators (LCAs), which have been studied much less. These accelerators are not tied to any particular core, and can potentially be shared among all cores on-chip; this, however, requires some form of arbitration and scheduling.

In order to increase the utilization of accelerators and allow application developers to take advantage of the performance and energy consumption benefits they offer, it is necessary to reduce the overhead involved in their use. This overhead currently comes in the form of interacting with the Operating System (OS) that is responsible for managing accelerator resources. Another key issue in such accelerator-rich architectures is efficient management for sharing of accelerators among different cores and across different applications. Additionally, an application author who targets a platform featuring accelerators produces code that is bound to that platform, because accelerators are potentially unique to a given platform. We aim to develop an efficient architectural framework and an associated set of algorithms that minimize the overhead associated with both using accelerators and targeting a platform that extends accelerators to an application.

With these goals in mind, we propose an accelerator-rich CMP architecture framework, named ARC, with a low-overhead resource management scheme that: (i) allows accelerators to be shared and virtualized in flexible ways, (ii) is minimally invasive to core designs, and (iii) is easy for application programs to use. We then examine the strengths and weaknesses of our initial proposed system, and propose a modified system that addresses our identified shortcomings. Our article provides the following contributions:

—an accelerator allocation protocol to avoid OS overhead in scheduling tasks to shared, loosely coupled accelerators;
—an approach to accelerator virtualzation that allows multiple accelerators to work collaboratively, either as a single complex virtual accelerator or as multiple copies of a simple accelerator, in a way that is transparent to program authors;
—a fully automated simulation tool-chain to support accelerator generation and management.

The rest of the article is organized as follows. Section 2 reviews some related work. The architectural support for our proposed method is reviewed in Section 3. Section 3 also discusses the algorithms we have developed to efficiently share and virtualize accelerators. Sections 4 and 5 discuss our evaluation methodology and experimental

results which support our proposed methods, and finally we conclude in Section 6. A preliminary version of this work is presented in Cong et al. [2012].

2. RELATED WORK

There is a large amount of work that implements an application-specific coprocessor or accelerator through either ASIC or FPGA [Bouris et al. 2010; Cong et al. 2009]. These works mostly consider a single accelerator dedicated to a single application. Convey [Convey Computer 2008] and Nallatech [2011] target reconfigurable computing in which customized accelerators are off-chip from the processors, unlike our work which targets CMP architectures with on-chip accelerators. Some previous work considered on-chip integration of accelerators. Garp [Hauser et al. 1997], UltraSPARC T2 [Johnson et al. 2007], Intel's Larrabee [Seiler et al. 2009], and IBM's WSP processor [Johnson et al. 2010] are examples of this. Most of these platforms (except WSP) are tightly coupled with processor cores (or core clusters). Our article focuses on loosely coupled accelerators in a way where accelerators can be shared between multiple cores. OS support for accelerator sharing and scheduling is presented in Garcia et al. [2008]. In contrast, we focus on hardware support for accelerator management. To the best of our knowledge, this is the first work to address this issue.

There have also been a number of recent designs of heterogeneous architectures, like EXOCHI [Wang et al. 2007], SARC [Ramirez et al. 2010], and HiPPAI [Stillwell et al. 2009]. Similar to our work, EXOCHI's focus is on a heterogeneous nonuniform ISA. HiPPAI, like our work, eliminates system overhead involved in accessing accelerators, only it does so using a software layer (portable accelerator interface). SARC also has a core and accelerator architecture similar to our work, yet it also lacks a hardware management scheme. Unlike these works that focus on software-based methodologies, our approach fully advocates the use of hardware for managing and interfacing with accelerators.

Prior art has also explored accelerator virtualization. VEAL [Clark et al. 2008] uses an architecture template for a loop accelerator and proposes a hybrid static-dynamic approach to map a given loop on that architecture. The difference between our virtualization technique and theirs is that their work is limited to nested loops, while in our approach we seek any accelerator such that its composition can be described by some set of rules. PPA [Park et al. 2009] uses an array of PEs which can be reconfigured and programmed. PPA uses a technique called virtualized modulo scheduling which expands a given static schedule on available hardware resources. Again in this work the input is a nested loop, where in our approach this is not a limitation. DySER [Govindaraju et al. 2011] implements an FPGA-like accelerator, with fixed functionality arithmetic units coupled with a configurable communication network. Our work is distinguished from this by allowing for communication between distant accelerators, relying on a packet-switched NoC for communication, and in that our accelerators are intrinsically all shared.

3. ARCHITECTURE SUPPORT OF ARC

In an accelerator-rich platform, one main issue is increasing the utilization of accelerators and also making accelerators reusable between multiple applications. Our approach uses several techniques, namely accelerator sharing and accelerator virtualization. In the following sections we first discuss the motivation for our work and then show how we efficiently address these concerns.

3.1. Motivation

In a typical heterogeneous system which uses accelerators, when a core wants to access an accelerator, it does so by using an accelerator driver (OS call) [Garcia et al. 2008;

Table I. OS Overhead to Access Accelerators (Cycles)

Operation	1 Core	2 Cores	4 Cores	8 Cores	16 Cores
Open driver	214,413	256,401	266,133	308,434	316,161
ioctl (average)	703	725	781	837	885
Interrupt latency	16,383	20,361	24,022	26,572	28,572

Sun et al. 2007]. Using the Simics/GEMS simulation [Magnusson et al. 2002; Martin et al. 2005] platform to model a system consisting of Ultra-SPARC III-i processors running Solaris 10, we measured the delay for different system call operations related to asynchronously communicating with hardware devices. These results are shown in Table I (ioctl is the system call for device-specific operations). In an accelerator-rich platform, this simplistic approach becomes very inefficient, both in terms of energy and performance. If we assume that accelerators are going to be numerous and used frequently, this overhead becomes significant. Moving the arbitration mechanism and communication mechanism that an OS driver would provide into hardware would both reduce the cost of interacting with accelerators for each thread and also allow for highly efficient sharing of accelerator resources. Furthermore, course-grain functionalities traditionally implemented by accelerators are difficult to reuse, as they typically implement a single, highly specialized functionality. Many accelerators may, however, feature parts of their internal pipeline that is shared among multiple different accelerated algorithms. If these accelerators are broken up into smaller pieces and allowed to communicate with one another, then the common elements can be shared among similar accelerated functionalities. This would, however, result in an increased amount of configuration on the part of controlling software, which highlights the need to implement efficient mechanisms of communicating with accelerators if software will be expected to choreograph a communicating group of accelerators, rather than a single monolithic computation engine.

From this motivation, we produced two goals for this work. First, minimize the overhead associated with interacting with and arbitrating for accelerator resources, so as to make use of those accelerators practical. Second, by breaking accelerators up into smaller pieces such as to expose components that are common among multiple accelerators, we would increase the utilization of these common components, thus allowing for a wider range of accelerator functionality to be packed into a given chip area.

3.2. Microarchitecture of ARC

In this work, we introduce two architectures that are intended to address arbitration of accelerator resources. The first architecture features a Global Accelerator Manager (GAM) as the primary mechanism of performing arbitration. The second architecture is centered around a revised Global Accelerator Manager (GAM+). The latter is an improvement on the former, and addresses a number of shortcomings identified during our implementation and study of the former architecture. This section is split into two smaller sections. Section 3.2.1 describes ARC built around GAM. Section 3.2.2 discusses a number of shortcomings we identified in our original GAM, and introduces ARC built around GAM+.

3.2.1. ARC with GAM. Figure 1 shows the overall architecture of ARC which is composed of cores, accelerators, the Global Accelerator Manager (GAM), shared L2 cache banks, and shared NoC routers between multiple accelerators. All of the mentioned components are connected by the NoC. Accelerator nodes include a dedicated DMA-controller (DMA-C) and Scratch-Pad Memory (SPM) for local storage and a small Translation Look-aside Buffer (TLB) for virtual-to-physical address translation. GAM is introduced to handle accelerator sharing and arbitaration.

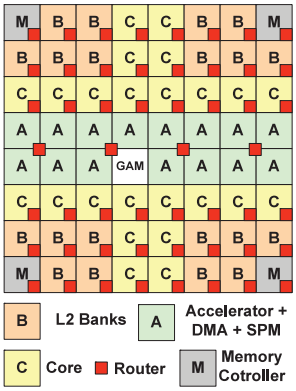

Fig. 1. Overall architecture of ARC.

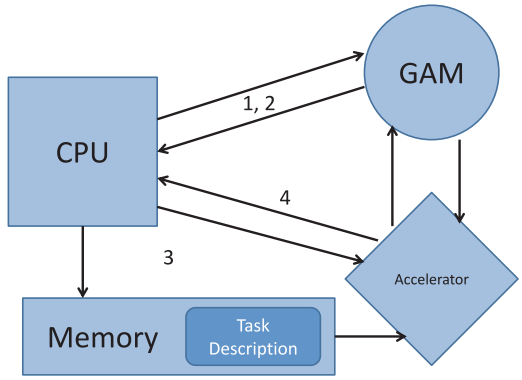

Fig. 2. Communication between core, GAM, and accelerator.

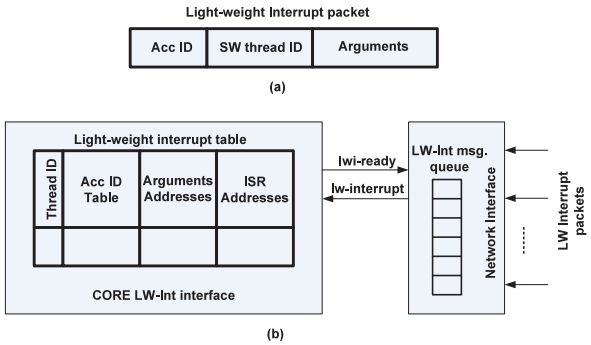

Fig. 3. Lightweight interrupt support.

In order to interact with accelerators more efficiently, we have introduced an extension to the instruction set consisting of four instructions used specifically for interacting with accelerators. These instructions are briefly described in Table II. A processor uses *lcacc-req* to request information about accelerator availability, consisting of pairs of accelerator identifiers and predicted wait times for each available accelerator. A processor will then use *lcacc-rsv* to request use of a specific accelerator. *lcacc-cmd* is used for interacting directly with an accelerator. When a job is completed, *lcacc-free* is used to release an accelerator to be used by another cpu. These instructions are accessible directly from user code, and do not require OS interaction. Communication with accelerators is done with the use of virtual addresses, accessing resources that are already accessible from user code. Execution of each of these instructions results in a message being sent to a device on the network, either the GAM or an accelerator. Attached to each of these messages is the thread ID of the executing thread that can be used to track requesting threads in an environment where context switches are possible.

Figure 2 shows the communication between a core, the GAM, an accelerator, and the shared memory detailing the use of an accelerator by a core. The numbers on the arrows in Figure 2 show the steps taken when a core uses a single accelerator. They are described next.

(1) The core requests an enumeration of all accelerators it may potentially need from the GAM (*lcacc-req*). The GAM responds with a list of accelerator IDs and associated estimated wait times.

Table II. Instructions Used to Interact with Accelerators

lcacc-req x	Request information from GAM about availability of accelerators implementing functionality *x*
lcacc-rsv x y	Reserve the accelerator with ID *x* for a predicted duration *y*
lcacc-cmd accl cmd addr x y z	Send a command *cmd* to an accelerator *accl* with parameters *x*, *y*, and *z*. Performs an address translation on *addr*, sending both logical and physical address.
lcacc-free accl	Sends a message to GAM releasing accelerator *accl*.

(2) The core sends a sequence of reservations (*lcacc-rsv*) for specific accelerators to the GAM. The core waits for the GAM to give it permission to use these accelerators. The GAM also configures the reserved accelerators for use by the requesting core.
(3) The core writes a task description detailing the computation to be performed to the shared memory. It then sends a command to the accelerator (*lcacc-cmd*) identifying the memory address of the task description. The accelerator loads this task description, and begins working.
(4) When the accelerator finishes working, it notifies the core. The core then sends a message to the GAM freeing the accelerator (*lcacc-free*).

3.2.2. ARC with GAM+. After implementing GAM and running a number of experiments, we identified some shortcomings in our system. Most significant of these was the inability to divide work among multiple accelerators, and potentially long wait times when a thread is stalled on waiting on accelerator reservations. To address these two problems, we revised the mechanism by which requesting threads communicate work to accelerators to divide the whole computation into a number of independent task groups. Each task group would not have any dependencies between any other task group, and can be scheduled in parallel. The GAM was also modified to act as an interface when communicating with accelerators, and assumed the additional roll of scheduling these task groups to accelerators, rather than relying on cores to interact with accelerators directly.

To accommodate these modifications, we replaced GAM with GAM+, which features a simple scheduling algorithm, a small local TLB, and a small table mapping requesting threads to information regarding the progress of accelerators. Instead of sending a task description to an accelerator directly, it would instead be sent to GAM+, and GAM+ would forward it to accelerators as resources became available. GAM+ would allocate a single task group to an accelerator at a time. After an accelerator finishes a task group, it notifies GAM+ that it is done. When all task groups are completed, GAM+ notifies the requesting thread that the entire computation is done. GAM+ similarly acts as a proxy for TLB misses encountered by the accelerators, but also serves as a filter to assure that TLB misses that are encountered by multiple accelerators do not result in redundant interrupts being sent to the requesting core.

Because of the simplification of communication described before, the instruction set no longer needs *lcacc-req*, *lcacc-rsv*, or *lcacc-free*. Only *lcacc-cmd* is required to send task descriptions to GAM+ and service TLB misses. The requesting thread is no longer needed to perform any interaction at all between the call to an accelerator and the time it is done with the assigned work. TLB misses would first check the local TLB of the original calling core, and in the event of a miss would trap the requesting core to a OS TLB miss handler directly. This simplified communication scheme also results in a reduction in the overhead involved in originally setting up an accelerator, and allows for complicated accelerator-core interactions to be modeled as independent accelerator calls. GAM+ allows a single requesting thread to utilize all accelerators of a given type in the entire system, furthering the original goal of increasing accelerator utilization.

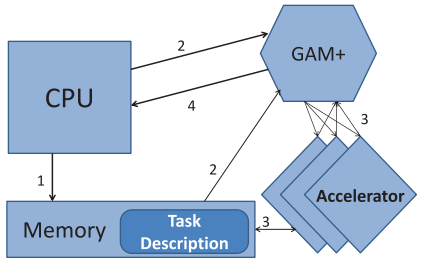

Fig. 4. Communication between core, GAM+, and accelerators.

In order to eliminate wait times for threads wanting to use an accelerator and move toward a more fair scheduling system, GAM+ interleaves task groups of multiple requesting threads. A simple round-robin scheduling policy was found sufficient to allow for fairness in accelerator use. This eliminates the problem of potentially long wait times in times of high contention, and results in a time slice model similar to that seen on conventional multiprocessors. While we did not implement a priority-based scheduling system, since we assumed all tasks to be of equal priority, implementing one would be simple.

The revised communication between devices in ARC featuring GAM+ is described in Figure 4, exempting TLB misses or errors found in the accelerator. The steps are as follows.

(1) The core writes a task description detailing the computation to be performed to shared memory.
(2) The core sends the address of the task description to GAM+. GAM+ reads the header in the task description to find how many task groups are expressed in this task description, and what kind of accelerators are needed.
(3) As accelerators become available, GAM+ forwards the task description to each accelerator, along with a specification of which tasks belong to the assigned task group. As accelerators complete their assigned task groups, they notify GAM+ of their progress.
(4) When all task groups are completed, GAM+ notifies the cpu that the entire computation is completed.

3.3. Lightweight Interrupt Support

A platform that features accelerators requires a mechanism for a processor to be notified of the progress of an accelerator. In the ARC platform, we handle this issue with the use of lightweight interrupts. ARC lightweight interrupts are interrupts handled entirely as user code, and do not involve OS interaction, as this interaction can be a major source of inefficiency. Table I shows the cost in cycles of interacting with accelerators through a device driver and the overhead associated with OS interrupts.

There are three main sources of interrupts associated with accelerator interaction in a system featuring GAM: (1) GAM responses; (2) TLB misses; (3) notification that the accelerator has finished working or has encountered an error. GAM responses come either because a core sent a request or a reserve message. TLB misses occur when an accelerator fails to perform address translation with the use of its own private TLB, and requires a core's assistance in performing the lookup. Interrupts notifying the completion of work arrive when an accelerator has completed all work given to it.

In systems featuring GAM+, interaction with the core is simplified, and there are only two sources of interrupts: (1) GAM+ notifying the core that work is completed or

Table III. Instructions to Handle Lightweight Interrupts

lwi-reg x y z	Register service routine *y* to service interrupts arriving from accelerator *x*. LWI message packet will be written to *z*
lwi-ret	Return from an interrupt service routine.

an error has occurred, and (2) TLB misses. Both of these interrupt types come to the core from GAM+, and are handled similarly to those of GAM.

Figure 3 shows the microarchitecture components added to the cores in ARC in order to support the lightweight interrupt. An interrupt is sent via an interrupt packet (shown in Figure 3(a)) through the NoC to the core that requested an accelerator. Each interrupt packet includes the thread ID which identifies the thread which this interrupt belongs to, and a set of interrupt-specific information. The main microarchiteture components added to support lightweight interrupt are listed next.

(1) *Interrupt controller located at the core's network interface.* This is responsible for receiving the interrupt packets and queuing them until being serviced by the core.
(2) *Lightweight interrupt interface in the core.* This is responsible for: (1) receiving the interrupt from the interrupt controller, (2) providing a software interface to set up the information needed to service the interrupt.

The interrupt controller has a queue for buffering the received interrupt packets, so they don't get lost if the core is busy handling other interrupts. Without loss of generality we assume that for each thread we can only have one level nest for interrupt. This means only one lightweight interrupt may be serviced at a time, and other interrupts that arrive during the servicing of a lightweight interrupt are left pending. If an interrupt arrives for a thread that is currently scheduled, it is executed immediately. If the thread is not scheduled, a normal OS-based interrupt occurs.

In order to support lightweight user-level interrupts, we introduce a set of instructions to enable user code to handle interrupts. These instructions are described in Table III. *lwi-reg* registers the interrupt handlers. *lwi-ret* returns from an interrupt handler routine. A program segment using accelerators is then designed as a series of interrupt service routines.

3.4. Invoking Accelerators

Whether using GAM or GAM+, the initial overhead associated with beginning to use an accelerator is large enough that it should be amortized over a large amount of work. To that end, we introduce two accelerator features that explicitly deal with efficiently processing large amounts of data: (1) task descriptions to limit communication between accelerators and the controlling core, and (2) methods to handle TLB misses.

To communicate with an accelerator, a program would first write to a region of shared memory a description of the work to be performed. This description includes location of arguments, data layout, which accelerators are involved in the computation, the computation to be performed, and the order in which to perform necessary operations. This detail is included to allow accelerators to be both general, and to allow coordination of accelerators in groups to perform more complex tasks through virtualization (described in Section 3.7). Evaluating the task description yields a series of steps to be performed in order, with each step consisting of a set of memory transfers and computations that can be executed concurrently. This allows accelerators to overlap computation with memory transfer within a given step. When all computations and memory transfers of a given step are completed, the accelerator moves onto the next step. In this work, we refer to these individual steps as tasks, and the structure detailing a sequence of tasks

as a task description. In the case of GAM+, we refer to a collection of tasks performing the same computation on different data as a task group, and a single task description would typically describe many task groups.

To further decouple the accelerator from the controlling core, each accelerator contains a small local TLB. This is required because the accelerator operates within the same virtual address space as the software thread that is using the accelerator. The accelerator relies on the controlling core to service any detected TLB misses. In a system featuring GAM, it does this by sending a lightweight interrupt to the controlling core when a TLB miss occurs with the address that caused the TLB miss. In a system featuring GAM+ the TLB miss is first sent from the accelerator to GAM+, and then from GAM+ to the controlling core if the page was not found in the TLB of GAM+. Handling this interrupt would involve the core executing the same TLB miss handler that is executed when the core normally encounters a miss in its own TLB. Because this is an OS action, and involves trapping to an OS handler regardless, it is not actually necessary that the original software thread that is using the accelerator be currently scheduled. If it is scheduled, the lightweight interrupt interface can be used to limit overhead associated with interrupt handling. Otherwise, the OS can be notified directly (e.g., by invoking a software interrupt or real hardware interrupt) without having to wait for or force a context switch to reschedule the controlling thread. The resolved address is then sent back to the accelerator that had encountered the TLB miss in systems featuring a GAM, or to the GAM+ in systems featuring a GAM+.

3.5. Sharing Accelerators with GAM

Sharing of accelerators in a system featuring a GAM is done jointly between software and the GAM. The GAM is responsible for providing a thread with information regarding current accelerator utilization including availability and estimated wait times, and arbitrating among threads, while the software thread is responsible for making use of the accelerators after allocation. Even in systems that feature many accelerators, a specific accelerator resource might be a point of high contention, and thus a bottleneck. In this situation, some of these threads may choose to eschew the use of the accelerator and simply execute the task to be offloaded using their own core resources. While the core is certainly less power efficient in executing this task, it may make sense for it to do so in situations where the wait time for an accelerator will eliminate any potential gains. In this article we propose a sharing and management scheme which can dynamically determine whether the core should wait to use an accelerator or should instead choose a software path, based on an estimated waiting time. This proposed sharing and management strategy is performed by the GAM. The GAM tracks: (1) the types of available accelerators; (2) the number of accelerators of each type; (3) the jobs currently running or waiting to run on accelerators, their starting time and estimated execution time (Section 3.5.1); (4) the waiting list for each accelerator and the estimated runtime for each job in the waiting list (Section 3.5.2). In cases where a task runs for a duration that is significantly longer than the estimated runtime, the GAM will evict a running task and raise an exception indicating this eviction to the controlling thread, and that the task was not successfully completed.

3.5.1. Accelerator Runtime Estimation (by the Core). The execution time of a certain job on an accelerator is data dependent. When an accelerator is reserved, the requesting thread submits an estimation of the duration for which the accelerator will be used. This estimate is determined with the use of a data-size-parameterized regression model, which is constructed by profiling several different executions of a given accelerator and fitting a low-order polynomial curve to the observed runtime. We empirically found

that a simple second-order polynomial is sufficient to estimate execution time within 1 to 2% on average (at most 6%). Once a model is generated for an accelerator, it is provided to the rest of the development flow via the accelerator DLL (see Figure 7).

3.5.2. Accelerator Wait-Time Estimation (by the GAM). After receiving the reserve request message from the core, the GAM will add the requesting core's ID to the tail of the waiting list for that accelerator. Note that the tasks being tracked in this waiting list are issued on a First-Come-First-Serve (FCFS) basis. Hence, the estimated wait time for the task being added to the end of the list can be derived by summing up the expected execution times of all jobs that already exist in the waiting list for that accelerator. This estimation algorithm is both simple and practical for hardware implementation.

3.6. Sharing Accelerators with GAM+

Systems featuring a GAM+ exhibit a simplified arbitration mechanism in the form of a hardware scheduler. When multiple on-chip cores are competing for the same accelerator, each of them simply sends their respective task descriptions to GAM+, and are not responsible for directly acquiring, scheduling, or freeing accelerator resources. The hardware scheduler located in GAM+ will iterate over all local task descriptions, and choose tasks to schedule. This results in an elimination of the time spent waiting for software to free accelerators or run reservation interrupt service routines, allowing accelerators to more quickly move from one task group to another. While this scheduling mechanism could potentially result in a performance deficit if there is no contention, a single accelerator, and the size of each individual task group is small compared to the number of task groups, we found in most cases GAM+ resulted in a net gain. Additionally, GAM+ had no need for a wait-time estimation, since no thread had to wait for the whole duration of a computation to complete.

3.7. Accelerator Virtualization

A key contribution of our work is to increase the utilization of the available accelerators by either composing different accelerator types to create new types of accelerators or to compose the same type of accelerators to create a larger accelerator. In the next two sections we discuss these two techniques.

3.7.1. Accelerator Chaining. In an accelerator-rich platform, there are many cases when the output of one accelerator feeds the input of another accelerator (like many streaming applications). In a traditional system, these two accelerators communicate through system memory, that is, the controlling core reads the output of the first accelerator from its SPM, stores it to shared memory, and writes it to the second accelerator's SPM. To remove this inefficiency, two DMA-controllers can communicate and the source DMA-controller can send the content of its SPM to another DMA-controller to be written in its SPM.

3.7.2. Accelerator Composition. For many types of problems, it is not practical to provide an accelerator to directly solve each possible problem instance. Additionally, it is not practical to demand that an application author target a single architecture. For this reason, we provide a set of virtual accelerators to decouple hardware design and software development. A virtual accelerator is an accelerator that is implemented as a series of calls to other physical accelerators, available in hardware (Figure 5(a)). A large library of virtual accelerators can be provided to the application author as if they were implemented in hardware. These accelerators would actually be implemented as a series of decomposition rules that break down a large problem into a number of smaller problems (Figure 5(b)), similar in style to the approach presented in Puschel et al. [2005]. These small problems would then be solved directly by hardware. These

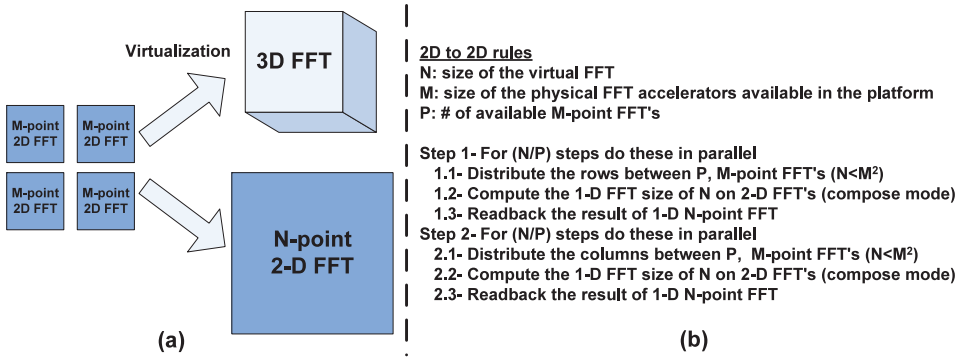

Fig. 5. An example of accelerator virtualization.

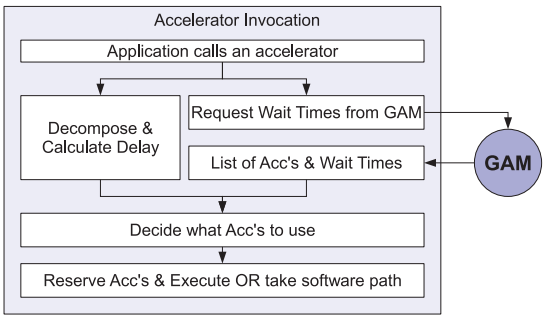

Fig. 6. Accelerator virtualization steps.

rules describe two things: (1) computation that must be performed by accelerators capable of solving subproblem instances, and (2) how data is communicated to, from, and between these various smaller accelerators. Rules would be applied recursively to express an implementation for each virtual accelerator in terms of calls to physical accelerators.

These statically determined decomposition rules can thus be applied at runtime. Figure 6 describes the process of invoking a virtual accelerator from within the application binary.

In systems featuring GAM, software is responsible for selecting the best decomposition strategy for numbers of accelerators. When preparing to decompose an accelerator, a *lcacc-req* message is sent to the GAM for wait times for all functional units that may be required by the decomposition result. While waiting on this request, the requesting core either begins calculating the decomposition or begins fetching the data structures associated with the statically computed solution. Once the GAM responds and the requesting core has a fully decomposed problem available, the core calculates the wait time for the entire computation. It does this by adding the delay calculated with the use of the regression model to the largest of the delays provided by GAM. The core then executes a series of *lcacc-rsv* instructions for each required accelerator, specifying the wait time for the entire operation as the estimated duration of use of each accelerator reserved. GAM will not assign any accelerators until it can assign all accelerators requested. The core releases accelerators in the same way as it normally would. With these mechanisms, an application author can use a simple API to invoke virtual accelerators, and a hardware developer can implement accelerators based on need and available resources.

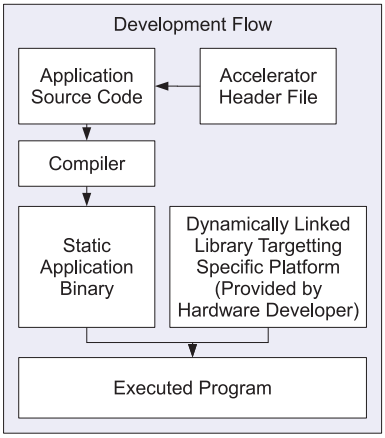

Fig. 7. ARC development flow.

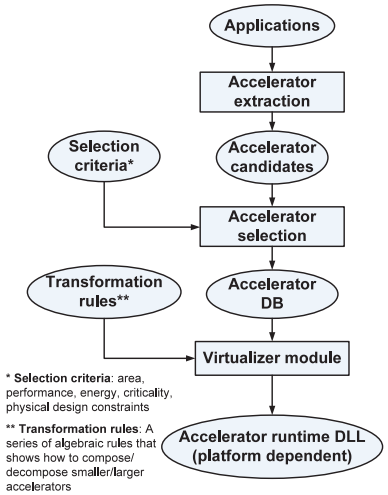

Fig. 8. Accelerator extraction methodology.

In systems featuring GAM+, decomposition is simpler. The core simply relies on a single decomposition strategy that produces the most simple final result, and relies on GAM+ to achieve higher performance by scheduling this single strategy to multiple copies of the communicating accelerator graph. This simplifies the task of the software while still allowing for high performance in most cases.

3.8. Programming Interface to ARC

The Application Programming Interface (API) involved in using accelerators is presented in Figure 7. For each type of accelerator, one Dynamic Linked Library (DLL) is provided. This DLL is specific to a target platform, and provides a mapping from accelerator calls to actual invocations of physical accelerators. Calls to accelerators have their implementations dynamically linked to application code.

3.9. Accelerator Extraction Methodology

Figure 8 shows the block diagram for accelerator extraction from a given application. Given an application, using a combination of static analysis and profiling, a list of candidates for accelerators are extracted. These candidates are weighted using a series of selection criteria, like area, performance, energy, criticality, and physical design constraints. This step generates an accelerator database that can be used together with a series of transformation rules to create larger or smaller accelerators from the available accelerators in the platform. This step is handled by a software module called the virtualizer, which outputs a DLL that is used to link to executable files shown in Figure 7.

3.10. Security

Eliminating OS involvement in interacting with accelerators means measures must be taken to assure that buggy or malicious programs cannot use accelerators to interfere with other running programs. Toward this end, we introduced functionality to extend safety guarantees to programs running on the ARC platform. Accelerators present four challenges, each of which is addressed in ARC: (1) a thread interferes with another thread's accelerator use, (2) a thread uses an accelerator to access memory it would not otherwise have permission to access, (3) a thread uses an accelerator to interfere with

another thread's accelerator use, and (4) a thread starves the system of accelerator resources. First this section will describe the mechanisms that protect against these problems for a system featuring GAM, and then for a system featuring GAM+.

3.10.1. ARC with GAM. To eliminate a thread's ability to interfere with the accelerator use of another thread, we introduce a strict concept of accelerator ownership. Each accelerator is owned by one thread at a time, and ignores all messages sent to it by other threads. Messages sent to accelerators or the GAM using our introduced instruction set extension have a thread ID attached to them that is filled by reading a privileged register. Accelerators are configured with an owning thread by the GAM. This is done immediately prior to the GAM sending a message to the controlling core notifying it that the accelerator is available for use. When an accelerator is freed, the GAM clears its owning thread.

As detailed in Section 3.4, accelerators feature a small local TLB, and operate over virtual addresses. When a miss occurs in this small local TLB, a message is sent to the controlling core to satisfy this TLB miss. It is the responsibility of the interrupt handlers to execute the normal TLB service routine and respond with the correct page table entry. Because execution of the *lcacc-cmd* instruction performs the translation it is not possible for a core to falsify the page table entry. Also, because the TLB service routine is executed on the controlling core memory protection can be enforced exactly as is done for normal user code. To prevent the accelerator's local TLB from containing stale entries, when the OS unmaps a page from an accelerator-using thread, the OS notifies the GAM of the page table entry being unmapped, which in turn forwards the notification to all accelerators owned by that thread. If any accelerator has that page table entry in its local TLB, the accelerator raises an error causing the OS to terminate the owning thread's process. Additionally, the accelerator's local TLB is cleared when it is assigned a new owning thread.

A handshaking process is performed when data is sent from one accelerator to another. This is done to eliminate the possibility that a malicious thread will attempt to use an accelerator to contaminate the memory of another accelerator. An accelerator will not write to another accelerator's SPM without first being invited to do so by the accelerator that it would write to. As a result, attempting to configure an accelerator to write to an accelerator owned by another thread will result in the source accelerator waiting indefinitely. Likewise, an accelerator emitting a request to read from another accelerator's SPM without a corresponding write request will also hang.

Lastly, the estimated runtime sent to the GAM described in Section 3.5.2 is used to eliminate the concern of starvation. If it is found that an accelerator is not released after a period much larger than the expected execution time, the GAM evicts the current owning thread from all accelerators it has allocated. The GAM then sends a message to the controlling core notifying it of this eviction. The controlling thread is allowed to rerequest accelerators, but it will be put to the end of the wait queue. A thread is able to perform large computations with the use of accelerators without risking eviction by submitting a large runtime estimate. This eviction mechanism also mandates that the estimated runtime must be somewhat accurate. The runtime estimates also feature a hard upper bound, necessitated both by starvation prevention and engineering pragmatism. If an accelerator is needed for a period longer than this, the computation must be broken into several individual accelerator invocations, between which arbitration must be repeated. Additionally, since estimated runtimes are used in aggregate by the GAM to supply requesting threads with an estimated wait time, threads that make the decision to wait for an accelerator are aware of how long they will be waiting. Because of this, threads can elect to fall back to software in cases of high contention.

3.10.2. ARC with GAM+. In this system configuration, GAM+ is the only device in the system capable of communicating directly to accelerators. For this reason, it is not possible for a thread to communicate with any accelerator directly, and is thus not possible for a thread to interfere with the computation of another thread's computation. Additionally, since accelerators are never in a state of being allocated to service a thread, but not yet performing computation, it is not possible for an accelerator-using thread to hold an accelerator indefinitely, thus eliminating the possibility of starvation. This assumes that accelerators themselves are performing computations that are guaranteed to terminate.

When GAM+ receives a task description, it will be responsible for providing a mapping from accelerator IDs specified in the task description to actual physical accelerator IDs. Since GAM+ establishes this mapping as an atomic action, there is no chance of an accelerator being configured to communicate with an accelerator that is not also configured to be part of the same computation, and is thus also in use by the same thread. This eliminates the possibility that an accelerator interferes with computations being performed by accelerators in use by a different thread.

Lastly, GAM+ features a TLB that extends memory protection in the same way that is extended by individual accelerators in the conventional ARC with GAM system.

3.11. Implications for Accelerator Design

A system that features a GAM or GAM+ puts constraints on the type of LCAs that can be part of the system. GAM demands that execution time of an LCA be finite, and also that it can be estimated. LCAs are assumed to be shared, and arbitration through GAM is not intended for LCAs that are permanently associated with certain threads or processes. GAM+ also adds a performance goal, though not a requirement, of LCAs performing tasks that are data parallel at some level. The granularity of data parallelism dictates the minimum task group size for which correct computation can be performed, and thus the limit on the degree to which work can by dynamically load balanced among multiple physical LCAs.

4. EVALUATION METHODOLOGY

In this section we first introduce the benchmarks and application domains used to evaluate ARC (Section 4.1), followed by a description of our automated tool-chain (Section 4.2) and customized simulator infrastructure (Section 4.3).

4.1. Benchmarks

To illustrate the effectiveness of our ARC platform, we evaluate a number of benchmarks from four distinct application domains: (1) Medical Imaging (MI), (2) CoMmercial (CM), (3) computer VISion (VIS), and (4) NAVigation (NAV). When analyzing contention between multiple threads executing in the same benchmark, we insert a barrier immediately before entering the benchmark kernel that is targeted for acceleration. This is done to maximize the observable effects of contention and model a worst-case scenario. All threads executing a benchmark can then be expected to enter this kernel at approximately the same time.

Our benchmarks are now described in more detail.

—*Deb*. Deblur features a series of phases consisting either of simple stencil-type computations, or application of a blur filter. The blur filter involves dependencies between computations, and results in the accelerator implementing the blur filter being constrained in the amount of parallelism it can exploit. All computations found in Deblur are cache friendly and very simple, allowing software to perform well.

—*Den*. Denoise features cache-friendly tiled stencil computations that are trivially data parallel. While the computation performed per element is small, the accelerator primarily sees benefit due to the use of a DMA-to-accelerator memory transfer.
—*Reg*. Similar to Deblur, Registration features a series of stencil-type computations interspersed with the computation of blur filters. The exact same blur filter accelerator of Deblur is used for Registration, and this benchmark exhibits the same performance characteristics for all the same reasons as Deblur.
—*Seg*. Segmentation features an access pattern similar to that of Denoise. The computation is more complex in software, but exhibits very little impact on throughput when implemented in hardware.
—*CS_MR*. Compressive Sensing spends a majority of time repeatedly computing a 3D FFT on complex numbers. The remaining computation has trivial impact on total execution time. Our accelerator performs this FFT very efficiently, in an almost fully streaming fashion along each dimension. The speedup observed, however, is relatively meager because we compare it against FFTW3 [Frigo et al. 2005], which takes full advantage of our software platform's vector instruction extensions, and is by far our most efficient software point of comparison.
—*BSc*. Black-Scholes is trivially data parallel, but exhibits no data reuse. The computation being performed is also simple. For this reason, the accelerator is heavily communication bound.
—*Stc*. StreamCluster requires algorithmic modification in order to be implemented as an accelerator. This modification results in computation of a partial result on a single pass over the input data, followed by a second pass that consumes the partial result and produces the final result. This modification, which is only for our accelerated version (i.e., software version is unchanged), significantly increases the total memory footprint required to perform the computation and thereby impacts performance.
—*LPCIP*. LPCIP examines multiple features randomly distributed throughout an image. Many of these features potentially overlap, which causes a significant amount of the target image to be stored in L2 cache. The result of this is an opportunity to be computation bound, due to highly efficient cache behavior for both software and the accelerator.
—*SURF*. Similar to StreamCluster, SURF requires algorithm modification to convert it into a form that an accelerator can efficiently target. This modification results in a large increase in the amount of memory being moved, which impacts performance negatively for the accelerator.
—*TexSyn*. Texture Synthesis can be implemented highly efficiently in an accelerator, except that there are a large number of coarse-grain data dependencies at statically identifiable points. For an accelerator, this means synchronization is enforced via an independent accelerator invocation on each pixel row of the generated image.
—*RobLoc*. Robot Localization features highly uniform memory accesses and is streaming in nature, with nearly no data reuse across multiple computations.
—*DispMap*. Disparity Maps consists generally of two phases: calculation of line integrals, and computation on the result of the line integrals. The calculation of line integrals involves a highly data-dependent sum operation; our accelerator therefore has long latency due to lack of exploitable parallelism.
—*EKF_SLAM*. EKF_SLAM is dense linear algebra that features a large computation load. The speedup we observe in this case is typical for computations that are compute-bound in software, but can be implemented efficiently in hardware.

Table IV. Benchmark Descriptions

Domain	Application	Algorithmic Functionality	# LCAs
MI	Denoise (Den) [Vese et al. 2004]	Total variation minimization	3
MI	Deblur (Deb) [Tadmor et al. 2008]	Total variation minimization and deconvolution	4
MI	Registration (Reg) [Yanovsky et al. 2008]	Linear algebra and optimizations	7
MI	Segmentation (Seg) [Chan et al. 2001]	Dense linear algebra, spectral methods, and MapReduce	1
MI	Compressive Sensing (CS_MR) [Lustig et al. 2007]	Dense linear algebra, FFT	6
CM	Black-Scholes (BSc) [C.Bienia et al. 2008]	Stock option price prediction using trivial floating point math	1
CM	Streamcluster (Stc) [C.Bienia et al. 2008]	Clustering and vector arithmetic	4
CM	Swaptions (Swp) [C.Bienia et al. 2008]	Computation of swaption prices using Monte Carlo (MC) simulation	4
VIS	Log-Polar Coordinate Image Patch (LPCIP) [Jurie 1999]	Log-polar forward transformation of image patch around each feature	1
VIS	Speedup Up Robust Features (SURF) [Bay et al. 2008]	Feature orientation and computation of gradient histogram	4
VIS	Texture Synthesis (TexSyn) [Venkata et al. 2009]	Procedural generation of texture image from patch; uses random number generation and random memory access	5
NAV	Robot Localization (RobLoc) [Venkata et al. 2009]	Monte Carlo Localization using probabilistic model and particle filter	1
NAV	Disparity Map (DispMap) [Venkata et al. 2009]	Calculate sums of absolute differences and integral image representations using vector arithmetic	4
NAV	Extended Kalman Filter-based Simultaneous Localization and Mapping (EKF_SLAM) [Blanco 2008]	Partial derivative, covariance, and spherical coordinate computations	2

Table V. Data Size Selection

Application	Data Size
Deb, Den, Reg, Seg, CS_MR	1 image cube of size $64 \times 64 \times 64$
BSc	131072 datasets of size 28B (containing stock option information)
Stc	stream of 16378 64-dimensional data points
Swp	16378 datasets of size 100B (containing option information)
LPCIP	32768 features from images of size 640×480
SURF	128 features from image of size 640×480
TexSyn	1 texture image of size 512×512
RobLoc, EKF_SLAM	131072 sets of sensor data (observation and motion)
DispMap	2 images of size 64×64

Table IV summarizes the selected benchmarks, providing brief descriptions of each application's computational characteristics and listing the number of accelerators used.

In addition, Table V shows the data sizes selected for each benchmark. The data sizes were adjusted in order to balance out the computation times of the applications in each domain. Since different applications may vary in the amount of time taken to compute over a given data point, the applications may vary in data size (even for applications in the same domain).

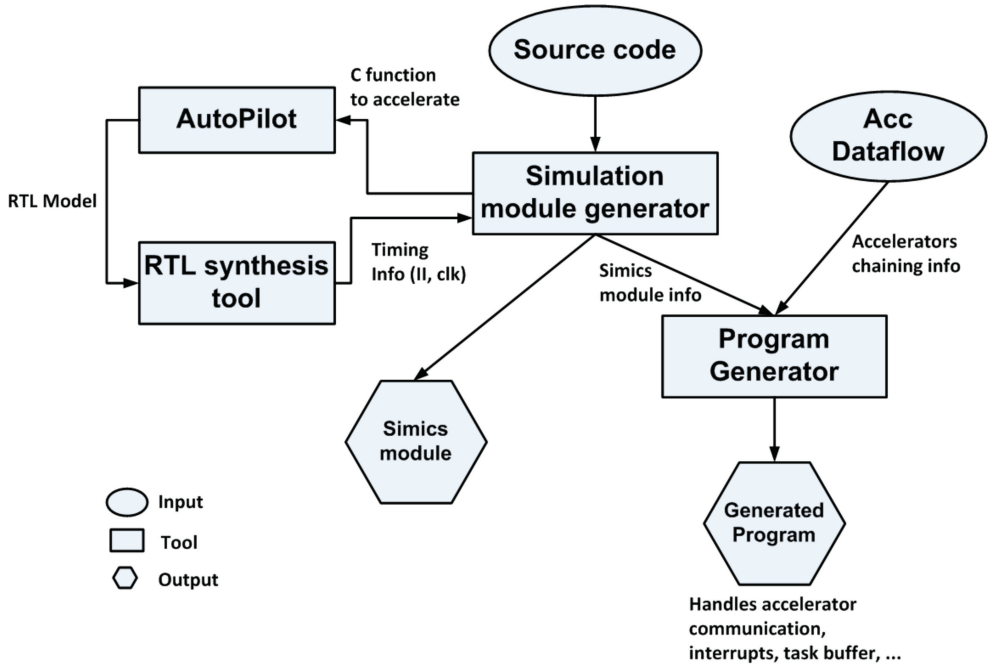

Fig. 9. Process used to generate simulation structures and accelerator using programs.

4.2. Automated Tool-Chain

In order to make the exploration of this topic practical, a number of supporting tools were created. These tools simplified the authoring of programs that used accelerators, and automated the process of implementing our chosen accelerators in our simulator framework. These tools were used in place of hand-written implementations and hand-adapted benchmarks to allow us to simulate systems that would have been prohibitively complex to manually author, such as those that utilized many accelerators or featured complicated inter-accelerator communication. Additionally, we believe that this is representative of what will be done in the development of future accelerator exploiting libraries, to simplify the job of programmers who would use these libraries without compromising any of the capabilities of these accelerators. Figure 9 illustrates the workflow described here.

With this tool-chain, generation of accelerators is only a matter of identifying a function in an application's source code to accelerate. We have automated the process of extracting these functions, compiling them into RTL models (using the AutoPilot behavioral synthesis tool [Cong et al. 2011]), and synthesizing these RTL models into ASIC implementations (using the Synopsys design compiler [Synopsys 2013]) to extract timing, area, and power information. Each synthesized result is then encapsulated into a simulator module by the *simulation module generator*. This yields a component that plugs into our cycle-accurate simulation infrastructure (detailed in Section 4.3), allowing it to be modeled as a hardware unit and executed in a pipelined fashion.

Once we select the functions we want to accelerate, typically encompassing the kernel of the benchmark, the *program generator* procedurally generates a program segment to use these accelerators. Communication between accelerators is described in a simple dataflow language that we use to generate C source code. These program segments together make up the platform-specific DLL mentioned earlier. This code

is responsible for coordinating interactions between accelerators, registering/handling interrupts, managing task descriptions, assigning accelerator resources, and dealing with synchronization between accelerators and the CPU.

4.3. Simulator Infrastructure

In this work we use a modified version of the GEMS [Martin et al. 2005] cycle-accurate simulator, which runs on top of Simics [Magnusson et al. 2002]. The base GEMS platform provides us with a timing model for processors, caches, and a packet-switched NoC. Normally, the NoC is used to connect a system of cache banks, processors, and memory controllers. Furthermore, the types of traffic these devices are capable of sending are rigidly defined, and only serve the purpose of implementing a cache coherence protocol. We therefore have added a new device that could be attached to the network, acting as a network interface stub capable of sending/receiving arbitrary data in a point-to-point fashion. A new packet type that could represent variable-sized datagrams has also been defined. The stub device provides an interface to which we could attach new devices that communicate with a protocol independent of the existing coherence protocol.

Our accelerators, GAM/GAM+, and processors are each attached to a network interface stub, allowing for flexible communication among all of these components. To keep the packets sent to/from the network interface stubs from interfering with the operations of the underlying coherence protocol, we have also added a new virtual channel to the NoC. As it is apart from those conventionally used by the coherence protocol, this channel eliminates the possibility of traffic from our communication protocols resulting in deadlocks in the coherence protocol. To allow for sending messages to/from devices, our processor model has also been augmented with an interface for timed messaging. These timed messaging interfaces are then used in the construction of the custom instructions (described in Section 3.2.1) and the interrupt mechanism (described in Section 3.3). For communicating interrupts, we have also added to each execution context a bit of state information, which acts as a jump-table for software-registered interrupt service routines.

Our next modification has been the addition of a device that acts as an arbitration mechanism, referred to as the GAM or GAM+. This device is affixed to one of the stub network interfaces mentioned before. Internally, this device is a simple state machine. A combination of the Autopilot behavioral synthesis tool [Cong et al. 2011] and Synopsys design compiler [Synopsys 2013] is used to synthesize critical pathways in the GAM/GAM+ for the purposes of calculating timing information of various actions taken for managing accelerator use. This timing information is then back-annotated into our simulated model of the GAM/GAM+.

Finally, we have added the accelerators themselves. Instead of authoring a set of specific accelerators, we have authored two separate mechanisms (described in Section 4.2) that allow for general-purpose accelerator incorporation: (1) a tool-chain that converts a function described in a high-level language into a compute engine; (2) a *simulation module generator* that encapsulates an arbitrary compute engine and allows it to be used in our simulator. The tool-chain not only generates the compute engine, it also provides both functional and timing modeling of a DMA engine, scratch-pad memories, and a small bit of circuitry used for decoding messages sent from CPUs and the GAM/GAM+. To generate a compute engine, this automated flow begins with a C kernel, which includes annotations used to describe control register state and data transfers from memory. This provides sufficient information to allow the AutoPilot and Synopsys tools to synthesize the C modules into ASIC, generating timing, area, and power values for the compute engine. These values, along with the original C code, are then inputted into the *simulation module generator*, which exposes a software interface by which the compute engine could interact with the rest of the system. Essentially, it

Table VI. Simics/GEMS Configuration

CPU	Ultra-SPARC-III-i @ 1.0GHz
Number of cores	1, 2, 4, 8, 16
Coherence protocol	MESI 2-level. Private L1. Shared
L1 cache	32 KB, 4 way set-associative
L2 cache	8 MB, 8-way set-associative
Memory latency	450 cycles
Network topology	Mesh
Operating System	Solaris10

Table VII. Synthesis Results

Domain	Application / Component	Area(um^2)	Power(mW)	SPM Banks
MI	Denoise	4.97E+05	16.50	6
	Deblur	2.01E+06	110.90	13
	Registration	3.85E+06	183.90	15
	Segmentation	6.88E+05	27.30	3
	CS_MR	3.66E+06	67.67	64
CM	BlackScholes	2.61E+06	51.60	7
	StreamCluster	1.85E+05	4.94	28
	Swaptions	2.52E+07	433.80	140
Vis	LPCIP	4.80E+05	10.10	5
	SURF	1.28E+08	2906.66	20
	Texture Synthesis	1.05E+06	29.80	159
Nav	Robot Localization	6.43E+06	119.00	8
	Disparity Map	4.79E+05	12.27	12
	EKF_SLAM	6.21E+07	951.00	8
-	SPM bank (2KB, 2R, 1R/W)	3.70E+04	17.50	-
-	DMA-C	1.01E+04	0.59	-
-	GAM	1.23E+04	0.59	-
-	GAM+	2.07E+04	0.66	-

is responsible for generating a module that meets the specifications of our simulator infrastructure, incorporating all the synthesized timing information into a form usable at runtime. Overall, this process ensures cycle-accurate accelerator simulations, and provides timing, area, and power measurements, which are combined with CPU power values from the McPat [Li et al. 2009] tool, for obtaining the overall area utilization and energy consumption of ARC.

The machine we model in our experiments is based on a multicore system consisting of a mix of Ultra-SPARC-III-i processors and accelerators. In order to create a fair comparison between machines of different configurations, we maintain a fixed cache and network configuration. Our network topology is a mesh modeled on a system normally used to support 32 processors. These nodes are then configured to either be processors, accelerators, or empty sockets. We feature a per-processor split L1 cache, and a distributed L2 spread across all nodes that rely on a directory-based coherence protocol. Table VI shows the machine configurations we model in our simulations, while Table VII shows synthesis results for the benchmark accelerators and ARC components.

5. EXPERIMENTAL RESULTS

In Cong et al. [2012], we have shown the following.

—the benefits of the ARC platform in a multiprocessor system when sharing LCAs between multiple cores;
—the benefits of using ARC in reducing OS overhead for LCA arbitration;
—the GAM estimation accuracy;
—the benefits of using a lightweight interrupt protocol.

In this section, we explore other important aspects of ARC, namely performance improvement, energy savings, and OS overhead reduction for new domains. We also show the benefits of using GAM+ over GAM, and explore the design space for ARC. We have used the following schemes for ARC evaluation.

—*Original benchmark (SW-Only)*. The baseline for the experiments is the native (nonsimulated) execution of these benchmarks on a 2 GHz Intel Xeon E5405 core.
—*Accelerators + HW management (GAM)*. This system features the architectural enhancements and hardware resource arbitration/management of the GAM.
—*Accelerators + HW management (GAM+)*. This system features the architectural enhancements and hardware resource arbitration/management of the GAM+.

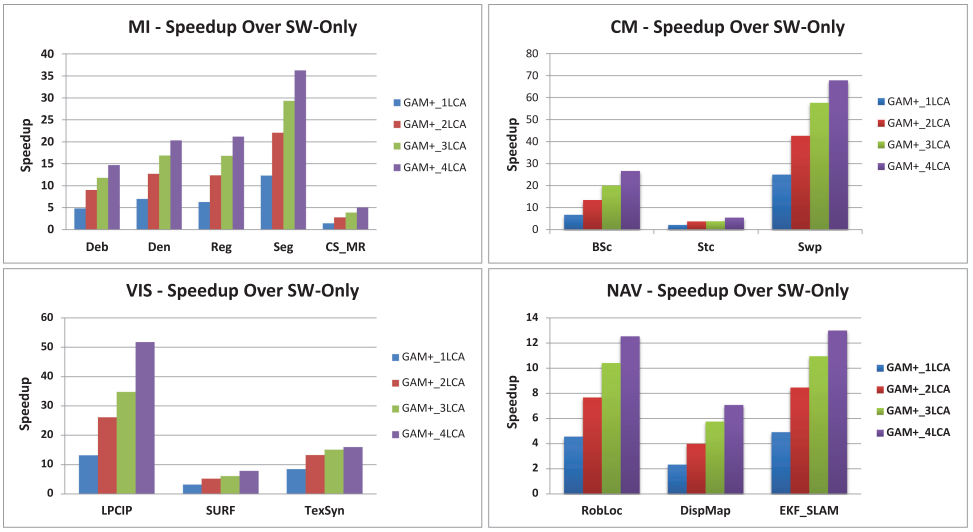

Fig. 10. All domains - performance improvements of GAM+ over SW-only versions.

5.1. Performance and Energy Improvement over Software

Figures 10 and 11 show the speedups and energy improvement we get for all the target domains for an ARC platform featuring GAM+. An increase in efficiency moving from 1LCA to 4LCA shows the benefit of dynamic load balancing. Energy efficiency trends closely with performance benefit primarily because of the relatively small impact that running accelerators has on the total consumed energy. The majority of the energy consumed is contributed by the NoC and associated caches. For this reason, running additional accelerators has a minimal impact on total consumed energy as compared to the benefit resulting from reducing overall runtime.

Some benchmarks do not scale as cleanly when increasing the number of accelerators, such as SURF, Texture Synthesis, and Stream Clusters. These two benchmarks in particular feature a large number of relatively small accelerated regions. For this reason, the amount of work that can be distributed amongst multiple accelerators is relatively small, resulting in uneven distribution in some regions, and even some regions that cannot be distributed dynamically at all. These regions either feature amounts of work that are too small, or work for which statically identifiable dependencies require not distributing work among multiple accelerators to run in parallel. The benefit of adding additional accelerators correlates heavily with the data parallelism inherent in a given implementation.

5.2. GAM Improvement Results

Figures 12 and 13 show a comparison between the performance and energy reduction when comparing a system featuring GAM+ to a system featuring GAM. GAM results are restricted to a single accelerator due to the need of a software-based acceleration solution. GAM is not capable of dynamically taking advantage of an arbitrary number of accelerators, which is a key distinguishing feature of GAM+.

Most results in this area are unsurprising. Energy benefit again correlates directly with performance benefit as expected. In most cases, GAM+ and GAM perform very similarly when a single accelerator is provided to both systems. There are, however, a few interesting cases in these results, particularly Texture Synthesis, and a number of the medical imaging benchmarks.

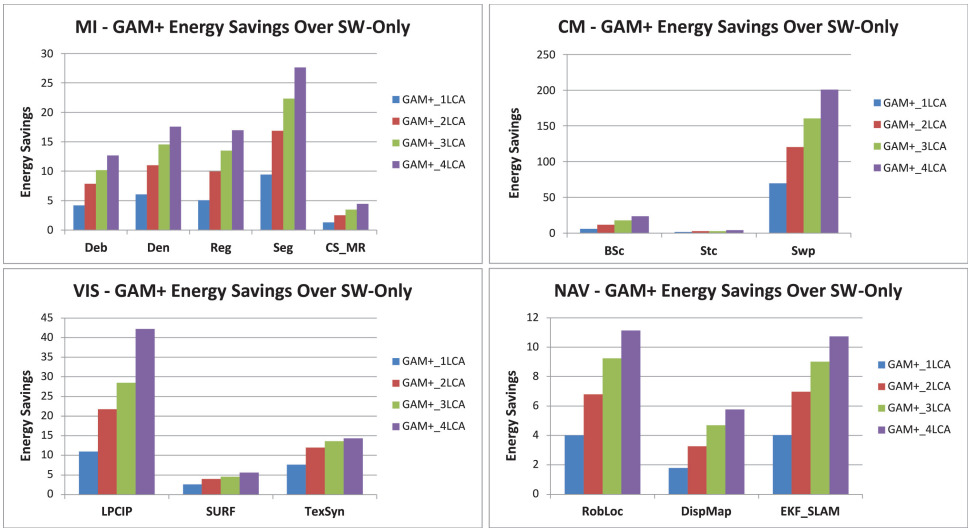

Fig. 11. All domains - energy improvements of GAM+ over SW-only versions.

When arbitrating with GAM, a CPU must undergo a fair amount of communication with GAM prior to acquiring an accelerator. There is also an amount of computation involved in deciding whether or not to use an accelerator, estimating runtime, and preparing registering interrupt handlers. Under most every circumstance, this overhead is insignificant due to the expectation that the accelerator being used will be used for a significant length of time relative to the amount of time required to prepare to use the accelerator. In the case of Texture Synthesis, however, the accelerator is used repeatedly for very small jobs. For this reason, the simplification of the process of starting an accelerator when using GAM+ begins to be visible. Even with a single accelerator, as much as 20% of the total execution time observed in Texture synthesis is contributed to this initialization phase. A side-effect of this is that there is relatively little work for GAM+ to distribute, which limits the benefit that can be gained by adding additional accelerators.

Several medical imaging benchmarks also highlight an interesting difference between the two systems discussed in this work. When GAM+ schedules work, it schedules a small portion of the total amount of work, a task group, to an accelerator. It does not assign more work to the accelerator until this amount of work is done. Thus, if the size of a task group is small relative to the total amount of tasks to perform, there is an overhead associated with using GAM+ that comes from repeatedly waiting for the compute pipeline to fill and empty between task group assignments. Medical imaging benchmarks highlight this drawback. While being highly scalable in the presence of many accelerators, systems featuring a single accelerator spend a reasonable amount of time repeatedly waiting for the compute engine of the accelerator to fill and drain.

Throughout our experiments, we configured the task group size to be 8 tasks, which was the minimum size needed to guarantee correctness across all benchmarks we examined. For every number of accelerators and data size, a different task group size is optimal. We chose not to vary the task group size in these experiments for the purpose of simplifying a discussion about our results. Generally though, a smaller task group size increases the overhead associated with allocation, while a larger task group size reduces the capacity of GAM+ to distribute work evenly amongst many accelerators.

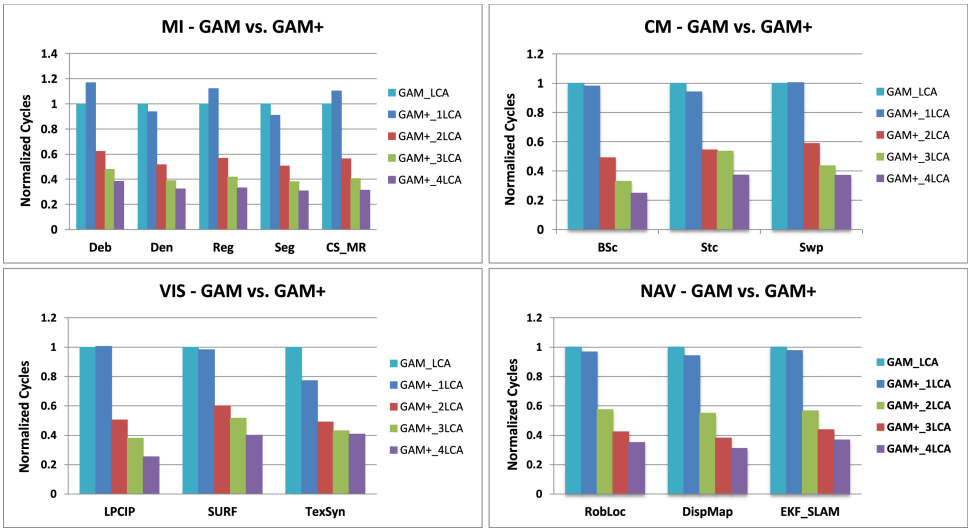

Fig. 12. All domains - performance of GAM vs. GAM+ (normalized to GAM).

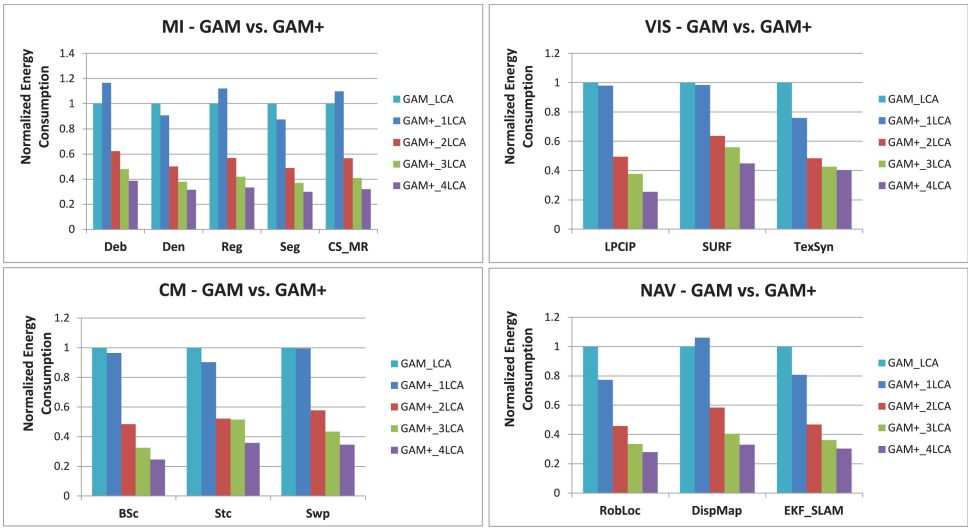

Fig. 13. All domains - energy consumption of GAM vs. GAM+ (normalized to GAM).

5.3. Design Space Exploration Case Study

We chose the medical imaging pipeline shown in Figure 14 for our design space exploration case study. For this case study, we picked a latency-critical workload model (finishing the pipeline within a given time constraint), and an area constraint, both shown in Table VIII. The design space parameters are shown in Table IX. Here we are modifying the number of accelerators (each one independently), the L2 cache size, and associativity, and the off-chip memory bandwidth. We have fixed the number of cores to a fixed value of 2 because most of the functionality of this pipeline is executed on the accelerators. Processing cores serve only to choreograph accelerator actions, and are thus not a critical parameter for this pipeline.

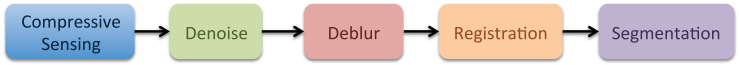

Fig. 14. Medical imaging pipeline used in design space exploration.

Table VIII. Design Space Exploration Constraints

Area	$75mm^2$
Latency	$Compressive\ Sensing\ <\ 300\ Seconds$ (#iterations = 30) $Denoise \rightarrow Deblur \rightarrow Registration \rightarrow Segmentation\ <\ 30\ Seconds$ (#iterations = 4)
Image Size	$512 \times 512 \times 256$

Table IX. Design Space Exploration Parameters

Parameter	Values
# Compressive Sensing LCAs	>= 1 (incremented by 1)
# Denoise LCAs	>= 1 (incremented by 1)
# Deblur LCAs	>= 1 (incremented by 1)
# Registration LCAs	>= 1 (incremented by 1)
# Segmentation LCAs	>= 1 (incremented by 1)
L2 Size	> 0 (incremented by 1MB)
L2 Associativity	>= 1 (in powers of 2)
Off-chip Memory Bandwidth	10, 20, 30 GB per Second

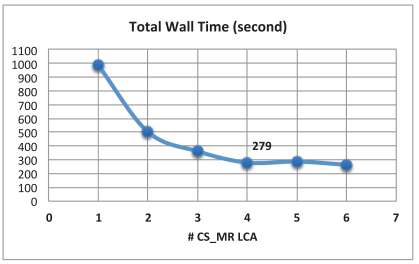

Fig. 15. Compressive sensing timing results.

Figure 16 shows our design space exploration methodology. We start with a minimum configuration (1 of each LCA type, minimum size L2, and minimum off-chip bandwidth). If it does not meet the latency and area constraints, we create the immediate neighbors for that configuration and select the one which gives us the largest benefit computed as $\Delta T/\Delta A$, where ΔT is performance improvement and δA is area increase. Here the area is computed using the values shown in Table VII and performance improvement is calculated via simulation.

In order to meet the latency constraint for Compressive Sensing (CS_MR), we need to have at least 4 CS LCAs as shown in Figure 15. These 4 CS LCAs (and the needed SPM banks) together with the 2 cores take up a constant area of the chip (here 67.3 mm^2). Figure 17 shows our design space exploration results for the medical imaging pipeline shown in Figure 14, given the constraints in Table VIII and parameters in Table IX. The start configuration has one LCA for each LCA type, 4MB of L2 cache, and 10GB of memory bandwidth. The final configuration that can meet both the area and latency constraints has 2 Denoise LCAs, 3 Deblur LCAs, 2 Registration LCAs, 1 Segmentation LCA, 4MB of L2 cache, and 30GB of off-chip bandwidth. Table X shows the final area for the chip.

5.4. Virtualization

In order to show the effectiveness of our proposed virtualization approach, we have virtualized a 2D and a 3D Fast Fourier Transform (FFT) [Cooley and Tukey 1965] computation using a 1D FFT engine. Here we compare our approach with the physical/monolithic implementation of 2D and 3D FFT engines. Figure 18 shows the result of virtualizing a 256×256 2D FFT using FFT 1D 64 points. Figure 19 shows the result of virtualizing a 64×64×64 3D FFT using FFT 1D 64 points. What is shown in these results is the ratio of time to do the computation on the virtual LCA over the time to do the computation on the physical LCA. For the 2D case, using 4 1D FFT we can get the performance of the physical 2D FFT, and for the 3D case, using 5 1D FFT we can get the performance of the physical 3D FFT.

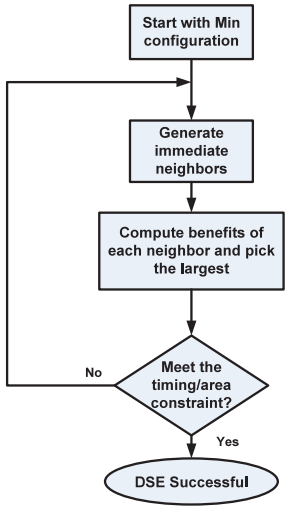

Fig. 16. Methodology for design space exploration.

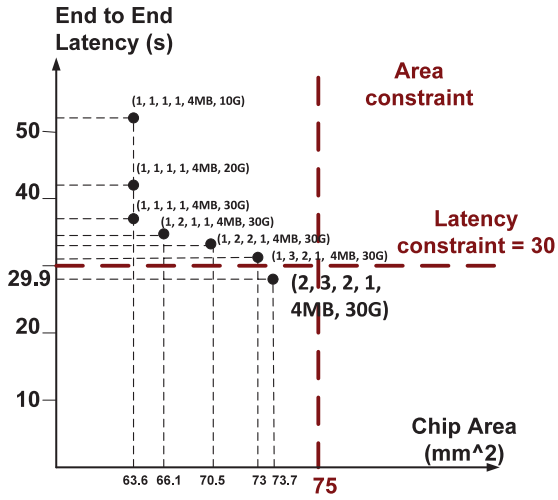

Fig. 17. Results of design space exploration for medical imaging pipeline.

Table X. Final Chip Area (mm^2) Based on Table VII

Cores	L2 Cache	Compressive Sensing	Denoise	Deblur	Segmentation	Registration	SPM banks	Total area
2×10.8 (scaled to 32nm) [Wikipedia 2001]	9.5 [HP-Labs 2008]	$i4.66$	2×0.496	3×2.01	1×0.688	2×3.85	596×0.037 [HP-Labs 2008]	73.7

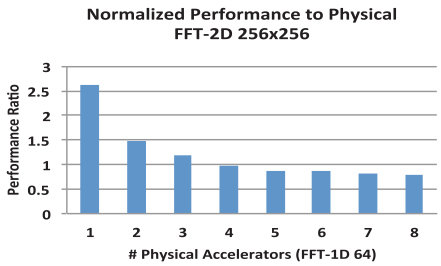

Fig. 18. FFT-2D virtualization.

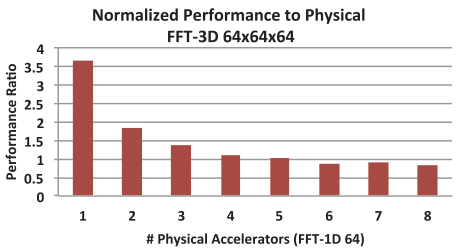

Fig. 19. FFT-3D virtualization.

6. CONCLUSION

We have discussed hardware architectural support for accelerator-rich CMPs. This was motivated by our belief that future supercomputers, especially green supercomputers, will improve their performance and power efficiency through extensive use of accelerators. With this work we have presented a hardware resource management scheme for sharing of loosely coupled accelerators and arbitration of multiple requesting cores. We have also presented a mechanism for accelerator virtualization, allowing multiple accelerators to efficiently compose a larger virtual accelerator, in addition to collaborating as multiple copies of a simple accelerator. All of this work is supported by a fully automated simulation tool-chain for both accelerator generation and management. Our results have not only demonstrated significant improvements over a software

implementation, they have shown additional benefits that result from enhanced load balancing and simplification of the communication for accelerator arbitration, thereby advocating the use of efficient hardware-based techniques for managing and interfacing with loosely coupled accelerators.

REFERENCES

D. Bouris, A. Nikitakis, and I. Papaefstathiou. 2010. Fast and efficient fpga-based feature detection employing the surf algorithm. In *Proceedings of the 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'10).* 3–10.

N. Clark, A. Hormati, and S. Mahlke. 2008. VEAL: Virtualized execution accelerator for loops. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA'08).* 389–400.

Convey Computer. 2008. Convey computer. http://conveycomputer.com/.

J. Cong. 2009. FPGA-based hardware acceleration of lithographic aerial image simulation. *ACM Trans. Reconfig. Technol. Syst.* 3, 1–29.

J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. 2011. High-level synthesis for fpgas: From prototyping to deployment. *IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst.* 30, 4, 473–491.

J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, and G. Reinman. 2012. Architecture support for accelerator-rich cmps. In *Proceedings of the 49th Annual Design Automation Conference (DAC'12).*

J. W. Cooley and J. W. Tukey. 1965. An algorithm for the machine calculation of complex fourier series. *Math. Comput.* 19, 297–301.

M. Frigo and S. G. Johnson. 2005. The design and implementation of fftw3. *Proc. IEEE* 93, 2, 216–231.

P. Garcia and K. Compton. 2008. Kernel sharing on reconfigurable multiprocessor systems. In *Proceedings of the International Conference on ICECE Technology (FPT'08).* 225–232.

V. Govindaraju, C. H. Ho, and K. Sankaralingam. 2011. Dynamically specialized datapaths for energy efficient computing. In *Proceedings of the 17th International Symposium on High Performance Computer Architecture (HPCA'11).* 503–514.

J. R. Hauser and J. Wawrzynek. 1997. Garp: A mips processor with a reconfigurable coprocessor. In *Proceedings of the 5th Annual IEEE Field-Programmable Custom Computing Machines (FCCM'97).* 12–21.

ITRS. 2011. ITRS system drivers. http://www.itrs.net/Links/2011ITRS/2011Chapters/2011SysDrivers.pdf.

W. Jiang and V. K. Prasanna. 2009. Large-scale wire-speed packet classification on fpgas. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'09).* 219–228.

C. Johnson, D. H. Allen, J. Brown, S. Vanderwiel, R. Hoover, et al. 2010. A wire-speed power tm processor: 2.3ghz 45nm soi with 16 cores and 64 threads. In *Proceedings of the IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC'10).* 104–105.

T. Johnson and U. Nawathe. 2007. An 8-core, 64-thread, 64-bit power efficient sparc soc (niagara2). In *Proceedings of the International Symposium on Physical Design (ISPD'07).* 2–2.

S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09).* 469–480.

P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, et al. 2002. Simics: A full system simulation platform. *Comput.* 35, 2, 50–58.

M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, et al. 2005. Multifacet's general execution-driven multiprocessor simulator toolset. *SIGARCH Comput. Archit. News* 33, 4, 92–99.

Nallatech. 2011. Nallatech fsb - Development systems. http://www.nallatech.com/Intel-Xeon-FSB-Socket-Fillers/fsb-development-systems.html.

H. Park, Y. Park, and S. Mahlke. 2009. Polymorphic pipeline array: A flexible multicore accelerator with virtualized execution for mobile multimedia application. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09).* 370–380.

M. Puschel, J. M. F. Moura, J. R. Johnson, D. Padua, M. M. Veloso, et al. 2005. SPIRAL: Code generation for dsp transforms. *Proc. IEEE* 2, 232–275.

A. Ramirez, F. Cabarcas, B. Juurlink, M. A. Mesa, F. Sanchez, et al. 2010. The sarc architecture. *IEEE Micro.* 30, 5, 16–29.

P. Schaumont and I. Verbauwhede. 2003. Domain-specific codesign for embedded security. *Comput.* 36, 4, 68–74.

L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, et al. 2009. Larrabee: A many-core x86 architecture for visual computing. *IEEE Micro.* 29, 10–21.

P. M. Stillwell, V. Chadha, O. Tickoo, S. Zhang, R., Illikkal et al. 2009. HiPPAI: High performance portable accelerator interface for socs. In *Proceedings of the International Conference on High Performance Computing (HiPC'09)*. 109–118.

N. Sun and C.-C. Lin. 2007. Using the cryptographic accelerators in the ultrasparc t1 and t2 processors. Sun BluePrints Online, november. http://www.oracle.com/technetwork/systems/archive/a11-014-crypto-accelerators-439765.pdf.

Synopsys. 2013. Synopsys design compiler. http://www.synopsys.com/Tools/Pages/default.aspx.

G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, et al. 2010. Conservation cores: Reducing the energy of mature computations. In *Proceedings of the 15th International Conference edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS'10)*. 205–218.

P. H. Wang, J. D. Collins, G. N. Chinya, H. Jiang, X. Tian, et al. 2007. EXOCHI: Architecture and programming environment for a heterogeneous multi-core multithreaded system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*. 156–166.

Received February 2013; revised November 2013; accepted December 2013