Architecture Support for Accelerator-Rich CMPs

Jason Cong
UCLA, CS and Center for Domain Specific Computing
cong@cs.ucla.edu

Mohammad Ali Ghodrat
UCLA, CS and Center for Domain Specific Computing
ghodrat@cs.ucla.edu

Michael Gill
UCLA, CS and Center for Domain Specific Computing
mgill@cs.ucla.edu

Beayna Grigorian
UCLA, CS and Center for Domain Specific Computing
bgrigori@cs.ucla.edu

Glenn Reinman
UCLA, CS and Center for Domain Specific Computing
reinman@cs.ucla.edu

ABSTRACT

This work discusses a hardware architectural support for accelerator-rich CMPs (ARC). First, we present a hardware resource management scheme for accelerator sharing. This scheme supports sharing and arbitration of multiple cores for a common set of accelerators, and it uses a hardware-based arbitration mechanism to provide feedback to cores to indicate the wait time before a particular resource becomes available. Second, we propose a light-weight interrupt system to reduce the OS overhead of handling interrupts which occur frequently in an accelerator-rich platform. Third, we propose architectural support that allows us to compose a larger *virtual* accelerator out of multiple smaller accelerators. We have also implemented a complete simulation tool-chain to verify our ARC architecture. Experimental results show significant performance (on average 51X) and energy improvement (on average 17X) compared to approaches using OS-based accelerator management.

Categories and Subject Descriptors

C.1 [**PROCESSOR ARCHITECTURES**]: C.1.3—*Heterogeneous systems*

General Terms

Design

Keywords

Chip multiprocessor, Hardware Accelerators, Accelerator Virtualization, Accelerator Sharing

1. INTRODUCTION

Power-efficiency has become one of the primary design goals in the many-core era. While ASIC/FPGA 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 [24]. 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 [24]. A recent industry trend to address this is the use of on-chip accelerators in many-core designs [16][25][17]. According to an ITRS prediction [2], 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 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 [28]. 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 *un-utilizable* 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., [17][15]); and 2) loosely coupled accelerators (e.g., [3]) where the accelerator is a distinct entity attached to the network-on-chip (NoC), which can be shared among multiple cores. This paper focuses on the efficient use of loosely coupled accelerators, 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 – but this does require 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 friendly for application programs to use. Our paper provides the following contributions:

- An accelerator allocation protocol to avoid OS overhead in scheduling tasks to shared, loosely coupled accelerators
- An approach to accelerator composability that allows multiple accelerators to work collaboratively as a single complex virtual accelerator that is transparent to program authors
- A fully automated simulation tool-chain to support accelerator generation and management

An early version of this work without virtualization, light-weight interrupt and visual-navigation study was presented in [9]. The rest of the paper 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. Section 4 discuss our experimental results which support our proposed methods.

2. RELATED WORK

There is a large amount of work that implements an application-specific coprocessor or accelerator through either ASIC or FPGA [4] [7]. These works mostly consider a single accelerator dedicated to a single application. Convey [1] and Nallatech [3], target reconfigurable computing in which customized accelerators are off-chip from the processors, unlike our work which target CMP architectures with on-chip accelerators. Some previous work considered on-chip integration of accelerators. Garp [14], UltraSPARC T2 [17], Intel's Larrabee [25] and IBM's WSP processor [16] are examples of this. Most of these platforms (except WSP) are tightly coupled with processor cores (or core-clusters). Our paper 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 [13]. 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 [29], SARC [23], and HiPPAI [26]. Similar to our work, EXOCHI's focus is on a heterogeneous non-uniform 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.

There are some related work in accelerator virtualization, namely VEAL [6] and PPA [21]. VEAL [6] 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 limits to nested loops, while in our approach we seek any accelerator such that its composition can be described by some set of rules. PPA [21] 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.

3. ARCHITECTURE SUPPORT OF ARC

In an accelerator-rich platform, one main issue is how to increase the utilization of accelerators and also how to make them reusable between multiple applications. Our approach uses several techniques, namely accelerator sharing and accelerator virtualization. In the following subsections we first discuss the motivation for our work and then show how we efficiently implement these techniques.

3.1 Motivation

In a typical heterogeneous system which uses accelerators, when a core wants to access an accelerator, it does that by using an accelerator driver (OS call) [13] [27]. Using the Simics/GEMS simulation [19] [20] platform to model a system consisting of UltraSPARC III-i processors running Solaris 10, we measured the delay for different system call operations. These results are shown in Table 1 (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. The first motivation for our work (efficient sharing) is to minimize this overhead when there are many accelerators. The second motivation for our work is to increase the utilization of these accelerators by creating new or larger accelerators through composition and chaining.

Table 1: 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

3.2 Microarchitecture of ARC

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 arbitration.

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 2. 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 below.

1. The core requests an enumeration of all accelerators it may

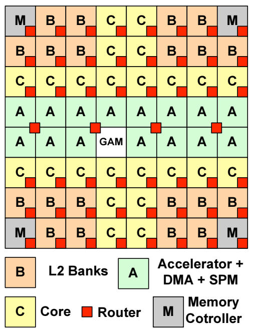


Figure 1: **Overall architecture of ARC**

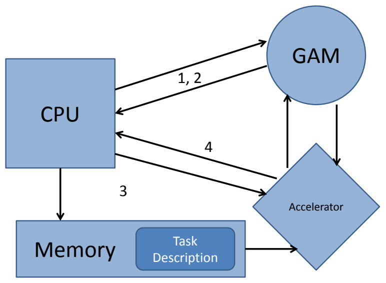


Figure 2: **Communication between core, GAM, and accelerator**

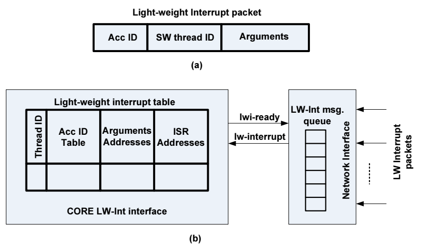


Figure 3: **Light-weight interrupt support**

Table 2: 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*.

potentially need from the GAM (*lcacc-req*). The GAM responds with a list of accelerator IDs and associated estimated wait times.

2. The core sends a sequences 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 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*).

3.3 Light-weight 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 light-weight interrupts. ARC light-weight 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 1 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: 1) GAM responses 2) TLB misses 3) notification that the accelerator has finished working. 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.

Figure 3 shows the microarchitecture components added to the cores in ARC in order to support the light-weight interrupt. An interrupt is sent via an interrupt packet (shown in Figure 3-a) through the NoC to the core requested 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 microarchitecture components added to support light-weight interrupt are listed below:

Table 3: Instructions to handle light-weight 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.

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. Light-weight interrupt interface in the core. This is responsible for: 1) receiving the interrupt from the interrupt controller, 2) providing a software interface to setup 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 no other light-weight interrupt will be serviced, while servicing another light-weight interrupt. 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 light-weight user-level interrupts, we introduce a set of instructions to enable user code to handle interrupts. These instructions are described in Table 3. *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

In this work, we assume an accelerator will be used to process a relatively large amount of data. The initial overhead associated with acquiring permissions 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 3.6). 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.

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. It does this by sending a light-weight interrupt to the controlling core when a TLB miss occurs with the address that caused the TLB miss. 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.

3.5 Sharing accelerators

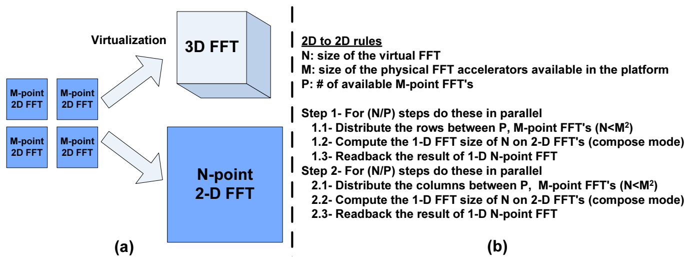
When accelerators are shared among all the on-chip cores, it is possible for there to be several cores competing for the same accelerator. Even in architectures with large numbers of accelerators, there may be a limited number of one particular type of accelerator that is suddenly in high demand. In this situation, some of these cores 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 paper 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 run time for each job in the waiting list (Section 3.5.2).

3.5.1 Accelerator run-time estimation (by the core)

The execution time of a certain job on an accelerator is data-dependent. For most of our examples, we found that a low order polynomial regression model was sufficient to estimate execution time. The regression model is provided by the accelerator application programming interface (more info in [11]).

3.5.2 Wait-time estimation algorithm (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. The estimated waiting time can be simply derived by summing the expected execution time of all jobs in the waiting list for an accelerator. These tasks are issued in a first-come-first-serve (FCFS) order. This is simple, and is practical for a hardware implementation.



2D to 2D rules
N: size of the virtual FFT
M: size of the physical FFT accelerators available in the platform
P: # of available M-point FFT's

Step 1- For (N/P) steps do these in parallel
  1.1- Distribute the rows between P, M-point FFT's ($N<M^2$)
  1.2- Compute the 1-D FFT size of N on 2-D FFT's (compose mode)
  1.3- Readback the result of 1-D N-point FFT
Step 2- For (N/P) steps do these in parallel
  2.1- Distribute the columns between P, M-point FFT's ($N<M^2$)
  2.2- Compute the 1-D FFT size of N on 2-D FFT's (compose mode)
  2.3- Readback the result of 1-D N-point FFT

Figure 4: An example of accelerator composition

3.6 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.

3.6.1 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, i.e., 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.

3.6.2 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 4(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 4(b)), similar in style to the approach presented in [22]. These small problems would then be solved directly by hardware. These rules describe two things: 1) computation that must be performed by accelerators capable of solving sub-problem 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 run-time. Figure 5 describes the process of invoking a virtual accelerator from within the application binary. When an accelerator is called, 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

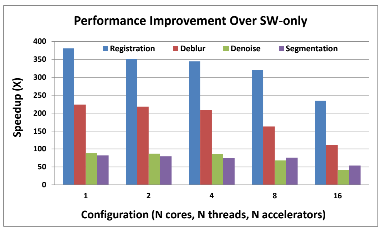


Figure 6: MI - Speedup over SW-only

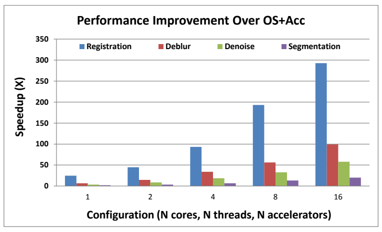


Figure 7: MI - Speedup over OS+Acc

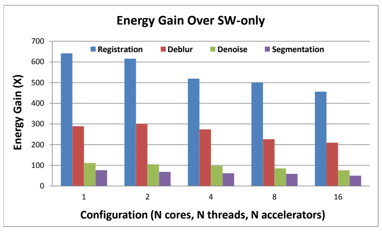


Figure 8: MI - Energy gain over SW-only

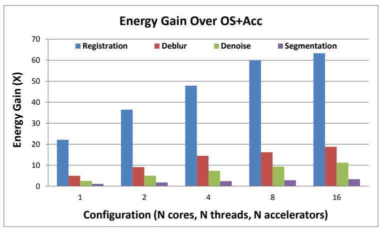


Figure 9: MI - Energy gain over OS+Acc

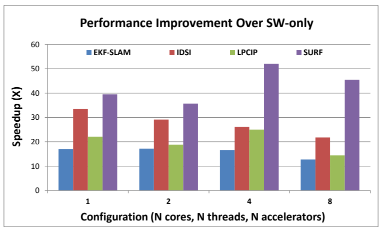


Figure 10: VN - Speedup over SW-only

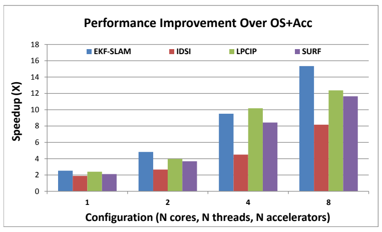


Figure 11: VN - Speedup over OS+Acc

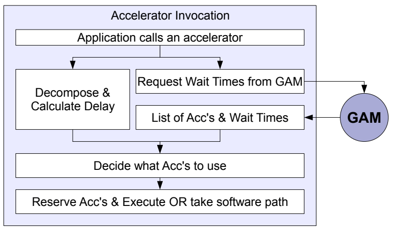

Accelerator Invocation
Application calls an accelerator
Decompose & Calculate Delay
Request Wait Times from GAM
List of Acc's & Wait Times
GAM
Decide what Acc's to use
Reserve Acc's & Execute OR take software path

Figure 5: Accelerator composition steps

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.

We will show more details on programming interface in ARC in Supplemental Section 8.1. More info on accelerator extraction methodology can be found in [11].

4. EXPERIMENTAL RESULTS

To illustrate the effectiveness of our ARC platform, we evaluate a number of compute intensive benchmarks, primarily from the medical imaging(MI) and computer vision and navigation(VN) domains. More information on our benchmark can be found in [11] and Supplemental Section 8.2.2. Our experiments were conducted using a heavily modified version of the Simics and GEMS [19] [20] simulation platform. More information about our simulation platform can be found in Supplemental Section 8.2. Additional experimental results not presented here can also be found in [11].

We used the following schemes for ARC evaluation:

- **Original benchmark (SW-only)**: The baseline for the experiments is the execution of these multithreaded benchmarks on a multiprocessor (one thread per processor).
- **Accelerators + OS management (OS+Acc)**: This is a system which has accelerators managed by OS drivers.
- **Accelerators + HW management (ARC)**: This is a system which features all enhancements discussed thus far, including hardware resource arbitration managed by the GAM.

We show the simulation configuration using Cc-Tt-Aa-Dd mnemonic. Here "C" is the number of cores, "T" is the number of threads, "A" is the number of replicates of each accelerator needed by a benchmark, and "D" is data size. For example, a benchmark featuring 4 cores, 2 threads, 1 replicate of each accelerator, and an argument that is 64-cubes of data would be described as 4c-2t-1a-64d. For MI benchmarks since data is cubic in form, "D" shows a cube of $D \times D \times D$ data elements for each argument. For VN benchmarks data is linear, thus "D" shows the absolute data size. Next the results for baseline speedup and energy improvement are discussed.

4.1 Speedup and energy improvements

Figures 6, 10, 8, and 12 shows the speedup and energy gain result for the ARC base configuration (Nc-Nt-Na) compared to running the software-only version of the benchmark on the same number of processors, threads, and data size. The highest speedup is for *Registration* (485X for 1c-1t-1a-32d case) and the lowest is for EKF-SLAM (13X for 16p-16t-16a case). The best energy gain is for registration with 641X improvement. On average we get 241X energy improvement over all the benchmarks and configuration. VN benchmarks are shown benefiting less from acceleration as compared to MI benchmarks due largely to data sizes selected. A study of the impact of data size on accelerator efficiency can be found in [11].

We observe a reduction in speedup as we increase the number of cores and threads. This reduction is attributed to several sources. First, we measure the time from the start of all threads, to the end of the last thread, thus the results shown are the measured time of the longest running thread. Adding more threads increases the likelihood of observing normal fluctuations in run time. Lastly, while we increase the number of cores and accelerators, we do not correspondingly increase network resources, memory bandwidth, or cache capacity. As a result, increasing the number of cores and threads resulted in additional contention for communication and memory resources. This impacted accelerated cases more than software-only cases because, while the same amount of data is accessed, the accelerated cases access this data over a much shorter time period.

Figures 7 and 11 show the speedup gain ARC achieves com-

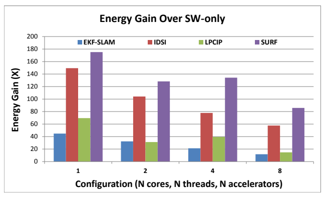


Figure 12: VN - Energy gain over SW-only

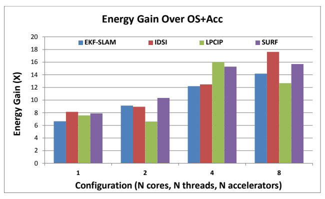


Figure 13: VN - Energy gain over OS+Acc

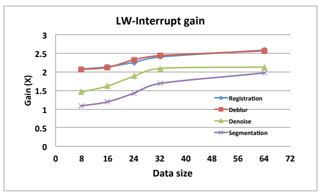


Figure 14: Benefit of using light weight interrupt

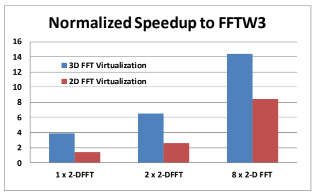


Figure 15: FFT virtualization (2D and 3D)

pared to the OS+Acc. Here, for larger base configurations we see an increase speedup compare to OS managed systems. The reasons for this are: 1) by increasing number of threads and processors, the OS management overhead (thread context switching, TLB services, ...) increases, and 2) for larger configurations, the number of interrupts are also increasing, which makes our system perform better due to the use of light-weight interrupt in the place of the OS interrupts. Figures 9 and 13 also show the energy improvement of ARC over the OS+Acc case. Here by making configurations larger, we see a better energy gain over OS+Acc system. Again registration performs best with 63X. On average we get 17X energy gain over OS+Acc case.

4.2 Accelerator virtualization results

Figure 15 shows the result of virtualizing a 512x512 2D FFT and a 128x128x8 3D FFT on multiple 128x128 2D FFTs. The SW case is compared to having 1, 2, and 8 copies of 128x128 2D accelerator on the chip (8 FFT is based on assigning a maximum 5% of the chip area to FFT). The SW case is the result of running FFTW3 [12]. In the best case for 3D-FFT we obtained 14.4X speedup and for 2D-FFT we obtained 8.4X speedup.

4.3 Light-weight interrupt benefit

To measure the benefit of light-weight interrupts, we examined a platform lacking light-weight interrupts to compare our ARC platform against a system that relies instead on OS handling of interrupts. Figure 14 shows the speedup measured over a platform lacking light-weight interrupts. ARC is up to 2.5X faster than an otherwise identical system that lacks light-weight interrupts. The larger the data size, the more interrupts are generated, so the benefits of ARC increases as the data size grows.

4.4 Accelerator sharing results

Run-time estimation was calculated using a simple regression model based on profiled runs. Additional details regarding this regression model can be found in [11]. Wait-time estimation was based on the accumulated run-time estimates. Our results shows that the estimated error ranges from $< 1\%$ to 6% of execution times on accelerators, which is sufficiently predictable for this to be a very practical approach.

5. CONCLUSION AND FUTURE WORK

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. First, we presented a hardware resource management scheme for sharing of accelerators and arbitration of multiple requesting cores. Second, we presented a mechanism that allows us to efficiently compose a larger virtual accelerator out of multiple smaller accelerators. Our results showed large performance and energy efficiency improvement over a software implementation, and also using OS-based accelerator management, with minimal hardware overhead

6. ACKNOWLEDGEMENTS

This research is partially supported by the Center for Domain-Specific Computing (CDSC) funded by the NSF Expedition in Computing Award CCF-0926127, GSRC under contract 2009-TJ-1984 and NSF Graduate Research Fellowship Grant # DGE-0707424.

7. REFERENCES

[1] Convey computer. `http://conveycomputer.com/`.
[2] ITRS 2007 system drivers. `http://www.itrs.net/`.
[3] Nallatech FSB - development systems. `http://www.nallatech.com/Intel-Xeon-FSB-Socket-Fillers/fsb-development-systems.html`.
[4] D. Bouris et al. Fast and efficient FPGA-based feature detection employing the SURF algorithm. FCCM '10, pages 3–10.
[5] A. Bui et al. Platform characterization for domain-specific computing. In *ASPDAC*, 2012.
[6] N. Clark, , et al. VEAL: Virtualized execution accelerator for loops. ISCA '08, pages 389–400.
[7] J. Cong et al. FPGA-based hardware acceleration of lithographic aerial image simulation. *ACM Trans. Reconf. Technol. Syst.*, pages 1–29, 2009.
[8] J. Cong et al. Accelerating vision and navigation applications on a customizable platform. In *ASAP*, 2011.
[9] J. Cong et al. AXR-CMP: Architecture support in accelerator-rich CMPs. 2nd Workshop on SoC Architecture, Accelerators and Workloads, February 2011.
[10] J. Cong et al. High-level synthesis for FPGAs: From prototyping to deployment. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(4):473 –491, April 2011.
[11] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, and G. Reinman. UCLA computer science department technical report #120008.
[12] M. Frigo et al. The design and implementation of FFTW3. *Proc. of the IEEE*, 93(2):216–231, 2005.
[13] P. Garcia et al. Kernel sharing on reconfigurable multiprocessor systems. FPT 2008, pages 225 –232.
[14] J. Hauser et al. Garp: a mips processor with a reconfigurable coprocessor. FCCM'97, pages 12 –21.
[15] W. Jiang et al. Large-scale wire-speed packet classification on FPGAs. FPGA '09, pages 219–228.
[16] C. Johnson et al. A wire-speed power$^{\mathrm{TM}}$processor: 2.3ghz 45nm soi with 16 cores and 64 threads. ISSCC'10, pages 104 –105.
[17] T. Johnson et al. An 8-core, 64-thread, 64-bit power efficient sparc soc (niagara2). ISPD '07, pages 2–2.
[18] S. Li et al. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. MICRO 42, 2009.
[19] P. S. Magnusson et al. Simics: A full system simulation platform. *Computer*, 35:50–58, 2002.
[20] M. M. K. Martin et al. Multifacet's general execution-driven multiprocessor simulator toolset. *SIGARCH Comput. Archit. News*, 33, 2005.
[21] H. Park et al. Polymorphic pipeline array:a flexible multicore accelerator with virtualized execution for mobile multimedia application. MICRO, 2009.
[22] M. Puschel et al. Spiral: Code generation for dsp transforms. *Proc. of the IEEE*, (2):232 –275, 2005.
[23] A. Ramirez et al. The SARC architecture. *Micro, IEEE*, 30(5):16 –29, Sep 2010.
[24] P. Schaumont et al. Domain-specific codesign for embedded security. *Computer*, 36:68–74, 2003.
[25] L. Seiler et al. Larrabee: A many-core x86 arch. for visual computing. *IEEE Micro*, 29:10–21, 2009.
[26] P. Stillwell et al. HiPPAI: High performance portable accelerator interface for SoCs. HiPC 2009.
[27] N. Sun et al. Using the cryptographic accelerators in the ultrasparc t1 and t2 processors. *Sun BluePrints Online*, 2007.
[28] G. Venkatesh et al. Conservation cores: reducing the energy of mature computations. ASPLOS '10.
[29] P. H. Wang et al. EXOCHI: architecture and programming environment for a heterogeneous multi-core multithreaded system. PLDI '07.

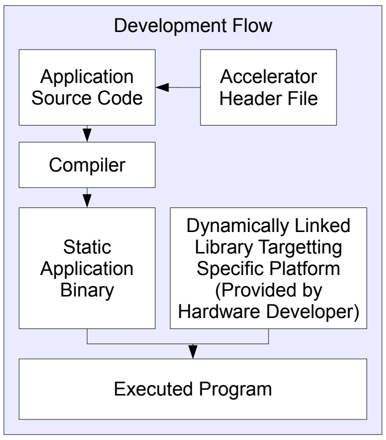

Development Flow
Application Source Code
Accelerator Header File
Compiler
Static Application Binary
Dynamically Linked Library Targetting Specific Platform (Provided by Hardware Developer)
Executed Program

Figure 16: **ARC development flow**

8. SUPPLEMENTAL

8.1 Programming interface to ARC

The Application Programming Interface (API) involved in using accelerators is presented in Figure 16. 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.

8.2 EVALUATION METHODOLOGY

8.2.1 Simulation 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.

With this toolchain, 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 these modules into VHDL, and synthesizing these modules to extract timing and energy information. This process yields a module that plugs into our cycle-accurate simulation infrastructure to model this hardware unit, and coordinates the execution of this selected function in a pipelined fashion.

Once we select the functions we want to accelerate, typically encompassing the kernel of the benchmark, we procedurally generate a program segment to use these accelerators. We described communication between accelerators in a simple data-flow language that we use to generate C source code. These program segments together make up the platform specific DLL mentioned above. This code is responsible for coordinating interactions between accelerators, registering and handling interrupts, managing task descriptions and accelerator resources, and dealing with synchronization between accelerators and the CPU. Figure 17 illustrates the work flow described here. The AutoPilot [10] behavioral synthesis tool

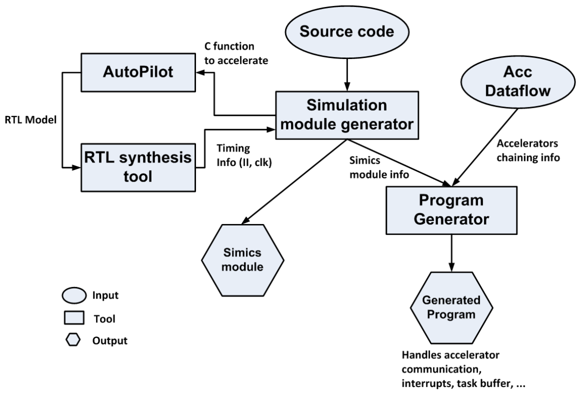


Figure 17: Process used to generate simulation structures and Accelerator using programs

is used to synthesize the C modules into ASIC.

8.2.2 Benchmarks

To illustrate the effectiveness of our ARC platform, we evaluate a number of compute intensive benchmarks from both the medical imaging domain as well as the computer vision and navigation domain. Information on medical imaging domain can be found in [5]. Information on computer vision and navigation domain can be found in [8].

8.2.3 Simulation Platform

Our experiments were conducted using a heavily modified version of the Simics and GEMS [19] [20] simulation platform. The machine we modeled was based on a multicore system consisting of a mix of Ultrasparc III processors and accelerators. In order to create a fair comparison between machines of different configurations, we maintained a fixed cache and network configuration. Our network topology was a mesh modeled on a system normally used to support 32 processors. These nodes were then configured to either be processors, accelerators, or empty sockets. We featured a per-processor split L1 cache, and a distributed L2 spread across all nodes that relied on a directory based coherence protocol. Table 4 shows the machine configurations which is modeled in simulations.

8.2.4 Area/Timing/Power Measurements

The AutoPilot [10] behavioral synthesis in combination with the Synopsys design compiler was used to synthesize the C modules into ASIC (using 32nm ASIC library from Synopsys). The timing information produced by the synthesis process was back-annotated to our accelerator modules to model cycle accurate accelerators. For computing energy we used power reports from Synopsys for accelerators and McPAT [18] for CPU power. Table 5 shows the synthesis results for the accelerators in our selected benchmarks together with the GAM and DMA-controller.

Table 4: **Simics/GEMS configuration**

CPU	Ultra-SPARC III-i @ 2.0GHz
Number of cores	1, 2, 4, 8, 16
Coherence protocol	MSI_MOSI_CMP_directory
L1 cache	32 KB, 4 way set-associative
L2 cache	8 MB, 8-way set-associative
Memory latency	1000 cycles
Network topology	Mesh
Operating System	Solaris10

Table 5: **Synthesis results**

	Deblur	Registration	Denoise	Segmentation	GAM	DMA-C
Clock(ns)	2	2	2	2	1	1
Area (μm^2)	2013228	3853095	496908	688298	12270	10071
Power (mW)	98.28	256.3	57.69	80.93	2.64	0.59