University of California
Los Angeles

Enabling Accelerator Centric Computing

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Michael Anthony Gill

2015

Abstract of the Dissertation

Enabling Accelerator Centric Computing

by

Michael Anthony Gill
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2015
Professor Glenn D. Reinman, Chair

With power limitations imposing hard bounds on the amount of a chip that can be powered simultaneously, but advances in manufacturing technologies continuing to pay dividends in terms of feature density [1], together leading to the presence of dark silicon [2], it becomes clear that continued advances in performance will come in the form of energy efficiency and customization rather than scaling processor count and cache size. This observation is the basis for the argument that accelerators, highly customized logic blocks that perform a particular task with both high performance and energy efficiency, are going to become increasingly relevant in future processors. It is predicted that the number of these accelerators will exceed 1500 by 2022 [1].

As accelerators become more responsible for shouldering a greater portion of computation, it becomes important to elevate accelerators to be considered a first-class computational primitive, rather than an unusual device that requires extraordinary measures to interact with. Simply having a powerful compute engine in a machine is meaningless if it is impossible to efficiently communicate with it, or if software that uses an accelerator is hard to write, or if interacting with the device involves complicated performance considerations which make it difficult to predict whether any benefit would be had by using the accelerator.

The work described herein attempts to address this issue, and providing architectural extensions that allow for accelerators to become a high performance, highly efficient, and

highly utilized compute elements. The effort comes from two directions: 1) introducing enabling technologies that allow accelerators to be efficiently used by software, and 2) redesigning system components to allow for accelerators to perform well and leverage existing system resources efficiently. This results in accelerator-centric designs, where conventional processing cores act more as choreographers for a communicating network of accelerators as opposed to cores acting as the primary mechanism of performing computation. The intent is to accomplish this in such a way as to place undue burden on application programmers, by introducing accelerators in such a way as to be compiler-friendly.

The dissertation of Michael Anthony Gill is approved.

Alex Bui

Todd Millstein

Jason Cong

Glenn D. Reinman, Committee Chair

University of California, Los Angeles

2015

Table of Contents

List of Figures

List of Tables

Vita

2007	B.S. (Computer Science), California Polytechnic in Pomona.
2010	M.S. (Computer Science), University of California in Los Angeles.
2007	Teaching Assistant, Computer Science Department, CalPoly. Taught networking and parallel/distributed computing.
2011	Teaching Assistant, Computer Science Department, UCLA. Taught Introduction to Computer Science and Introduction to Computer Organization.
2011-2015	Graduate Student Researcher for The Center for Domain Specific Computing (CDSC).
2013-2015	Graduate Student Researcher for The Center for Future Architectures Research (C-FAR).
2012	Intern at Xilinx. Modified compiler and tool chain for targeting experimental platform.
2014	Intern at Intel. Implemented simulation models and compilation tool-chain for hardware accelerators.

Publications

AXR-CMP: Architecture support for accelerator-rich CMPs (SAW '12)

Compilation and Architecture support for custom vector instruction extensions (ASP-

DAC '12)

BiN: A Buffer-in-NUCA for accelerator-rich CMPs (ISLPED '12)

CHARM: A composable heterogeneous accelerator-rich microprocessor (ISLPED '12)

Stream Arbitration: Toward efficient bandwidth utilization for emerging on-chip interconnects (TACO '13)

Composable accelerator-rich microprocessor enhanced for adaptivity and longevity (ISLPED '13)

Architecture support for domain-specific accelerator-rich CMPs (TECS '14)

Accelerator-rich architectures: Opportunities and Progress (DAC'14)

CHAPTER 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 [3]. 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 [3]. A recent industry trend to address this is the use of on-chip accelerators in many-core designs [4–6]. According to an ITRS prediction [1], 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). On-chip accelerators are combined with general-purpose cores in an effort to amortize the cost of the design across many application domains. Accelerators can capture the most commonly executed kernels of one or more application domains. They are relatively simple to design/optimize (compared to the entire application), and the general-purpose cores can be used to handle the rest of the application.

Accelerator-rich architectures also offer a good solution to overcome the *utilization wall* as articulated in the recent study reported in [7]. 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. Moreover, once an accelerator is used, it provides much higher performance/power efficiency, due to its customized implementation, compared to the general-purpose cores.

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. Currently, this overhead comes both in the form of a performance penalty rooted in operating system(OS) and driver interaction, along with logistical problems associated with actually interacting with the accelerator and authoring programs that make use of specialized hardware. Because accelerator designs are not universal, programmers need to write code that explicitly makes use of accelerators, and thus requires refactoring to the design of the program, knowledge of what accelerators are available in a given platform, and restricting a program to execute exclusively to a specific platform.

These issues gave rise to the work discussed in this document. This work focuses on simplifying the use of accelerators, and simplifying their use. While specific compute engine designs are referenced within this work, they are not the focus of it. The focus of this work is resource management, and eliminating barriers that make accelerators attractive in the abstract, but impractical in reality. By analyzing the strengths of hardware and the needs of software, this work aims to design hardware that can be efficiently used by software, instead of designing the fastest possible hardware, and rendering the platform unusable by software. This takes the form of four independent works, which can be briefly introduces as follows:

- ***ARC:*** Resource management and arbitration is conventionally the responsibility of software drivers, and OS calls designed to allow for interaction with drivers. These calls not only introduce their own overhead, but also introduce long-lasting impacts on program performance, such as flushing translation look-aside buffers (TLBs) or local caches as a result of switches between process memory spaces.

ARC introduces the first architectural accelerator resource manager that aims to extend all of the benefits conventionally extended by a driver, while eliminating the performance overhead. ARC pushes the use of accelerators entirely into user-space, and eliminates the interaction of the OS entirely. This work is discussed in Chapter 2

- ***CHARM:*** Large accelerators are highly efficient and feature high performance, but come with two shortcomings: 1) because they are highly specialized, they are often infrequently utilized, and 2) it is intractable to automate the process of finding occurrences of large accelerators in source code, and thus a programmer needs to explicitly invoke a large accelerator. To address these issues, the CHARM architecture breaks accelerators up into small parts, and expresses a complex accelerator as a network of small communicating accelerators. Because these pieces can communicate with one another in an arbitrary way, any given component is capable of being used in many different configurations. For this reason, it becomes possible to achieve high utilization of these small accelerators. To achieve additional performance and utilization, CHARM also introduces hardware-assisted work distribution, thus allowing a great number of accelerators to contribute toward the computation of even simple compute tasks. Additionally, these small accelerators can be found in arbitrary program code, and thus it is possible to compile programs for this platform without requiring specialized knowledge from a programmer. This work is discussed in greater detail in Chapter 3

- ***CAMEL:*** To further improve on the utilization benefits realized in the CHARM architecture, CAMEL introduces an amount of programmable fabric for the purpose of instancing infrequently used accelerators. This allows for ASIC resources to be dedicated to implementing a greater amount of frequently needed accelerators, rather than being dedicated to rarely used accelerators for the purposes of achieving workload coverage. CAMEL also introduces elements facilitating the management of programmable fabric, that allows the system to continue to operate as a cohesive

whole, entirely within user space. CAMEL is discussed in Chapter 5

- ***AIM:*** Nearly all proposals for introducing accelerators to a platform involve extensive modifications to a number of components of a system, including the cpu, memory, operating system, and often the communication mechanisms and protocols between these components. While these are not problems in the context of an academic study, it does make it extremely difficult to actually introduce accelerators in an actual platform, due to the number of industry players and different organizations whose cooperation would be required. The AIM project puts forward a design intended to introduce acceleration in a more down-to-earth way, in a fashion that can be introduced to existing systems without requiring modification of any existing components. This is done while achieving many of the desirable qualities of the previous works as well, such as operating over shared memory. This is done by introducing a new device that seats into an existing systems' DRAM memory interfaces. This also comes with the advantage that it is possible to construct large systems of accelerators, since the accelerator resources would scale relative to the size of the memory system. This work is discussed in Chapter 6

Additionally, my experience with working with accelerator-centric platforms led me to pursue a number of additional works that deviated from my primary interest, and instead focus on performance concerns of accelerators themselves from a system-wide perspective. Two selected works of this type are introduced below:

- ***BiN:*** Some accelerators benefit from large local memory for internal use, and are able to compute more efficiently if they can internalize a portion of memory reuse. If many such accelerators are on a chip, it becomes unreasonable to construct large private memories for each accelerator, due to space constraints. To address this issue, we propose a method of allocating and managing buffers in a non-uniform cache architecture (NUCA). This work, Buffer in NUCA (BiN), aims to allow for compute engines to operate efficiently on larger amounts of data without requiring private buffers. For algorithms whose computational efficiency coorelates well

with the amount of data being operated over at a given time, this can improve performance substantially. BiN is discussed in Chapter 7.

- ***STREAM:*** While designing high performance compute engines is relatively easy, developing mechanisms to deliver necessary data to a compute engine is much more difficult. We examine trends in network on chip (NoC) design and developed a protocol in the context of emerging network technologies. We evaluated radio frequency interconnection (RF-I) networks, and developed a protocol named STREAM that enables efficient exploitation of the enormous bandwidth potential of new network technology. This work is further discussed in Chapter 8.

CHAPTER 2

Architecture Support for Accelerator-Rich CMPs

Conventional accelerators, such as GPUs or other dedicated PCI devices, conventionally use a set of drivers to allow user software to interact with hardware. This driver primarily provides two services: 1) a portal through which software can communicate with a device, and 2) an opportunity for an operating system to ensure some measure of safety and process isolation. A driver however is a complicated and heavy bit of software, and comes with many performance penalties, as will be shown. In an accelerator-rich system, these penalties constitute a non-trivial cost, and complicates the performance expectations that a programmer may have of a presumably higher performing device.

This work, entitle Accelerator-Rich CMPs(ARC) [8], is the first work to argue for addressing these performance penalties with a dedicated hardware resource manager, and eliminating the concept of a device driver entirely. This work was intended to elevate an accelerator to a first-order primitive, providing hardware primitives that allow for an accelerator to operate directly over shared memory in process space, without involving the operating system or any other auxiliary software abstractions. This results in accelerators being usable for small tasks, and managed as an as-needed set of compute engines that software can tap into, and simplifies the performance model associated with using an accelerator.

2.1 Microarchitecture of ARC

Figure 2.1 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

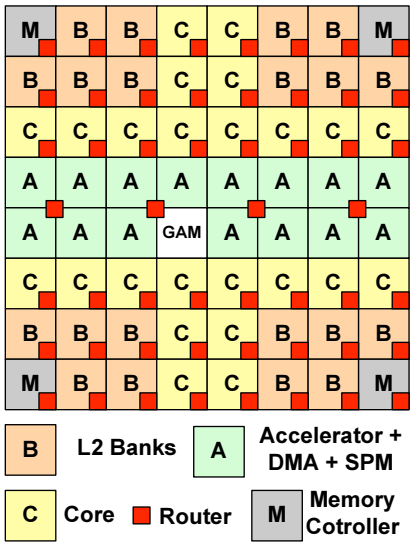


Figure 2.1: Overall architecture of ARC

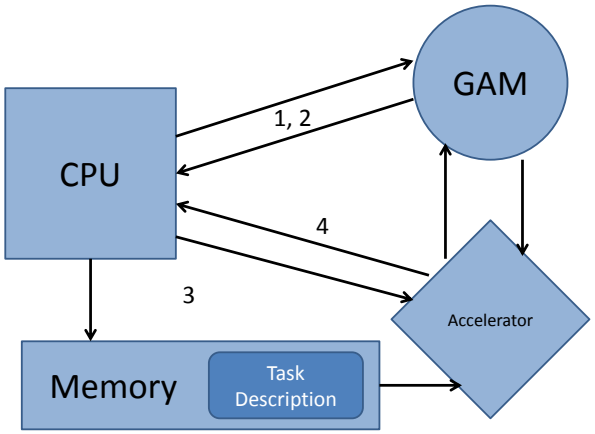


Figure 2.2: Communication between core, GAM, and accelerator

between multiple accelerators. All of the mentioned components are connected by the NoC. Accelerator nodes include a dedicated DMA controller (DMA-C) and scratchpad 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.

2.1.1 Instruction Set Extension

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 2.1. 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

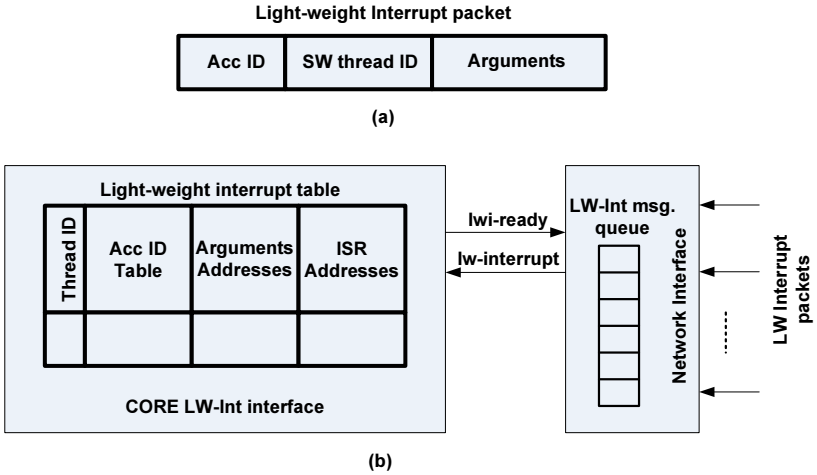


Figure 2.3: Light-weight interrupt support

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 2.2 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 2.2 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 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 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*) iden-

Table 2.1: 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*.

Table 2.2: 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.

tifying 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*).

2.1.2 Light-Weight Interrupt Support

Table 2.3: 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

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 2.3 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, and (3) notifications that the accelerators have 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 2.3 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 2.3-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 microarchitectural components added to support the light-weight interrupt are listed below:

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, and (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.

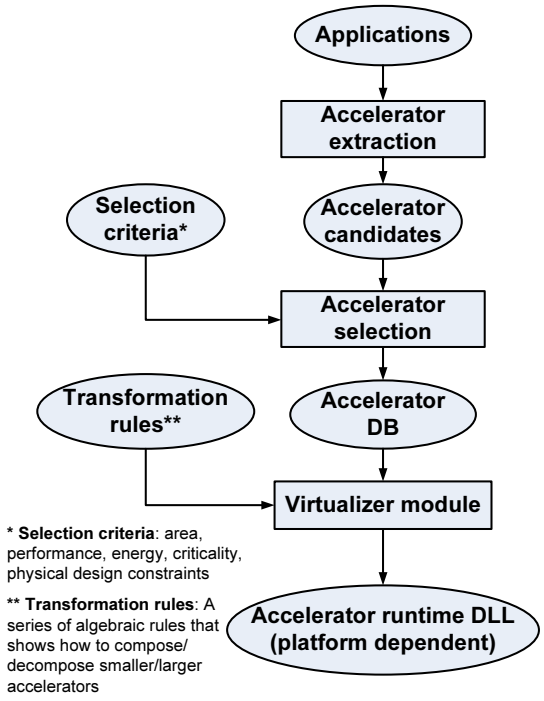


Figure 2.4: Accelerator extraction methodology

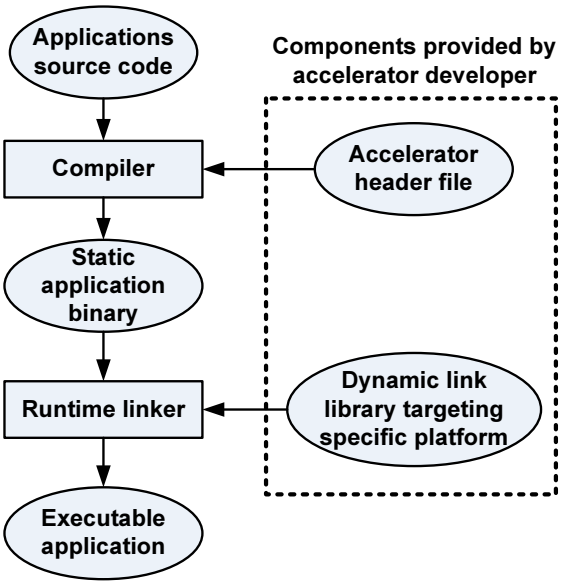


Figure 2.5: ARC development flow

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 2.2. *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.

2.1.2.1 Accelerator Extraction Methodology

Figure 2.4 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, such as area, performance, energy, criticality, and physical design constraints. This step generates an accelerator database, which 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 module called the *virtualizer*, which outputs a DLL that is used to link to executable files, as shown in Figure 2.5.

2.1.3 Programming Interface to ARC

The application programming interface (API) involved in using accelerators is presented in Figure 2.5. 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.

2.1.4 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 both be general as well as allow coordination of accelerators in groups that perform more complex tasks (described in Section 2.1.6). 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 con-

currently. 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.

2.1.5 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

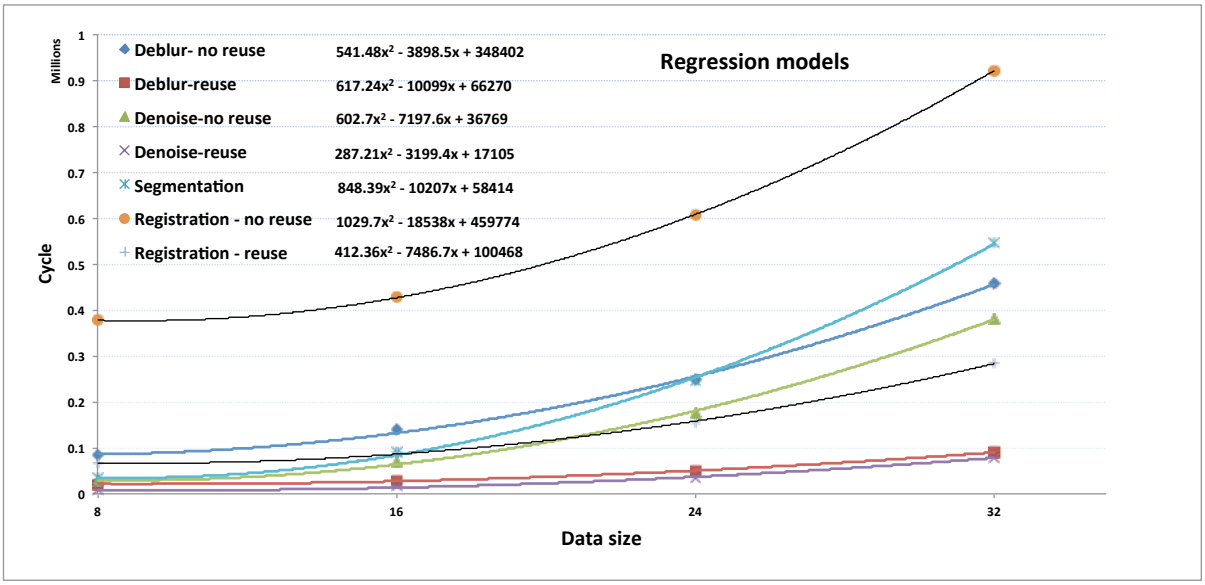


Figure 2.6: Regression models for medical imaging benchmarks

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 2.1.5.1); 4) the waiting list for each accelerator and the estimated run time for each job in the waiting list (Section 2.1.5.2).

2.1.5.1 Accelerator Run-Time 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 has been constructed based on profiled executions. We empirically found that a simple second order polynomial is sufficient to estimate execution time within 1-2% 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 2.5). Figure 2.6 shows the models we used in this work.

2.1.5.2 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. Note that the tasks being tracked in this waiting list are issued on a first-come-first-served (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.

2.1.6 Accelerator Composition

A key contribution of our work is the increased utilization of the available resources by either chaining accelerators together or otherwise composing accelerators to virtualize larger ones. In the next two subsections we discuss these techniques.

2.1.6.1 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.

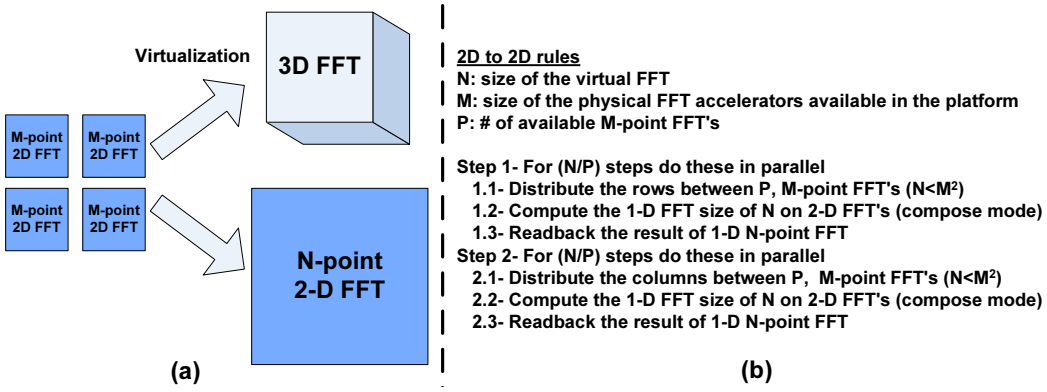


Figure 2.7: An example of accelerator composition

2.1.6.2 Accelerator Virtualization

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 2.7(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 2.7(b)), similar in style to the approach presented in [9]. 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 2.8 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

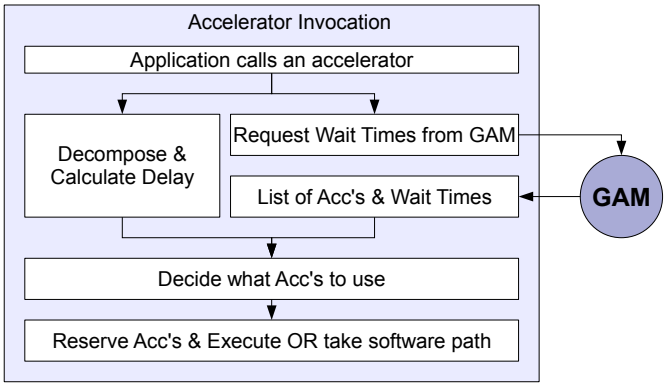


Figure 2.8: Accelerator composition steps

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.

2.2 Evaluation Methodology

2.2.1 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. Using shared LCAs, we have accelerated four algorithms from each of these two domains. Tables 2.4 and 2.5 provide brief descriptions of each

Table 2.4: Accelerated medical imaging algorithms

Application	**Algorithmic Functionality**	**# LCAs**
Denoise [10]	Total variation minimization	3
Deblur [11]	Total variation minimization and deconvolution	4
Registration [12]	Linear algebra and optimizations	7
Segmentation [13]	Dense linear algebra, spectral methods, MapReduce	1

application's computational characteristics and include the numbers of accelerators used.

FFT is a computation common to a wide range of scientific computing and signal processing algorithms, including use in a number of our chosen medical imaging benchmarks. We used FFT to show our virtualization results. As a point of comparison we are using FFTW [18] v3.3 for our software implementation.

When analyzing contention between multiple threads executing the same benchmark, we insert a barrier immediately before entering the benchmark kernel that was 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.

2.2.2 Simulation Tool-Chain

In order to make the exploration of this topic practical, a number of supporting tools have been created. These tools simplify the authoring of programs that used accelerators, and automate the process of implementing our chosen accelerators in our simulator framework. These tools are used in place of hand-written implementations and hand-adapted benchmarks to allow us to simulate systems that would have been prohibitively complex

Table 2.5: Accelerated computer vision and navigation algorithms

Application	**Algorithmic Functionality**	**# LCAs**
IDSI [14]	Computation of histograms based on intensity and distance of pixels	1
LPCIP [15]	Log-polar forward transformation of image patch surrounding each feature	1
SURF [16]	Feature orientation and computation of gradient histogram	1
EKF-SLAM [17]	Partial derivative, covariance, and spherical coordinate computations	2

to manually author, such as those that utilize many accelerators or feature 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 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 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 the 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 describe 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 previously. This code is responsible for coordinating interactions between accelerators, registering/handling interrupts, managing task descriptions and

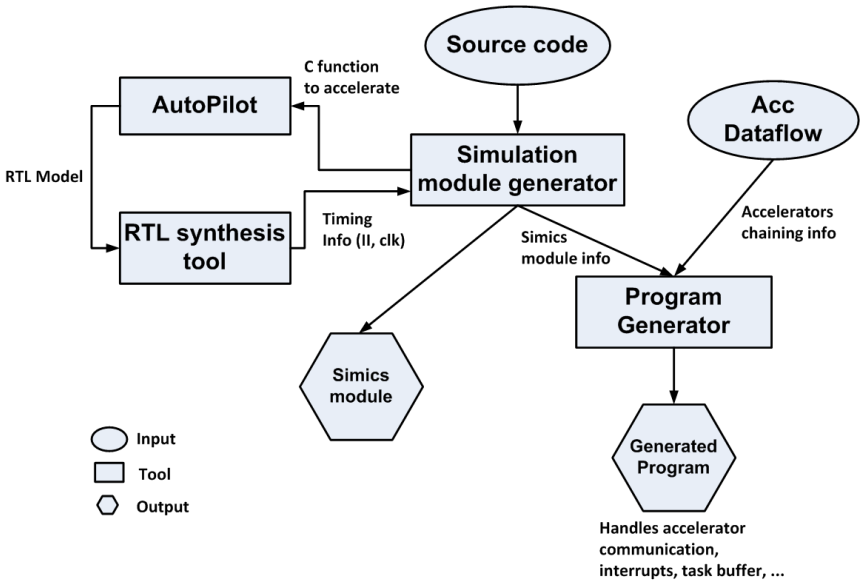


Figure 2.9: Process used to generate simulation structures and accelerated programs

accelerator resources, and dealing with accelerator-CPU synchronization. Figure 2.9 illustrates the work flow described here.

2.2.3 Simulation Platform

Our experiments were conducted using a heavily modified version of the Simics [19] and GEMS [20] simulation platform. The machine we model 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 2.6 shows the machine configurations we model in our simulations.

Table 2.6: 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 2.7: Synthesis results

	Deblur	Registration	Denoise	Segmentation	GAM	DMA-C
Clock (ns)	4	4	4	4	2	2
Area (μm^2)	4419917	12253775	1935539	2890354	12270	10071
Power (mW)	98.28	256.3	57.69	80.93	2.64	0.59

2.2.4 Area/Timing/Power Measurements

The AutoPilot behavioral synthesis tool [21] in combination with the Synopsys design compiler [22] are used to synthesize the C modules into ASIC. The timing information produced by the synthesis process is back-annotated to our accelerator modules to model cycle accurate accelerators. For computing energy we use power reports from Synopsys for accelerators and McPAT [23] for CPU power. Table 2.7 shows the synthesis results for the accelerators in our selected benchmarks together with the GAM and DMA controller.

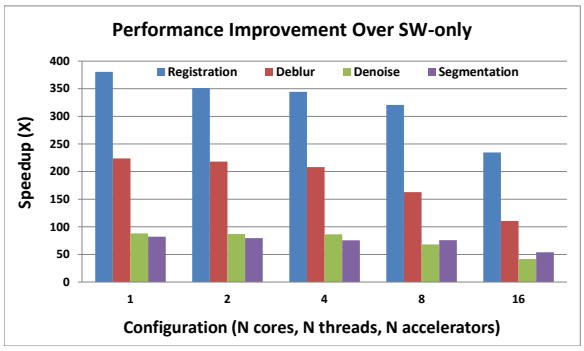


Figure 2.10: MI – Speedup over SW-only

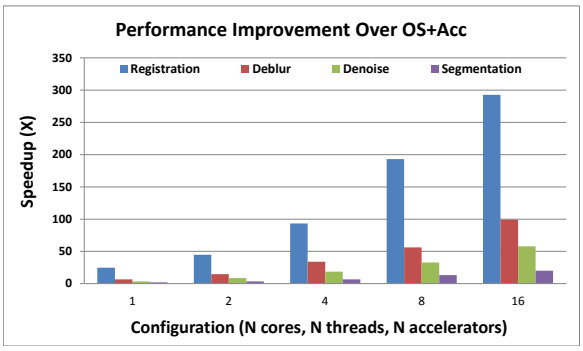


Figure 2.11: MI – Speedup over OS+Acc

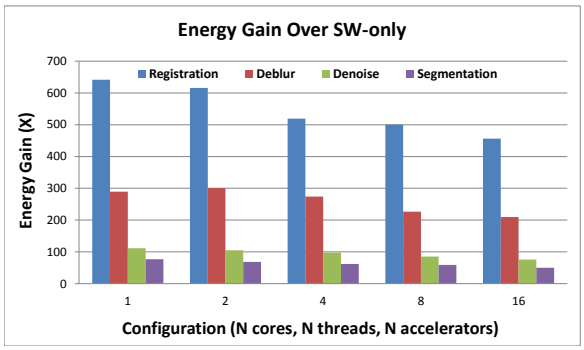


Figure 2.12: MI – Energy gain over SW-only

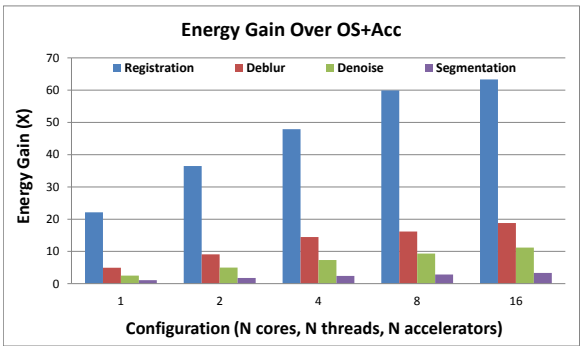


Figure 2.13: MI – Energy gain over OS+Acc

2.3 Experimental Results

To illustrate the effectiveness of our ARC platform, we evaluate a number of compute-intensive benchmarks from the domains of medical imaging (MI) as well as computer vision and navigation (VN). The following evaluation schemes are used:

- **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 resource arbitration managed by the hardware-based GAM.

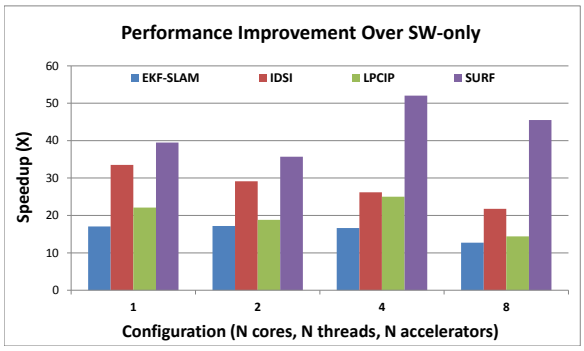


Figure 2.14: VN – Speedup over SW-only

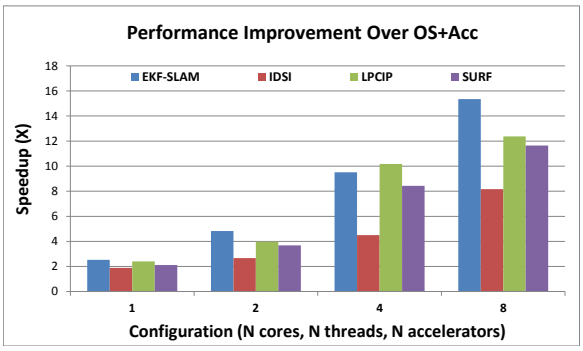


Figure 2.15: VN – Speedup over OS+Acc

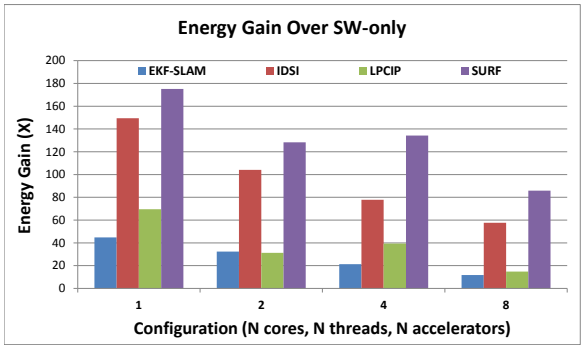


Figure 2.16: VN – Energy gain over SW-only

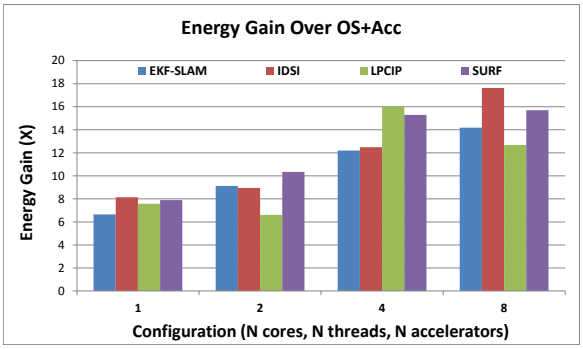


Figure 2.17: VN – Energy gain over OS+Acc

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

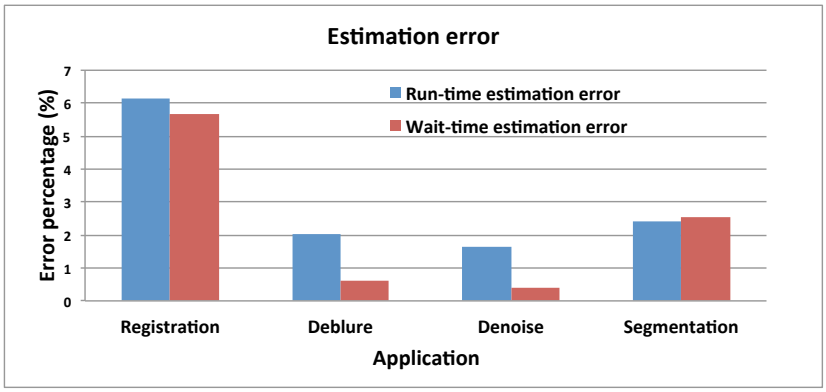


Figure 2.18: Core and GAM estimation error

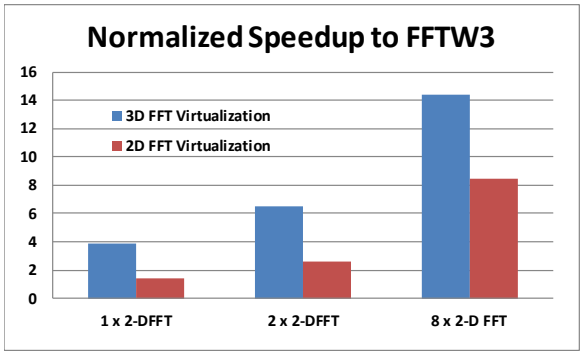


Figure 2.19: FFT virtualization (2D and 3D)

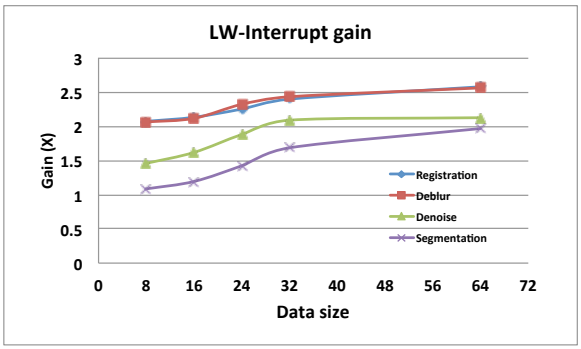


Figure 2.20: Benefit of using lw-int

2.3.1 Speedup and energy improvements

Figures 2.10, 2.14, 2.12, and 2.16 show the speedup and energy gain results for the ARC base configuration (Nc-Nt-Na) compared to running the software-only version of the benchmark on the same number of processors, number of 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. The VN benchmarks are shown to benefit relatively less from acceleration than the MI benchmarks, yet this is largely due to smaller data sizes being used for VN. As the data sizes are increased, more computation can be streamed through the accelerators, resulting in more utilization and efficient execution.

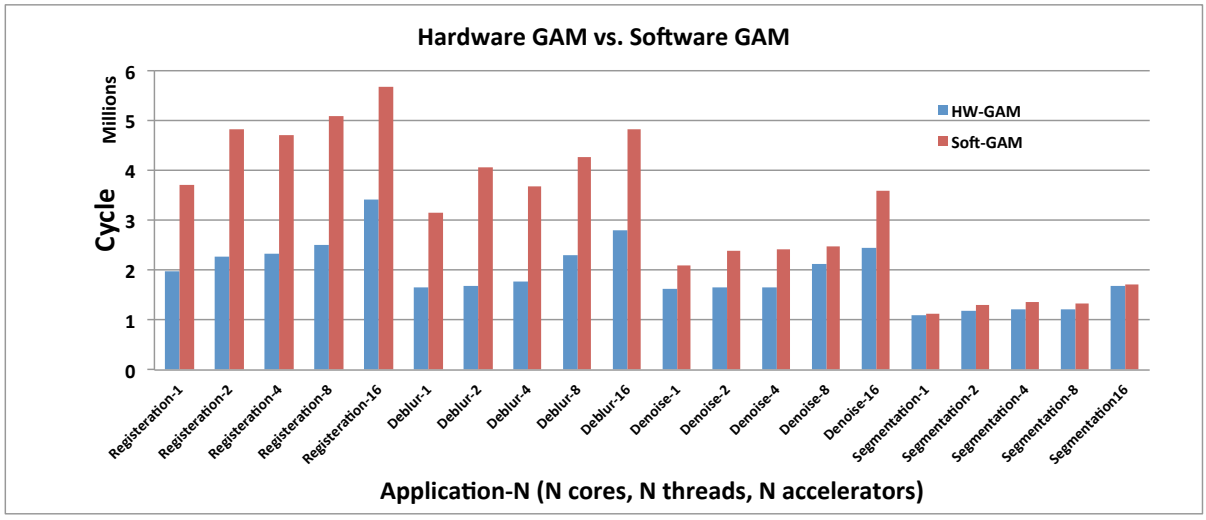


Figure 2.21: Benefit of using hardware GAM over SW-GAM

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 2.11 and 2.15 show the speedup gain ARC achieves compared 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 the number of threads and processors, the OS management overhead (thread context switching, TLB services, etc.) increases, and (2) for larger configurations, the number of interrupts also increase, which makes our system perform better due to the use of the light-weight interrupt in the place of the OS interrupts.

Figures 2.13 and 2.17 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.

2.3.2 Accelerator Sharing Results

Figure 2.18 shows the observed error for both the run-time and wait-time estimates for MI benchmarks. As can be seen by our estimated errors ranging from $< 1\%$ to 6%, execution times on our accelerators are sufficiently predictable for this to be a practical approach.

2.3.3 Accelerator Virtualization Results

Figure 2.19 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 [18]. In the best case for 3D-FFT we obtained 14.4X speedup and for 2D-FFT we obtained 8.4X speedup.

2.3.4 Benefits of Light-Weight Interrupt

To measure the benefits of the 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 2.20 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.

2.3.5 Benefits of Hardware-Based GAM

We examined the possibility of using an OS process, called the SW-GAM, to handle the responsibilities normally associated with the GAM. This approach differs from the previous OS-managed approaches in that it still does not rely on a accelerator driver for communication. Added instructions are still included for accelerator communication, and the light-weight interrupt interface is used both by the calling thread and the SW-GAM. To give the most opportunity for the SW-GAM to compete with the dedicated hardware GAM, we allocated a processor exclusively for the SW-GAM. Figure 2.21 shows the benefit for the N cores, N threads, N accelerators (N = 1, 2, 4, 8, 16) configurations for fixed data size. The best results is for registration (almost 2X), since there are more accelerators and thus the GAM is responsible for allocating more resources. On the other hand segmentation has only one accelerator, which reduces the advantage of using a dedicated hardware GAM (only 10% benefit). The larger the configuration size, the more interaction with GAM which is why we see more benefits for hardware GAM for larger values of N.

CHAPTER 3

CHARM: A Composable Heterogeneous Accelerator-Rich Microprocessor

While large-scale accelerators, like those discussed in Chapter 2, offer a large degree of performance and energy efficiency, they are not appropriate for algorithms that are not mature computations, and see low utilization. LCAs are only appropriate for targeting mature computations because of the immutable nature of hard logic. As soon as an algorithm changes, the LCA implementing that algorithm becomes useless. Low utilization stems from two facts: 1) LCAs implement a very specific functionality, and thus anything that requires a different functionality cannot use the LCA, and 2) a programmer must know that an LCA exists that implements their desired functionality, and actually make use of that hardware. The second point primarily stems from the fact that it is intractable to prove equivalence of subprograms over general code.

To deal with both of these problems, we put forward an architecture that focuses more on composition of compute engines rather than development of a series of monolithic compute engines. This architecture was called the Composable Heterogeneous Accelerator-Rich Microprocessor (CHARM) [24] architecture, and introduced sophisticated computation via communication between a series of simple components. These simple components, called accelerator building blocks (ABBs) are distributed throughout the system, and each implement a relatively small functionality. A program would the construct a graph of communicating ABBs that together describe a complex operation, and send this graph to a resource management device referred to as the accelerator block composer (ABC). The ABC would then act as a proxy for communication between a communicating sea of ABBs and the host processor, thus lending not only high perfor-

mance and energy efficiency, but fully virtualized accelerator resources and scheduling, along with hardware managed work distribution.

This compute model not only allowed for compilers to assist in discovering accelerator regions, since ABBs implement functionalities that are individually small enough to discover in source code, but also introduces a more expressive accelerator framework that is able to target many applications using a small set of common functions. Hardware work scheduling and balanced work distribution also enabled very high resource utilization, with all ABBs in a system able to cooperate in performing even small computations via hardware managed task distribution.

3.1 Microarchitecture of CHARM

In this section, we address our design goals by way of an architecture that provides flexibility, scalability, and design reuse.

3.1.1 CHARM Software Infrastructure

CHARM's software component is responsible for: (1) *LCA candidate selection* – identifying program hotspots that would benefit from LCA implementation [8]; (2) *ABB selection* – generating a set of ABBs to cover a set of LCAs under physical design constraints (area, timing, power) [25] and (3) *ABB flow graph creation* – used to compose LCAs from ABBs. While the LCA candidate selection is done manually, the processes of ABB selection and flow graph creation are automated.

The structure of the ABB flow graph is the same as that of a task flow graph [26] (see Figure 3.1). Each node is a task that is represented by a desired ABB invocation, with edges representing memory transfers between ABBs. In memory, this graph consists of a list of ABBs that are part of the LCA, followed by an enumeration of memory transfers. Each ABB node consists of a type, an enumeration of starting addresses for locating argument streams in a virtually addressed private scratchpad memory (SPM)

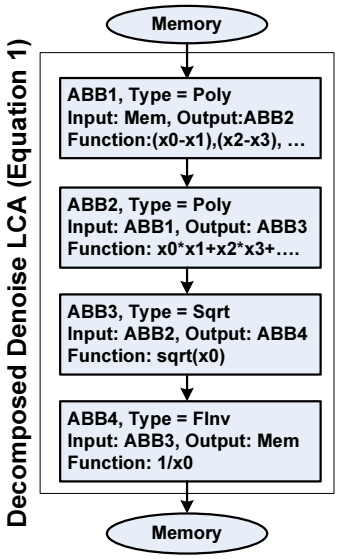


Figure 3.1: Composition

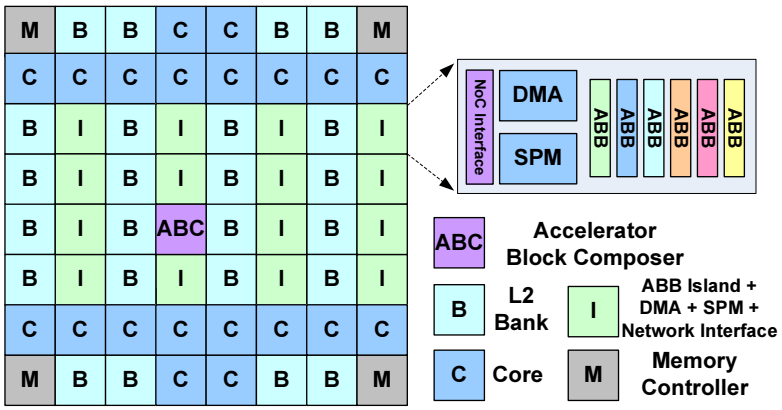


Figure 3.2: Microarchitecture of CHARM

region, and settings for any local configuration registers. Each memory transfer consists of an identifier for a source and destination device (either memory or an ABB node). It also includes a starting address and a series of size-stride pairs describing a polyhedral (regular, high-dimensional) space for both the source and destination. This graph is easy to parse, and consists mostly of values that are directly usable by various control registers on the ABBs and associated DMA. We further note that when a data flow graph is created, it is not tied to any physical instance of the ABBs. The flow graph simply connects virtual ABBs together as a template of an LCA. Figure 3.1 illustrates this for one of the LCAs used in Denoise, whose functionality is formulated by Equation 3.1:

$$1/\sqrt{\sum_{i=0}^{6}(x_c - x_i)^2} \tag{3.1}$$

The hardware will then map physical instances of ABBs to this LCA template on the fly to instantiate a virtual LCA.

3.1.2 Hardware Infrastructure

While the software is responsible for specifying candidates for acceleration and detailing how ABBs may be composed into particular LCAs, it is the hardware's responsibility to allocate ABB resources to particular threads to satisfy software demand. For this paper, we will restrict each ABB to be allocated to at most one LCA at a time. Our hardware will arbitrate use of the ABBs and LCAs among multiple competing threads/cores, and allocate resources in a way that maximizes the utilization of available resources (i.e. load balances requests from one or more cores among multiple LCAs). Note that additional complications exist due to variation in latency when streaming data to LCAs (i.e. caused by TLB and cache misses, congestion on the NoC, etc.) and varying contention for the use of any given ABB. A dynamic solution is preferable in order to adapt to nondeterminism in LCA memory latency and to the varying LCA demand across different cores.

Figure 3.2 shows an example of the CHARM microarchitecture. It consists of cores, L2 cache banks, memory controllers, ABB islands and an *accelerator block composer* (ABC), which is the means of control for composing ABBs and essentially the mechanism by which we provide dynamic adaptation. We describe the ABC in more detail below.

Each ABB island has a small dedicated SPM, dedicated DMA engine and NoC interface. The SPM allows ABBs, when composed into an LCA, to have a fixed data access latency. By using memory streaming and task partitioning, and by overlapping communication with computation, the SPM size can be kept small. The allocation of SPM to each ABB is handled by the ABC.

The dedicated DMA engine in each ABB island is responsible for transferring data between the SPM and the L2 cache, and also between SPMs in different ABB islands (i.e. accelerator chaining or remote DMA [27]). In addition, each DMA has a small internal TLB, allowing LCAs to work with virtual addresses. In the event of a TLB miss, the DMA will forward its request to the ABC (see Section 3.1.2.1 for details on ABC TLB handling).

3.1.2.1 ABC Design

In our scheme, the ABC is contacted by cores that need access to an LCA. It then allocates ABBs to satisfy this request. An LCA can consist of any number of ABBs, provided that number is less than the number of ABBs that is available in the system. The ABC uses five components to manage its collection of ABBs: a Resource Table, a Composed LCA Table, a collection of Task Lists, a TLB, and a Data Flow Graph Interpreter.

Resource Table: The ABC has a Resource Table that it uses to track the allocation of different ABBs to LCAs. When a core requests the use of an ABB, the Resource Table is queried to determine which ABBs are available. If enough ABB resources are available, multiple instances of a particular type of LCA may be instantiated, assuming the computation to be done is large enough for these multiple instances to each perform non-trivial amounts of work. The ABC uses a two-tiered allocation policy to decide which ABBs to compose into a given LCA. First, the ABC will attempt to balance the concentration of memory-accessing ABBs across the entire system. The purpose of this is to limit contention in the DMA associated with each node. Second, the ABC will employ a simple greedy approach to select ABBs that are local to other ABBs they communicate with. This is done in order to minimize the cost of communication between ABBs. To further reduce latency, ABBs within the same island may use a common SPM for communication (rather than each using their own SPM in their respective islands) and eliminate the need to communicate through the NoC. When ABB resources are scarce, the above metrics degrade to greedily constructing LCAs out of any available ABBs, rather than waiting for more optimal choices to become available.

Composed LCA Table: To eliminate the need to repeatedly compose the same LCA out of the same ABBs when tasks are completed, a Composed LCA Table is introduced. This table tracks ABB allocation, and is used to remove the overhead of remapping patterns when an LCA is already composed.

Task Lists: When the ABC receives a request for an LCA, the requested computation

is split into a number of fixed-size data chunks to enable efficient parallelism. Each of these is referred to as a task and the ABC maintains these in a Task List. Each entry in the task list consists of a marker identifying which LCA the task belongs to, which task of the whole computation the entry belongs to (for that specific LCA invocation), and a bit flag marking it as runnable or not runnable. As tasks are added to the Task List, the ABC iterates over the memory addressed by the task, and checks its local TLB. If all addresses in a task are resolvable by the internal TLB, the task is marked as runnable. Otherwise, it is marked as not runnable, and the ABC issues a TLB miss to the requesting core. The ABC uses a round robin scheduling policy to iterate through all LCAs that have at least one task marked as runnable. So long as there are tasks that are marked as both runnable and for which there are enough ABBs to compose, the ABC continues attempting to compose more LCAs, and continues issuing tasks. We plan to implement more complex scheduling policies based on task priority and criticality in the future.

TLB: The ABC maintains a shared TLB that caches address translations among all tasks in its task list. This allows the ABC to prescreen tasks for TLB misses prior to composition. If multiple ABBs under control of the ABC would have encountered the same TLB miss, the ABC can avoid sending duplicate requests to the corresponding core and simply satisfy these misses locally with its own TLB.

Data Flow Graph Interpreter: Our software framework provides composition instructions in the form of a data flow graph. These graphs are fed as resource instantiation templates from the cores to the ABC. Each node in the data flow graph needs to be allocated to a particular ABB, and each ABB is only assigned to a single graph node, and a single LCA, at a time. When an ABB finishes with the work for a single task, it notifies the ABC that it is free for reassignment. If there are more tasks marked as runnable associated with the LCA to which the ABB was allocated, it is given another task from this set. If there are no runnable tasks associated with that LCA, the ABB becomes eligible for composition into a different LCA. We considered keeping LCAs composed for a longer duration to exploit potential locality of use of a particular LCA, but found that the overhead involved in mapping a set of ABBs to an LCA template is small enough

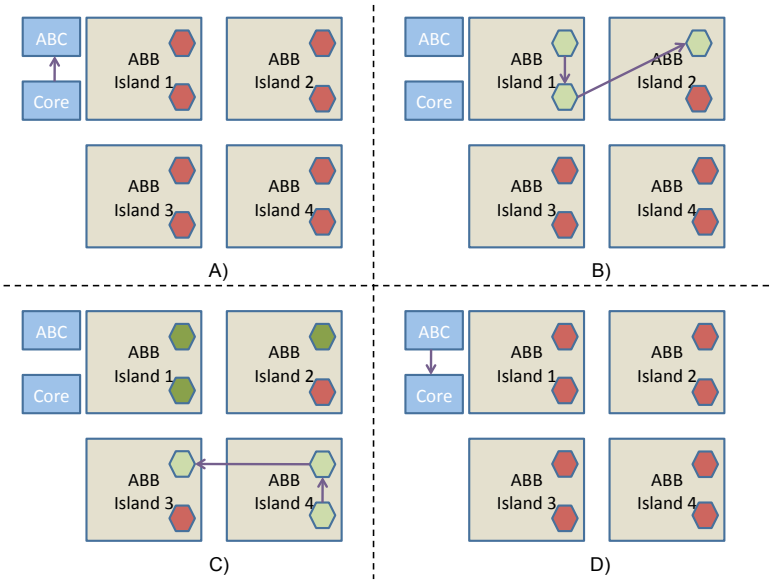


Figure 3.3: LCA composition example: A) A core sends a request for an LCA to the ABC; B) An LCA instance is allocated; C) An LCA instance is allocated with consideration for balancing DMA utilization; D) The ABC signals completion to the core.

such that releasing resources immediately is preferable due to the improved utilization of ABBs across multiple LCAs. This means that ABB utilization varies over the course of execution of a particular task, and it may be possible for there to be multiple constructed copies of a particular LCA at a given time, even if this is not possible when a given core initially requests an LCA. Therefore, as long as the ABC has runnable tasks in the Task Lists for a particular LCA, we allow it to attempt to compose additional copies of that LCA. In this way, the ABC can eventually make use of all available resources. In this paper, we do not allow ABB preemption (except in the event of error, such as an access violation in the requesting core), but we will explore this for future work.

3.1.2.2 Example of Composition

Figure 3.3 shows an example of LCA composition for an architecture with 4 ABB islands. This example architecture has eight ABBs (shown as hexagons), with two of them in each ABB island. We assume for the sake of simplicity that all ABBs in this example are of the same type. The ABC and a requesting core are shown in the upper left-hand side

of the figure. In this example, the core requests the composition of an LCA consisting of three ABBs in sequence, with the first ABB reading from memory and the last ABB writing to memory. The core sends a data flow graph (DFG) of the desired LCA to the ABC (Figure 3.3A). The ABC then interprets the DFG, splits the request into tasks, and begins cycling over the addresses each computation will access. It puts each of these chunks in the task list. For this example, we assume there is more than one task associated with this LCA invocation, and that the ABC's local TLB has the required pages to make all tasks immediately runnable. The ABC then examines the availability of ABBs, discovering that they are all free, and begins allocating.

Since at least one task is made runnable, the ABC proceeds to execute the allocation algorithm described in Section 3.1.2.1. After finding a match, consisting of two ABBs in Island 1 and a single ABB in Island 2 (Figure 3.3B), the ABC makes an entry in its Composed LCA Table, marking these ABBs as belonging to this specific LCA. At this point, it chooses a runnable task from the task list belonging to this LCA type, and dispatches a task. The ABC then begins attempting to map another instance of the requested LCA to available ABBs, and finds two ABBs in Island 4 and one in either Island 2 or Island 3. The allocation algorithm chooses to use Island 3 for the last ABB instead of Island 2 to distribute load across more DMAs (Figure 3.3C). This process is then stopped since there are not enough ABBs to construct any additional LCAs. As ABBs finish their assigned work, they signal to the ABC that they are finished. Each time the first ABB in an LCA signals the ABC of completion, the ABC checks its task list for runnable tasks. If it finds a task, it begins sending a new task to each ABB in that composed LCA. If it does not find a task, it marks this LCA as retiring, and marks the associated ABB(s) as free. Each time an ABB that was part of a retiring LCA is marked as free, it is made available to be recomposed into a new LCA. When all ABBs of all clones of a retired LCA are freed in this manner, an interrupt is sent to the requesting core marking the completion of the requested computation (Figure 3.3D).

Table 3.1: Simulation parameters

Parameter	Value
Processor	Ultra-SPARC-III-i @ 2.0GHz
Operating system	Solaris 10
L1	32-KB, 4-way set-associative: 1-cycle
L2	8-MB, 8-way set-associative: 10-cycles
Coherence protocol	Shared banked L2-cache, L2:MOSI, L1:MSI
Memory	1000-cycles, Directory 6-cycles
Network topology	MESH, latencies: link 1-cycle, router 5-cycles

3.2 Evaluation Methodology

To evaluate the CHARM architecture, we have modified Simics [19] and GEMS [20] to model accelerator-rich many-core architectures. Table 3.1 shows the parameters used in our simulations.

We have also implemented a series of supporting tools to automatically generate accelerators as well as application code that makes use of these accelerators. For calculating energy, we used the power result output from Synopsys for LCAs and ABBs, and used McPat [23] to generate power values for cores and caches.

Table 3.2 and Table 3.3 show the area and power overhead (using the Synopsys 32nm SAED library and CACTI 5.3 [28]) for the selected ABBs and LCAs corresponding to each benchmark. We have also included the synthesis results for the ABC that implements the ABB allocation algorithm mentioned in Section 3.1. To study the overhead of ABBs, we have synthesized the Poly16 ABB, the results of which are shown in Table 3.2. The internal structure of a Poly ABB is shown in Figure 3.4 (this Figure is actually showing a Poly8). It consists of adder/subtractor/multiplier (ASM) modules, an SPM bank, and control logic which controls access to the SPM bank. The SPM bank has 3 sub-banks (for simultaneous read/compute/write) each one with 1 read/write port. One sub-bank is connected to the ASMs and two are ported to the DMA controller (DMAC). For the

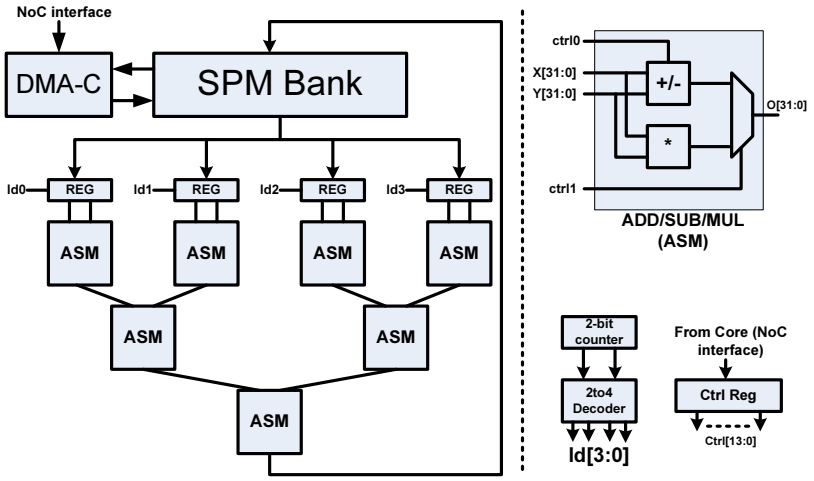


Figure 3.4: Poly ABB Details

Table 3.2: Area/Power results – CHARM

Name	**A (u^2)**	**P (mW)**	**Total #**
FDiv	4949	0.264	12
Poly16	38276	1.608	96
FInv	3503	0.141	12
FSqrt	58683	1.83	8
SPM-4KB 1R/W	13591	17.6	288
SPM-768B 1R/W	2545	7	72
ABC	8383	0.066	1

experimental results, each ABB island in our design has 16 ABBs and 16 SPM banks to provide concurrent access to all the ABBs. We have used 128 ABBs in our design: 8 ABB islands, each having 3 FInv/FDiv, 1 FSqrt, and 12 Poly16 modules, along with 16 SPM banks. Table 3.4 shows the area for the main components of the chip. Note that "CHARM HW" accounts for the area of the ABB islands and the ABC.

We modeled a system consisting of 1 to 8 processors, and a set of either physical LCAs or ABBs. When modeling a system consisting of physical LCAs, we included all the accelerators required to run a single instance of each benchmark, without contention.

Table 3.3: Area/Power results – LCAs

Name	**A(u^2)**	**P(mW)**	SPM Banks
Denoise	496908	16.5	6
Deblur	2013228	110.9	9
Segmentation	688298	27.3	6
Registration	3853098	183.9	18
EKF-SLAM	1188252	42.0	24
LPCIP	239159	6.11	6
SPM-2KB 2R,1R/W	37043	17.5	–

Table 3.4: Area (mm^2) for various chip components

Core	**NoC**	**Cache & Dir**	**CHARM HW**	**CHARM Total**	**LCA HW**	**LCA Total**
10.8 (scaled to 32nm)	0.3 Ref [29]	39.8 Ref [28]	8.3 (Table 3.2) (14%)	59.2	8.5 (Table 3.3) (14.3%)	59.4

To illustrate the load balancing capacity of our ABC, we modeled instances in which we have some multiple of this number of accelerators. When modeling a system featuring ABBs, the number of ABBs corresponds to the total amount of area that would have otherwise been devoted to LCAs. As a baseline (i.e. 1x ABB area), the total area consumed by the ABBs equals the total area of all LCAs required to run a single instance of each benchmark (this can be verified by the number of ABBs in Table 3.2 and the LCA area numbers in Table 3.3). All ABB numbers are multiples of this base amount. We configured our system to have 8 ABB islands, and scaled the number of ABBs present on each island. We also scaled the amount of SPM space on each ABB island proportionally.

3.3 Experimental Results

We compare the following architectures to evaluate CHARM:

Physical LCA sharing with Global Accelerator Manager (LCA+GAM): In

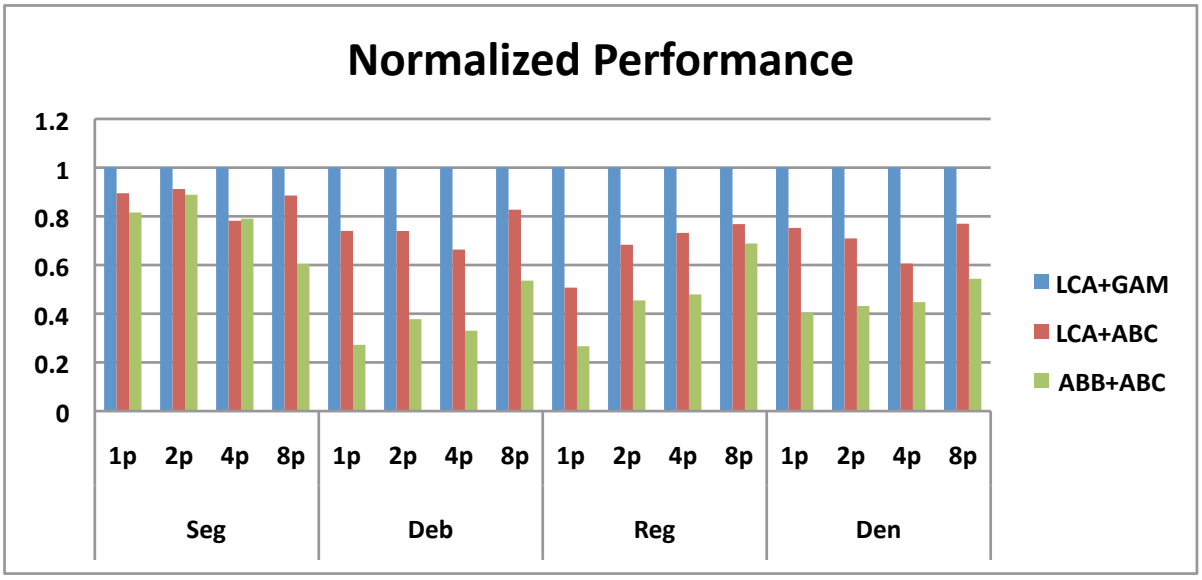


Figure 3.5: Performance improvement

this architecture, physical LCAs can be shared between multiple cores. Each benchmark in our domain is accelerated with special-purpose accelerators. A global accelerator manager (GAM) is implemented in hardware to dynamically allocate physical LCAs to cores. We examine cases where there are between 1 and 8 replicates of each required accelerator. This allows for the concurrent execution of multiple instances of any specific benchmark, one for each accelerator in the system. Also, LCAs are powered off when not in use. This approach is similar to the architecture in [8].

Physical LCA sharing with ABC (LCA+ABC): In this architecture, a core may share physical LCAs using a centralized hardware ABC. In addition, the ABC can load-balance the available physical LCAs. We examine cases where there are between 1 and 8 replicates of each required LCA. Since the ABC can split tasks among multiple LCAs, we are able to take advantage of all LCAs of a given type, even when only a single instance of that benchmark is executing. In the cases where there are more LCAs than can be allocated to available tasks, extra LCAs are left powered off.

ABB composition and sharing with ABC (ABB+ABC): In this architecture, a centralized hardware ABC is responsible for composing and managing available ABBs, load balancing the tasks, and managing TLB requests from ABBs. We examined multiple

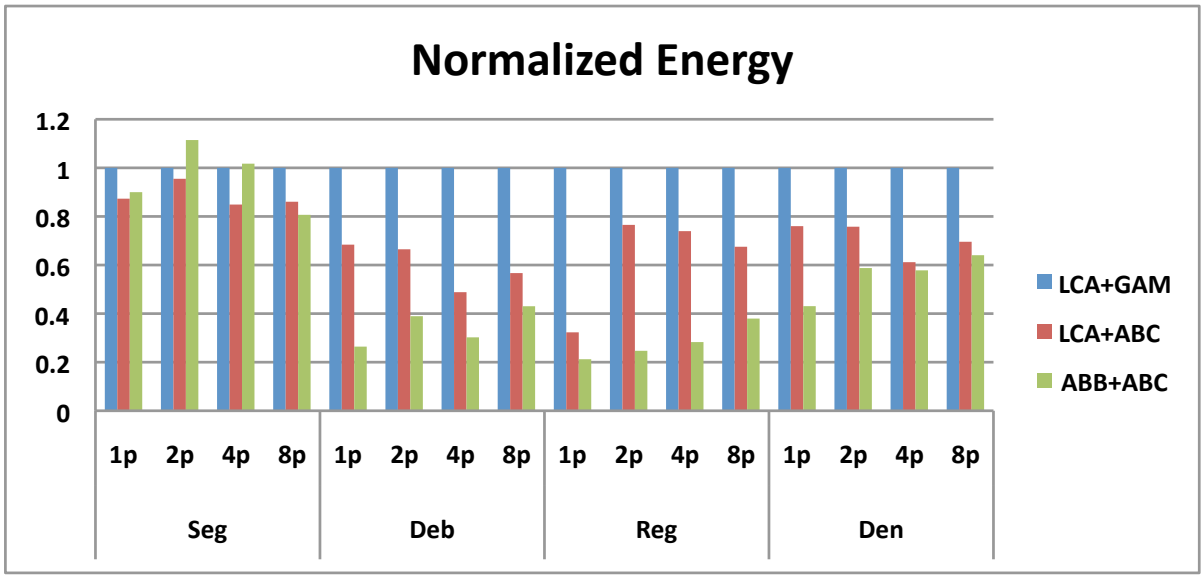


Figure 3.6: Energy improvement

ABB quantities. For the purposes of making a comparison, we will refer to a quantity of ABBs with area equal to a single replicate of each LCA in the domain to be comparable to the case where we have one of each physical LCA in the domain. Typically these ABBs can be used to make multiple virtual LCAs, but this gives us a metric by which to make a fair comparison to the LCA+GAM and the LCA+ABC cases. In the cases where there are more ABBs than can be constructed into LCAs, the extra ABBs are left powered off.

For all cases, unless otherwise stated, we ran our 4 selected benchmarks from the medical imaging domain. The benchmarks were run with volumetric images of 32-pixel cubes in multiple iterations.

3.3.1 Improvement over LCA-based systems

Figure 3.5 and Figure 3.6 show the normalized performance and energy improvements we observed by using the ABB+ABC scheme compared to the LCA+ABC and LCA+GAM schemes. All numbers shown here are normalized to the corresponding LCA+GAM result. In each case, we have the same number of processors, threads and accelerators

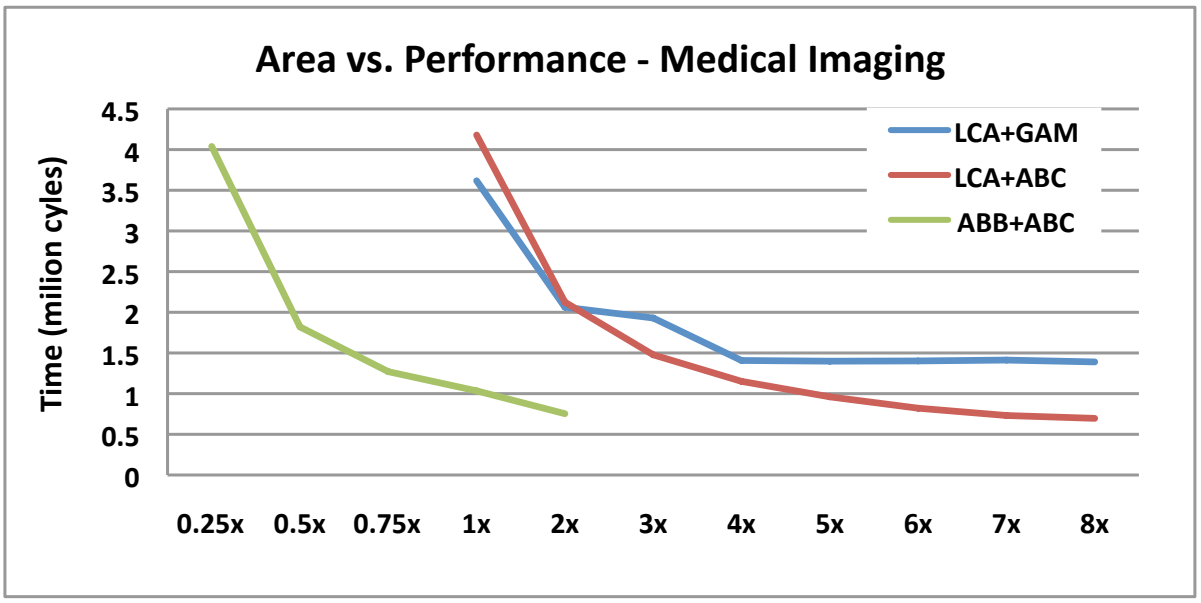


Figure 3.7: Effect of increasing accelerators

(e.g. the 4p case has 4 processors, 4 threads, and 4 accelerators). On average we observe more than 2.4X energy improvement over LCA+GAM (maximum 4.7X) and 1.6X energy improvement over LCA+ABC (maximum 3X). In general, as the number of independent tasks increases, ABB+ABC shows better performance because the ABC starts composing ABBs to create new LCAs (so long as ABBs are available in the system). This creates more parallel tasks, thereby achieving better performance and consuming less energy. We note that Segmentation-2p using ABB+ABC shows higher energy usage compared to other schemes. The reason for this is that the performance for segmentation improves only slightly. Therefore, the overhead of constructing LCAs and coordinating communication between ABBs consumes more energy than what is conserved by the slight reduction in execution time.

3.3.2 Effect of adding accelerators

Figure 3.7 shows the effect of adding more accelerator resources on the performance in all of our studied schemes. We observed a very similar result for energy improvement as well. For this experiment, we fix the number of processors and threads at 4. For LCA

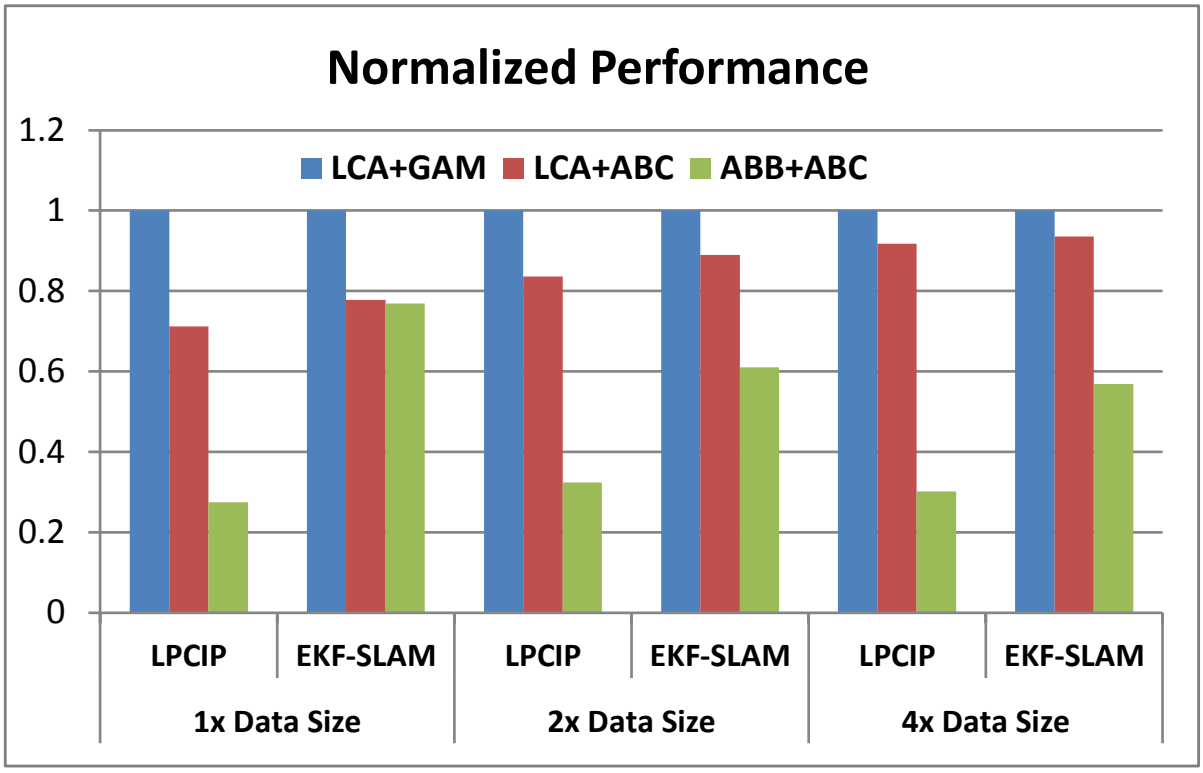


Figure 3.8: Performance improvement for computer vision and navigation

cases (LCA+GAM and LCA+ABC), the number of LCAs ranges from 1 to 8. For the ABC+ABB scheme, the quantity of ABBs ranges from 1/4 of the ABB number that area-wise matches one set of the LCAs in the domain, to two times that number.

There are several observations for these results. First, adding more accelerator resources in general improves speedup and energy. Second, as accelerator resources are increased, significant performance improvements are seen much earlier in the ABB+ABC case than in the LCA schemes (notice 1.5x-2x case in ABB+ABC vs. 6x-8x case in LCA+GAM and LCA+ABC). The reason behind this is that in the ABB+ABC scheme even 1x area allocation can reconstruct many copies of a virtual LCA to run concurrently. This is because a benchmark using physical LCAs only uses those of a specific type, and thus only a small number of the total LCAs. The ABB+ABC is free to replicate virtual LCAs out of the entire sum of accelerator resources, rather than leaving area unutilized. An implication of this is that the ABB+ABC case saturates much more quickly in the acceleration that it can offer, either exhausting potential parallelism or becoming mem-

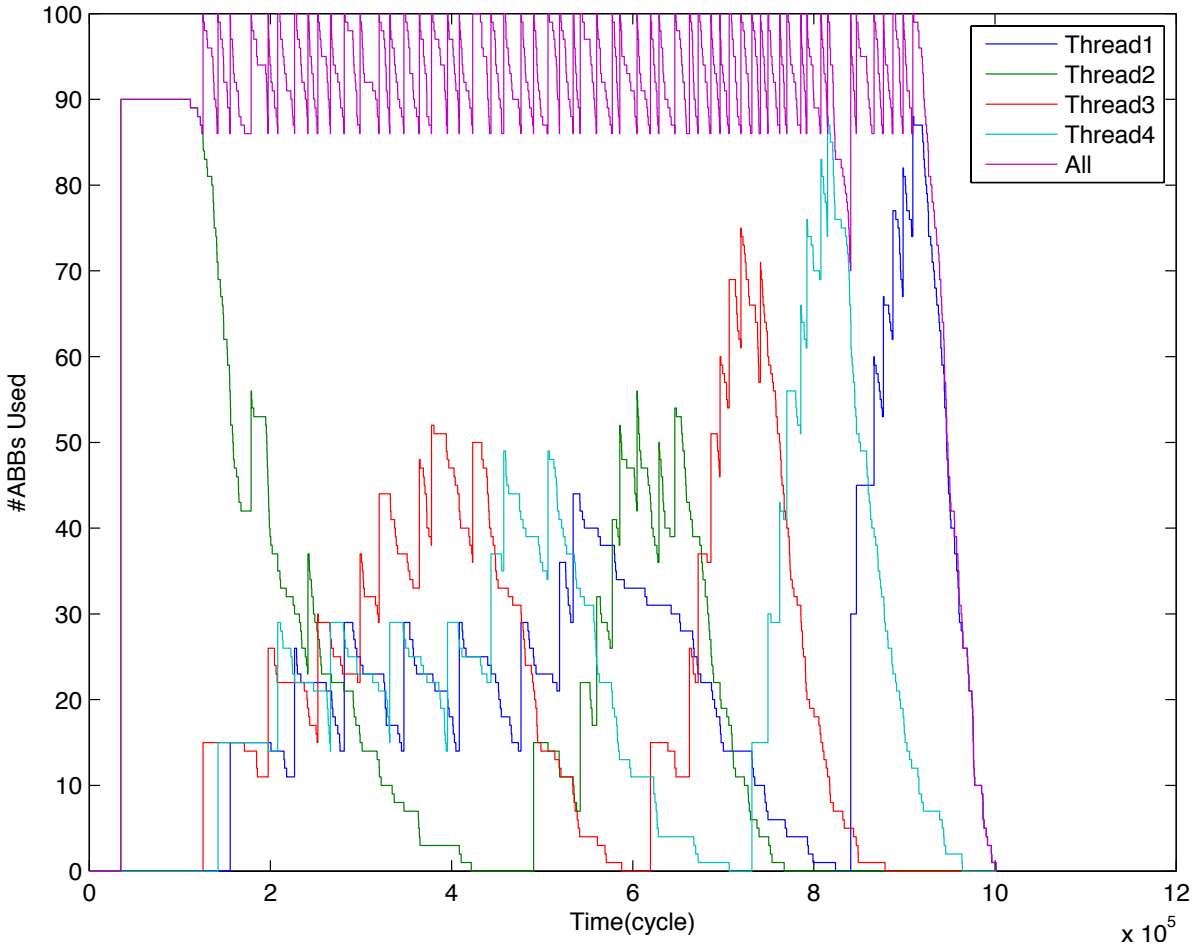


Figure 3.9: Utilization of ABBs given a task-grain of 8

ory bound. Third, after 4x, the LCA+ABC case still continues improving performance and energy, but LCA+GAM flattens. This is because ABC splits each individual LCA invocation into multiple tasks and load balances these tasks among accelerator resources, thereby benefitting from having more than one LCA per accelerator invocation. The GAM, which allocates accelerators directly to the calling thread, is not capable of doing this without the software actually having requested multiple accelerators.

3.3.3 Effect of changing task-grain

Task-grain is the maximum number of individual computations in each task assigned to a set of composed ABBs. The smaller the task-grain, the more parallelism in cases

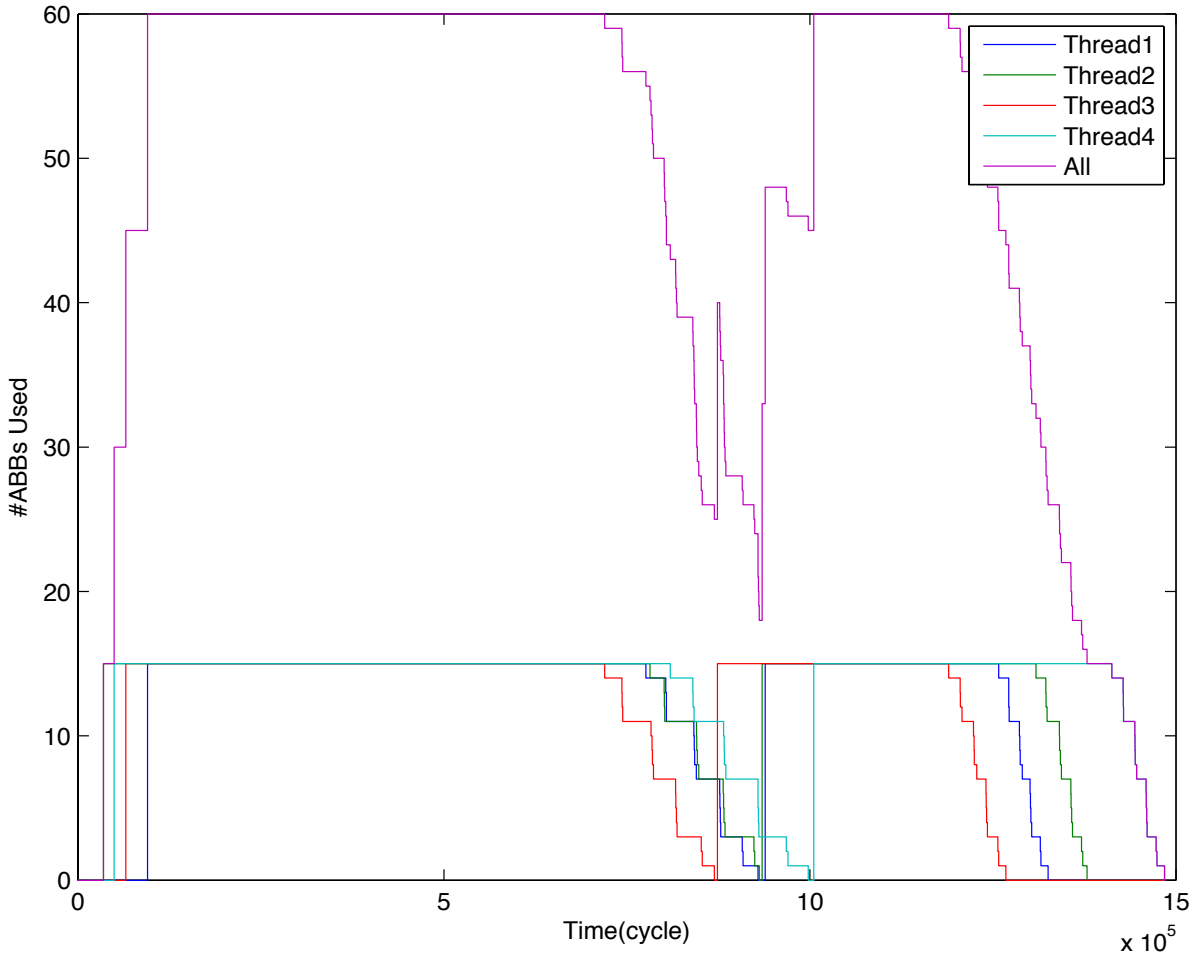


Figure 3.10: Utilization of ABBs given a task-grain of 128

where computations can be performed independent of one another. In order to measure the effect of task-grain on the ABB usage of each thread, we measured the number of ABBs allocated to LCA invocations by each thread for every moment of execution. For brevity, we are only showing results for registration, but all benchmarks we examined exhibited the same characteristics. We show this utilization for two cases: task-grain of 8 and task-grain of 128 as shown in Figure 3.9 and Figure 3.10, respectively. Each figure shows the ABB usage by each thread and the total number of ABBs used (the upper most curve). When the task-grain is 8, there is more parallelism and so more ABBs can be quickly allocated to a given thread (e.g. the initial spike seen in Figure 3.9). When the task-grain is 128, only one set of ABBs is used by each thread. The values shown in Figure 3.10 describe the ABBs allocated for a single LCA instance per thread. Also

shown in Figure 3.9 is the impact of our round robin scheduling. This assures a measure of fairness when allocating ABBs. The jagged total use is the result of freeing ABBs prior to their reassignment.

3.3.4 Platform flexibility

An original argument we put forward as a justification for this approach was the reusability of this system in terms of block design as well as retargetability. To substantiate this argument, we examined two applications from two domains that are completely unrelated to medical imaging: computer vision and navigation. Computer vision and navigation require compute-intensive data processing, consisting heavily of linear algebra and floating point computation, to attain high levels of situational awareness. We examine log-polar coordinate image patches (LPCIP) [15] from computer vision and extended Kalman filter-based simultaneous localization and mapping (EKF-SLAM) [17] from navigation. A more detailed description of these two applications and existing acceleration strategies can be found in [30]. Figure 3.8 shows results comparing the use of our medical imaging platform (unmodified and implementing virtual LCAs) against custom physical LCAs that specifically target these new domains. This illustrates that our platform is flexible, and is much more broadly targetable than a typical platform featuring custom LCAs.

CHAPTER 4

Progress on Developing Accelerator-Rich Architectures

Chapter 3 primarily focused on the microarchitectural impacts on a system of introducing CHARM, but discussed only briefly on the the microarchitecture of many of the devices introduced as part of CHARM. This chapter focuses on the design of the ABB islands themselves, along with details about internal structures. These were considered as separate works primarily because of the scope of the topic to cover.

The methodology guiding the design of ABB islands differs from conventional embedded system design in that we can leverage compute resources outside of island to allow us to make assumptions about the design of internal components. For example, the network providing connectivity between components internal to an ABB island does not need to provide uniform connectivity between all internal devices, because we can leverage the controlling core and ABC to make allocation decisions that bias toward certain communication patterns. This chapter discusses this, and other observations that have allowed us to create an efficient ABB island.

4.1 Progress on Developing Accelerator-Rich Architectures

We began our investigation of accelerator-rich architectures in 2010 and developed three generations of architecture templates. The first generation of architectures focused on hardware support for accelerator management (ARC) [8]. Figure 4.1-A shows the overall

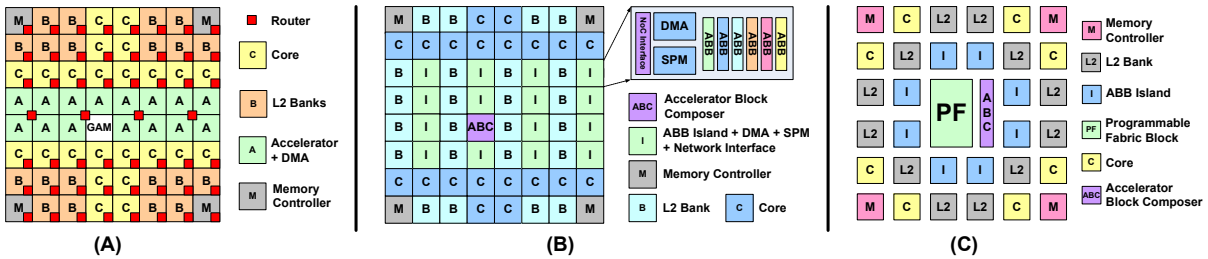


Figure 4.1: Overview (not to scale) of accelerator-rich architectures: (A) ARC; (B) CHARM; (C) CAMEL

architecture of ARC, which is composed of cores, accelerators, the Global Accelerator Manager (GAM), shared L2 cache banks, and shared network-on-chip (NoC) routers between multiple accelerators. These components are all connected by the NoC. Each accelerator node includes a dedicated DMA-controller (DMA-C) as well as scratch-pad memory (SPM) for local storage and a small translation look-aside buffer (TLB) for translating from virtual to physical addresses. In this architecture, we first introduce the GAM, a hardware resource management scheme that provides support for sharing a common set of accelerators among multiple cores. Using a hardware-based arbitration mechanism, the GAM provides feedback to cores indicating the wait time for a particular resource to become available. In addition, a lightweight interrupt system is introduced to reduce the overhead incurred by the OS for handling interrupts, which can occur frequently in an accelerator-rich platform. ARC also provides architectural support allowing for the composition of a larger *virtual* accelerator out of multiple smaller accelerators. On a set of medical imaging applications (our original driver applications at the CDSC), ARC shows significant performance improvement (on average 16X) and reduction in energy consumption (on average 13X) compared to software-based execution on an Intel Xeon E5405 server running at 2GHz.

Although ARC produces impressive performance and energy improvements, it has two limitations. First, it has narrow workload coverage. For example, the highly specialized monolithic accelerator for Deblur cannot be used for Segmentation (refer to the medical imaging pipeline in [31]). The second limitation is that each accelerator has repeated

resources, such as the DMA engine and scratchpad memory (SPM), which are underutilized when the accelerator is idle. To overcome these limitations of ARC, we introduced CHARM [24] (shown in Figure 4.1-B), a Composable Heterogeneous Accelerator-Rich architecture that provides scalability, flexibility, and design reuse. We noticed that all the ARC accelerators for the medical imaging domain could be decomposed into a small set of computing blocks, such floating-point divide, inverse, square root, and 16-input polynomial functions. These blocks are called the accelerator building blocks (ABBs). Our compiler decomposes each compute-intensive kernel (i.e. code region selected as a candidate for acceleration) into a set of ABBs at compile time, and stores the data flow graph describing the composition [24]. The GAM is extended to include an "accelerator block composer" (ABC), which uses data flow graphs at runtime to dynamically allocate and compose available ABBs in order to *virtualize* monolithic accelerators. Therefore, although each composed accelerator is somewhat slower than the dedicated accelerator, we can potentially obtain more copies of the same accelerator, leading to better acceleration results. Our ABC is also capable of providing load balancing among available compute resources to increase accelerator utilization. With respect to the same set of medical imaging benchmarks, the experimental results on CHARM demonstrate improved performance (over 2X better than ARC) and similar gains in energy efficiency [24].

In addition to improving performance/energy efficiency, the CHARM architecture provides better flexibility and wider workload coverage compared to ARC. As shown in [24], by using the same set of ABBs designed for the medical imaging domain, one can compose accelerators in other domains, such as computer vision and navigation, while still achieving impressive speedup and energy reduction. However, it is possible that CHARM misses some ABB types that are necessary for composing functions in a new application domain. To address this issue, we proposed CAMEL [32], which features programmable fabric (PF) to extend the use of ASIC-based composable accelerators and support algorithms beyond the scope of the baseline platform. Figure 4.1-C presents an overview of the CAMEL architecture. With a combination of hardware extensions and compiler support, we demonstrate an average of 12X performance improvement and 14X

energy savings compared to a 4-core 2GHz Intel Xeon E5405 processor across benchmarks that deviate from the original medical imaging domain used for our baseline platform. More details are available in [32].

4.2 Ongoing Research on Composable Accelerator-Rich Platforms

Throughout our work on composable accelerators, and in particular CHARM, we have come to better understand associated performance characteristics. A large part of our ongoing work is to find an optimal design point that better facilitates communication between ABBs. An important limiting factor on the overall performance of a CHARM system is the NoC connecting the various islands to memory resources, and off-chip memory bandwidth. These elements place a hard constraint on the potential performance of a CHARM system. For the purposes of this study, we fix the design of all system components except structures internal to the ABB island, so as to focus on the implications of various design decisions for components internal to an island. Details regarding the evaluated system can be found in Section 4.3.

4.2.1 Anatomy of an ABB Island

In order to evaluate the quality of an ABB island design, we must first categorize the individual components of an island, and understand the design goals of these components. Each ABB island consists of a series of ABBs that serve as the accelerator compute engines, a set of SPM banks that serve as local storage for the ABBs, a DMA engine to coordinate memory traffic between shared memory and the island, and a pair of networks for internal connectivity.

The two networks of this system, which together constitute the elements of greatest cost and greatest impact on performance, are networks connecting the ABB to the SPM (ABB↔SPM), and connecting the SPM memory to the DMA engine (SPM↔DMA).

The design objective of the ABB↔SPM network is to provide low and uniform latency. Latency fluctuations in this network result in stalls in the ABB compute engine. The design objective of the SPM↔DMA network is high bandwidth between the DMA and the individual SPMs. Latency in this network is less critical.

The original CHARM architecture used a crossbar for both the ABB↔SPM and SPM↔DMA networks. While this provides low latency and reasonable bandwidth, crossbars scale poorly. This becomes a concern as the size of the island increases, and the number of ABBs on a single island grows beyond a small number. The ABB↔SPM crossbars in the original CHARM design also allowed for sharing of SPM banks between multiple accelerators. However, sharing in the ABB↔SPM network artificially limits the number of ABBs that can be active at any given time, and introduces complexity to scheduling. To eliminate SPM sharing conflicts and make more efficient use of memory resources, it therefore becomes necessary to have each SPM bank allocated to only one ABB at a time.

4.2.2 Design Space Exploration Parameters

Our design space exploration begins by adjusting the number of islands while keeping the system-wide total number of ABBs fixed, resulting in configurations with different numbers of ABBs per island. In particular, we vary the number of islands from 3-24 while maintaining a total of 120 ABBs in the system. In terms of memory, while the amount of SPM dedicated to a given ABB is fixed by the type of ABB, we vary the number of ports of this SPM from the minimum to two times this quantity. The minimum is defined as the number of ports (in aggregate) that are necessary to allow the ABB to run at peak throughput. Adding SPM ports beyond this minimum keeps the ABB compute engines from observing the impact of bank conflicts.

We also evaluated two potential designs for the ABB↔SPM network: (1) a crossbar that connects the ABB to a set of private SPM banks, and (2) a wider crossbar that connects each ABB to both its most local SPMs and the SPM banks of its neighbors.

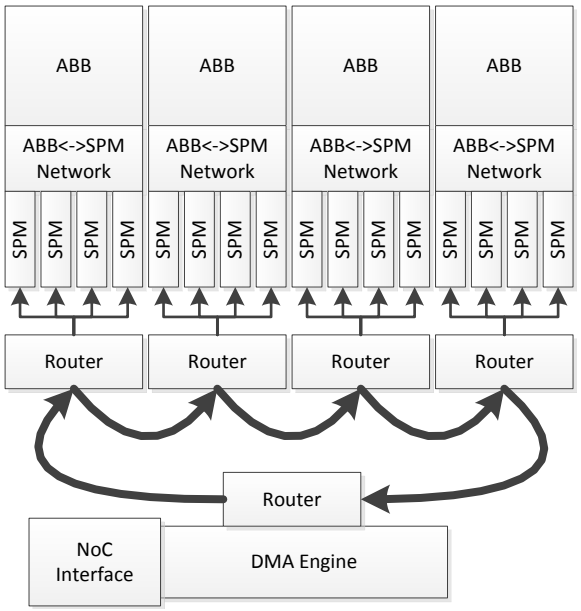


Figure 4.2: Island design using ring for SPM↔DMA network

This second design allows for sharing of SPM banks, and potentially allows for fewer SPM banks to be included, while also allowing for an increase in utilization of SPM resources.

As for the SPM↔DMA network, we evaluated three potential designs: (1) a unidirectional ring network, an example of which can be seen in Figure 4.2, (2) a crossbar connecting the DMA to every SPM bank, and (3) a crossbar connecting all SPM banks to each other as well as to the DMA. Chaining in the second option involves sending data from the source SPM to the DMA, then to the destination SPM. For this reason we refer to the second option as the *proxy crossbar* design. Chaining in the third option involves sending data directly from the source SPM to the destination SPM; we refer to this option as the *chaining-optimized crossbar* design.

4.3 Simulation and modeling details

Our evaluation uses a detailed full-system cycle-accurate simulator based on Simics [19] and GEMS [20]. Our modifications primarily consist of adding simulation support for the ABB types described in [24], as well as modeling the ABC, the compute engines, and

the internal mechanisms of ABB islands. Table 2 in [32] describes the tools we used for modeling timing and power of the various components of our island-based architecture. For the ring network featured in this work, we model area and energy of the routers and links using the Orion [29] tool, estimating link lengths based on island size. Furthermore, the parameters of the simulated system can be found in Table 2 of [24], with the exception that the system in this work is configured with 4 memory controllers (avg. 180-cycle latency @ 10 GB/s) and 120 ABBs (78 polynomial, 18 divide, 9 sqrt, 6 power, 9 sum) with uniform distribution of ABBs among the islands and islands among the processor.

The workloads used for this evaluation are drawn from the Medical Imaging and the Navigation domains, and can be found detailed in our prior work [8, 24, 32]. Our software infrastructure includes a compiler framework [24, 32] for automating the process of analyzing a given accelerator kernel, determining a minimum set of ABBs to cover the kernel, and generating an ABB flow graph to be used for dynamically composing that accelerator.

4.4 Results

For the purpose of discussing the findings of this study, we will consider the simplest possible island construction as our baseline. This island would feature conservative SPM porting, the *proxy crossbar* for the SPM↔DMA network, and no sharing of SPMs.

4.4.1 SPM Sharing

The original CHARM architecture featured partial crossbars between the ABBs and the local SPM banks. The purpose of this was to allow for SPM sharing, and reduce the amount of area devoted to the SPM. Each ABB is connected both to its own memory and to the memory of its neighbors, and some subset of these SPM banks are needed to use this ABB. Since the allocation of a particular ABB requires assigning the shared memory to be temporarily owned by the newly allocated ABB, the act of allocating an ABB renders other near-by ABBs unusable.

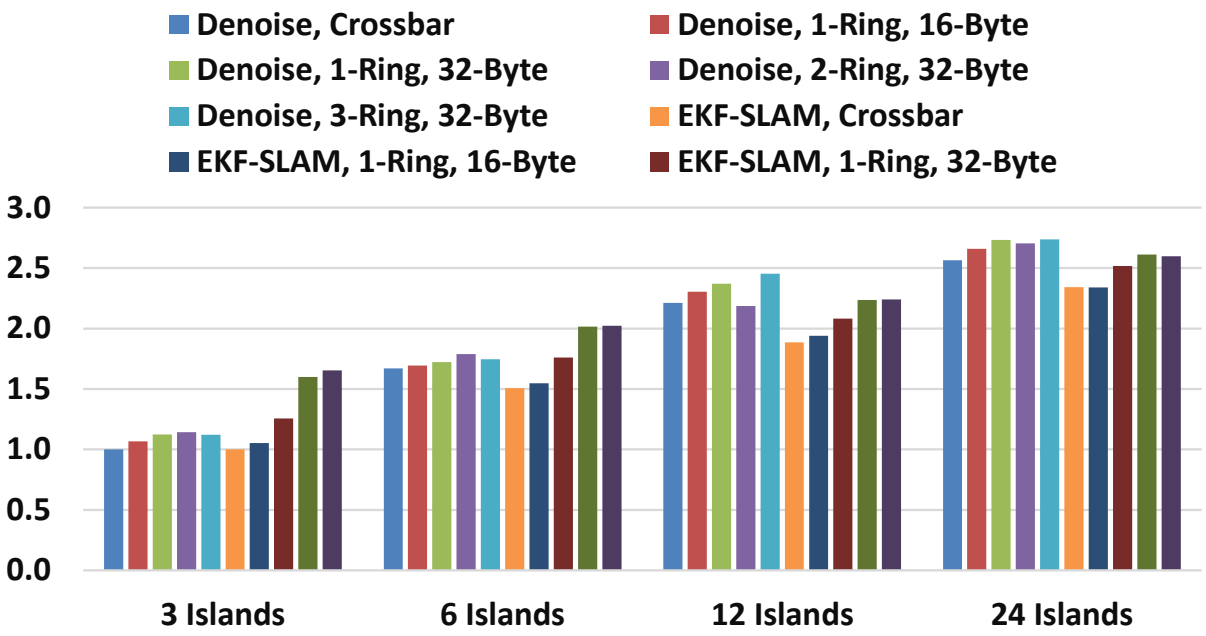


Figure 4.3: Performance impact of utilizing different SPM↔DMA networks while adjusting number and size of ABB islands; normalized to baseline for 3 islands

The given architecture exhibits three main costs: (1) the algorithm that performs allocation must take into consideration side-effects when making an allocation decision, resulting in a considerably more complex ABC, (2) the ABB↔SPM crossbar is larger than it would be were the SPM banks private to a given accelerator, and (3) even if a system has a large number of ABBs, the effective usable amount of ABBs reduces as the degree of sharing increases. This third point is especially critical in a system like CHARM, since the total number of ABBs is heavily dominated by a single type of ABB (polynomial), making it impossible to arrange a sharing scheme that allows for effective utilization of the available compute resources.

While all of the above points are valid arguments against sharing, the most quantitatively concise point against sharing is the increased complexity of the crossbar joining the ABB and SPM banks. Because the SPM banks are individually quite small, the increase in crossbar complexity eliminates area savings. We found that introducing a crossbar that allows an ABB to share the SPM of only its immediate neighbors, a modest amount of sharing, grows the ABB↔SPM crossbar by 3X its original size, and potentially reduces

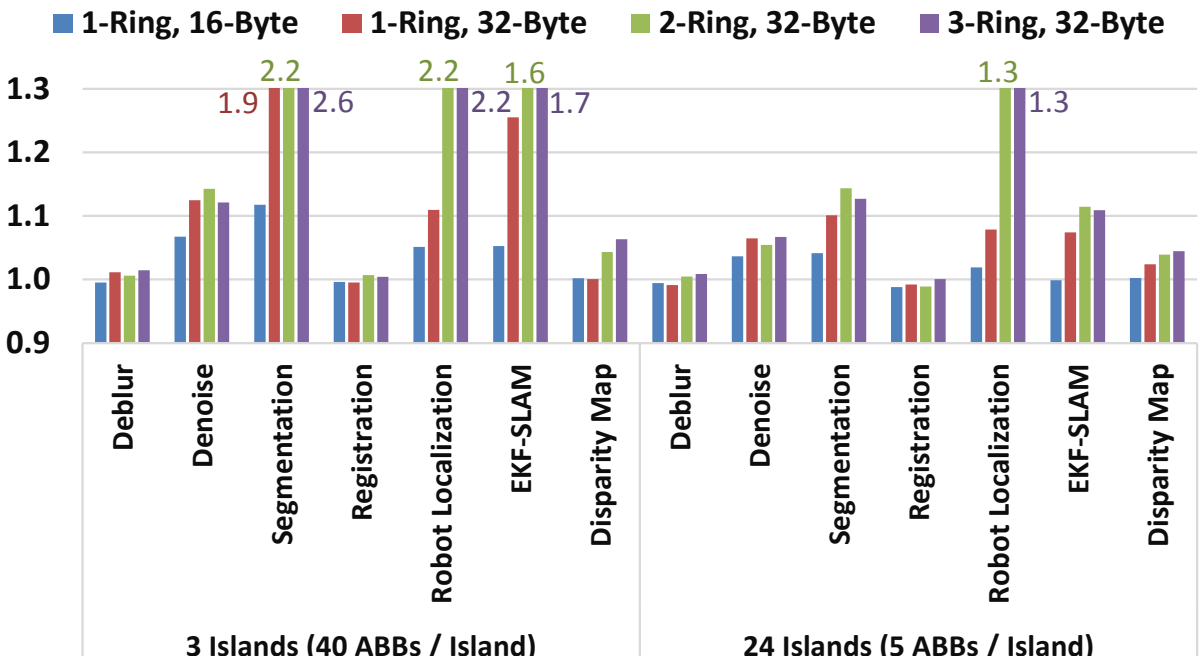


Figure 4.4: Performance of various SPM↔DMA ring networks; shown for 3 islands (40 ABBs / island) and 24 islands (5 ABBs / island); normalized to baseline for respective number of islands

the number of SPM banks by 0.66X. With the volume of SPM banks allocated to a given ABB already constituting about 20% as much area as the ABB↔SPM crossbar (reduced to 7% with sharing), this is a poor trade. For these reasons, we will show no further results regarding SPM bank sharing, and dismiss it as a poor design choice.

4.4.2 Chaining-Optimized Crossbar Topology

A *chaining-optimized crossbar*, as described in Section 4.2.2, is attractive for performance reasons as intra-island communication constitutes a non-trivial amount of the total communication between ABBs. In terms of performance, this crossbar conceptually would be an optimal choice for enabling intra-island chaining. However, we have found that this design does not scale beyond the smallest islands. For large islands, such as those with 40 ABBs, the SPM↔DMA network accounts for over 99% of the total island area, while contributing only modest performance improvements. The reason performance is

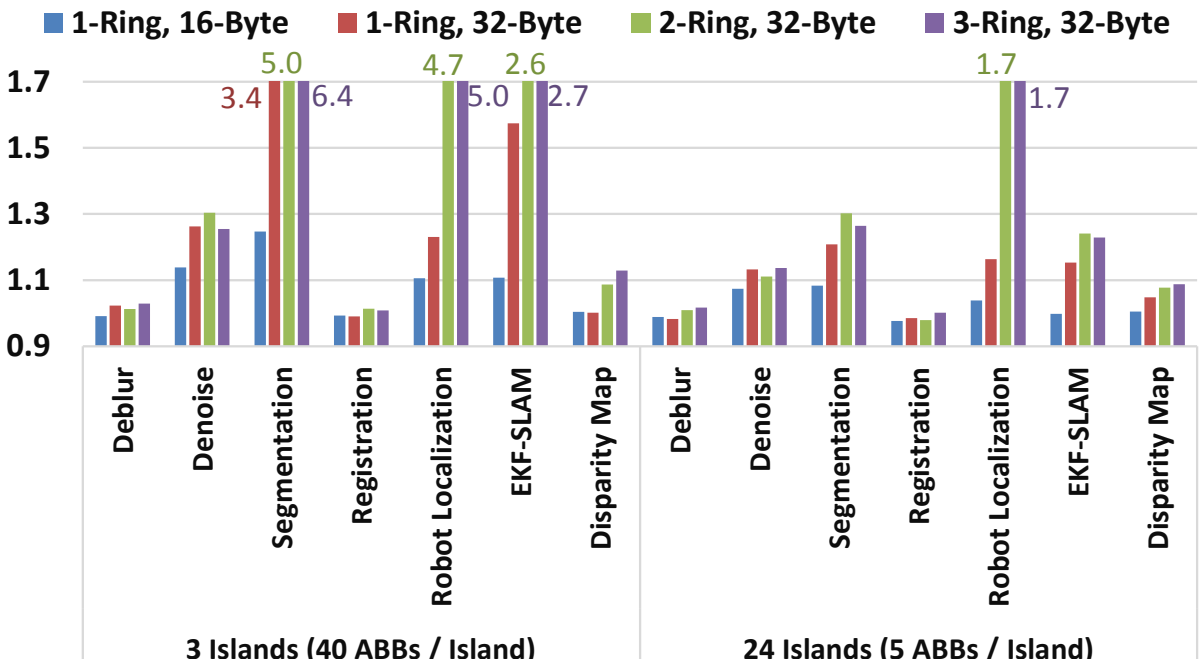


Figure 4.5: Performance per unit energy of selected designs; normalized to baseline for respective number of islands

not improved more significantly is that not only is there extra latency for routing through the large crossbar (which will be discussed in greater detail in Section 4.4.5), but more importantly, **chaining on this network is not observed to constitute the primary performance bottleneck**. At any given time, most ABB pairs are not communicating with one another, which becomes increasingly apparent as the size of islands increases. Therefore, this *chaining-optimized crossbar* topology provides a great deal of connectivity, but severely over-provisions the capacity for chaining relative to what is needed in practice.

4.4.3 Ring Network Width & Ring Count

We have evaluated various bit-widths (16-byte and 32-byte link widths) for the SPM↔DMA network. In the cases of ring networks, we have also evaluated the benefit of adding multiple rings. We have found that a 2-ring network with 16-byte wide channels performs almost identically to a 1-ring network with 32-byte wide channels, and does so with re-

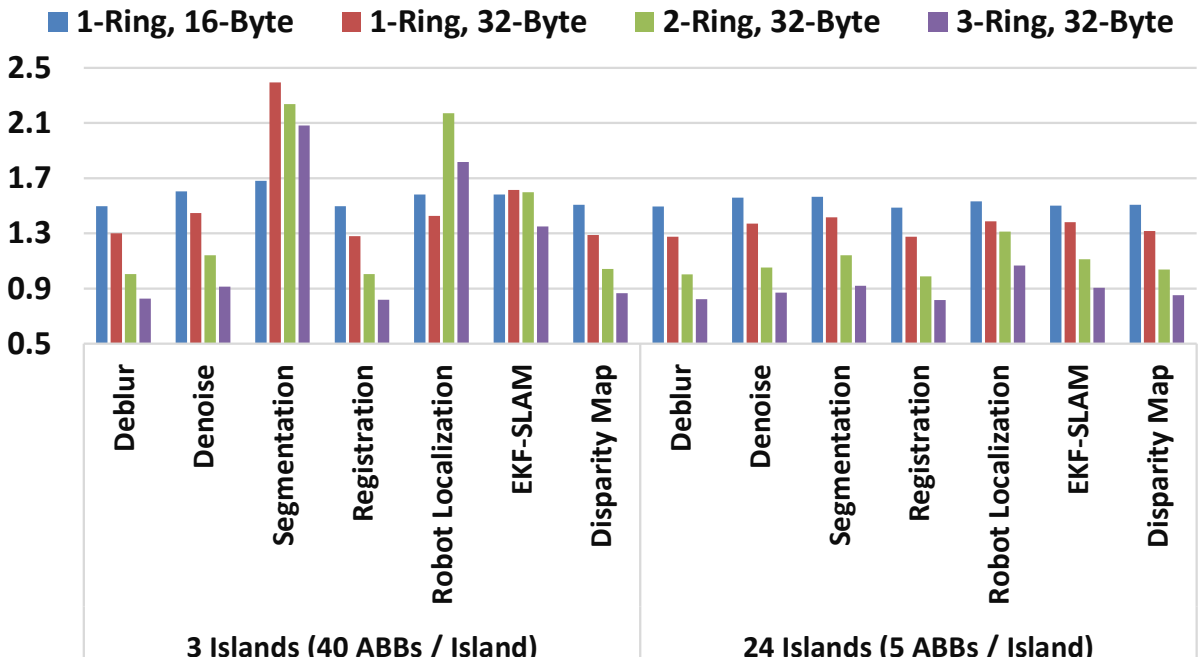


Figure 4.6: Performance per unit area of selected designs; normalized to baseline for respective number of islands

duced ring router complexity. The primary benefit for having a larger number of narrow rings is to make better use of bandwidth in the case where transmitted packets are smaller than the ring width, which would allow for transmission of multiple flits simultaneously. Because the SPM↔DMA network almost exclusively transmits data at the granularity of cache blocks (64-byte) or half-blocks (32-byte), reducing the bit-width below a half-block size does not lead to an improvement. As such, we will not show further results for network configurations with 16-byte link widths except in the case of a single ring, since this data point provides a reasonable distinction from the 32-byte-wide rings.

4.4.4 SPM Porting

Intuitively, bank conflicts on local memory have the potential to constitute a substantial performance shortcoming. For this reason, we have evaluated two SPM porting configurations. The first configuration features exactly the number of ports required to keep the compute engines functioning at peak throughput. The second configuration features twice

this amount, with the intent that bank conflicts can be overcome by over-provisioning SPM bandwidth. We have found that adding ports to SPM banks contributed very little to the total amount of performance, if at all. The primary reason for this is that software has control over the layout of data in the SPM, and even a superficial effort to place data in a favorable SPM bank could eliminate almost all SPM bank conflicts. Over-provisioning of SPM ports therefore only eliminates a negligible amount of conflicts, thereby marginally improving the throughput of the attached ABB. Also, because the ABB performance is not the primary limiting factor for this entire system, as discussed in Section 4.4.5, this marginal drop in ABB performance is of little consequence under most circumstances. Furthermore, increasing ports increases the area and power consumption of SPM banks, along with the size of the ABB↔SPM crossbar (if used). As such, we conclude that designing an island with exact provisioning of SPM ports is not only sufficient, but preferable.

4.4.5 Performance

We have consistently found that one of the primary performance limitations in this accelerator-rich architecture is the interface between the ABB island and the NoC, particularly the NoC bandwidth. This bottleneck is the primary reason for the results shown in Figure 4.3, which displays performance for a selection of benchmarks using several SPM↔DMA network configurations with different numbers of islands (results are normalized to the baseline configuration for 3 islands). In almost all island configurations, the link connecting the ABB island to the rest of the system has been fully utilized. As ABBs are distributed across more islands (i.e. fewer ABBs per island), there is likely more inter-island communication, which causes performance to be more heavily dominated by the NoC. For benchmarks with small amounts of ABB chaining (e.g. Denoise), compared to benchmarks with more ABB chaining (e.g. EKF-SLAM), inter-island communication is less probable and constitutes a smaller portion of the total traffic on the NoC. As such, when the number of islands is increased, benchmarks with less chaining exhibit larger improvements in average performance across all the SPM↔DMA network

configurations.

Figure 4.4 shows the performance impact of adjusting the topology of the SPM↔DMA network. As shown, the majority of ring configurations outperform the *proxy crossbar* (i.e. the baseline to which the results are normalized), though the impact is reduced as the total number of islands increases. The crossbar also exhibits particularly poor performance for cases with large amounts of ABB chaining, such as with the Segmentation, Robot Localization, and EKF-SLAM benchmarks. Unlike a crossbar, the ring network presents a more scalable solution, and exhibits bandwidth provisioning that is easier to fine-tune.

4.4.6 Energy & Energy Per Computation

Figure 4.5 shows performance per unit energy for several configurations. This shows the efficiency with which we are able to achieve a given performance point. This graph clearly shows that over-provisioning interconnect resources allows for more energy-efficient operations. The reason for this is because a more robust interconnect allows for higher performance, but uses very similar power per bit-transferred. Also, comparing the 24-island configuration with the 3-island one reveals that having more islands results in smaller efficiency gains as the interconnect strength is increased. This is to be expected since performance is more heavily dominated by the NoC interface when the number of islands increases (as described in Section 4.4.5).

4.4.7 Area & Compute Density

The SPM↔DMA network accounts for 16-40% of the total island area for a ring network (depending on the bit-width of links and the number of rings), and 44-50% of the total island area for crossbar networks for large islands. For this reason, under-provisioning this resource, and thus maximizing network utilization, allows for an increase in compute density, even though performance suffers. Figure 4.6 shows this clearly, with compute density (i.e. performance per unit area) dropping as network resources are added to

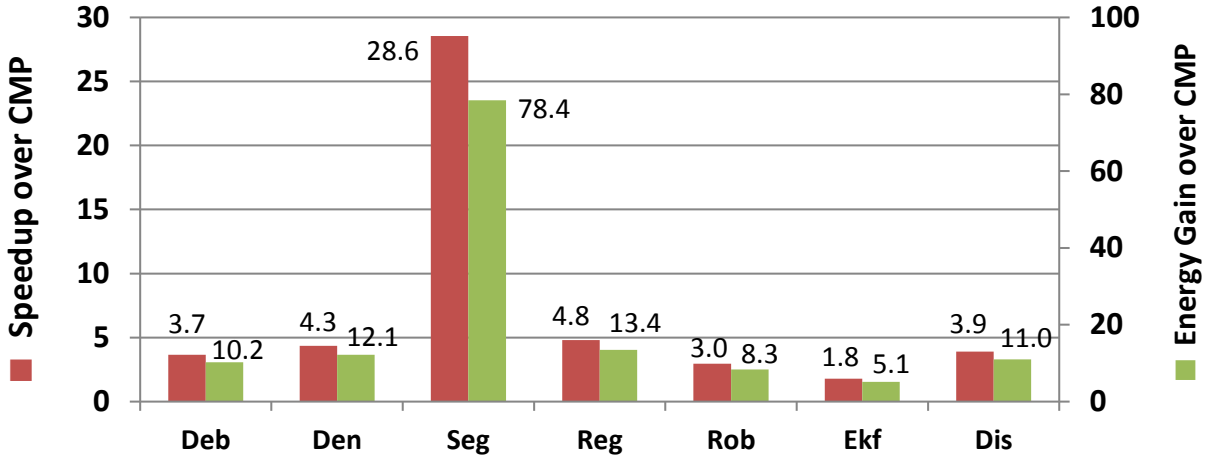


Figure 4.7: Performance and energy gains of "best" accelerator-rich design configuration over chip multi-processor (CMP)

increase the system's performance. Small networks see high utilization, and even limit accelerator throughput severely in some cases. However, due to the NoC interface bottleneck described in Section 4.4.5, there is little justification for enlarging the SPM↔DMA network capacity very much beyond the bandwidth cap instituted by the NoC.

4.4.8 Comparison to Chip Multi-Processor (CMP)

Based on our design space exploration, the configuration that performs the best in terms of average performance, energy efficiency, and compute density is the 24-island design with a 2-ring SPM↔DMA network of 32-byte links, and with no SPM sharing and no over-provisioning of SPM ports. In Figure 4.7, we compare this design to a 12-core 1.9 GHz Intel Xeon E5-2420 processor, where on average, our accelerator-rich design achieves 7X speedup and 20X energy savings. Comparing to the 4-core CMP used in [32], we see 25X speedup and 76X energy savings. Furthermore, this design maintains an average ABB utilization of 18.5% with a peak utilization of 43.5%.

CHAPTER 5

Composable Accelerator-rich Microprocessor Enhanced for Adaptability and Longevity

In the CHARM architecture, discussed in Chapter 3, any given kernel consisted almost entirely of a small set of common components. These components, described in Chapter 3 as poly, divide, and pow ABBs, were used in all computations, and in great number. As more workloads are considered in the design of a single platform, the portion of ABBs that are shared heavily begins to reduce, as new ABBs that are specific to a particular workload need to be introduced to maintain program coverage, and enable acceleration of more regions. This begins to cause a problem when it becomes necessary to sacrifice highly utilized ABBs to add low utilization ABBs for the purpose of allowing an unusual kernel to be accelerated that otherwise could not be accelerated.

To help curb this problematic trend, the Composable Accelerator-rich Microprocessor Enhanced for Adaptability and Longevity (CAMEL) [32] architecture is introduced. CAMEL adds to a CHARM system a small region of programmable fabric for the purpose of implementing these small seldom used accelerators, while limiting the impact that including these ABBs has on system-wide accelerator utilization. The type of programmable fabric CAMEL features is FPGA, though this is not necessarily the only possible design. While it is understood that an FPGA implementation of a compute engine instead of ASIC is on average 40x larger, 3.2x slower, and 12x less power efficient [33], the advantage of including an FPGA allows for a sharp increase in the degree to which ASIC resources can be utilized, and thus yields system-wide performance improvement and energy savings.

The main contributions of this work are the following:

- **Compiler and Runtime Framework to Support ASIC and PF Allocation** - Our compilation framework generates a task flow graph of interconnected building blocks for a given kernel; it can also perform platform-aware partitioning of the task flow graph into subgraphs that can be accommodated by on-chip resources; at runtime, our resource manager uses these graphs to compose accelerators by allocating either ASIC- or PF-based building blocks.

- **Slack Analysis and Rate Matching** - Our compiler statically identifies imbalance in the task flow graph, and compensates for the computational slack in shorter paths by allocating extra buffer space; our hardware reduces PF performance overhead through *rate-matching*, where it instantiates multiple PF-based building blocks to collectively match the ASIC design throughput.

- **Design Space Exploration** - We demonstrate the enhanced flexibility from our approach through analysis on four distinct application domains, examining the benefits our approach provides to design extensibility and longevity; while we analyze our results on one candidate architecture for accelerator composition, our techniques are more generally applicable to other composable architectures.

5.1 Microarchitecture of CAMEL

The CAMEL architecture uses a combination of software and hardware components to improve flexibility and longevity. The hardware components are responsible for the actual accelerator composition, where the virtual accelerators, or loosely-coupled accelerators (LCAs), are dynamically constructed using either the available accelerator building blocks (ABBs) in ASIC or ABBs that have been instantiated in PF. While our contributions in the CAMEL architecture are generally applicable to composable architectures, in this paper we implement our techniques and analyze results on the CHARM architecture [24]. An overview of the CAMEL microarchitectural components is presented (not to scale)

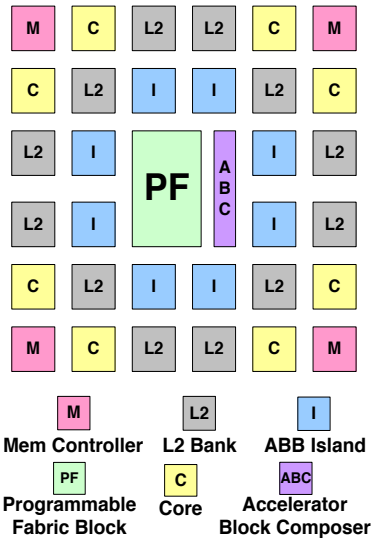


Figure 5.1: CAMEL Microarchitecture

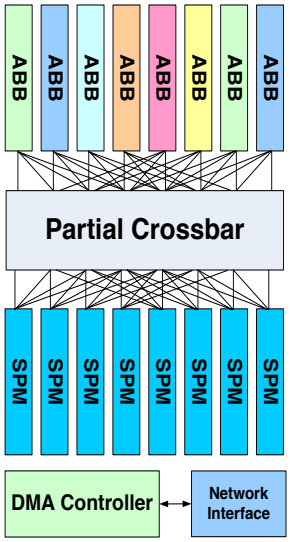


Figure 5.2: ABB Island

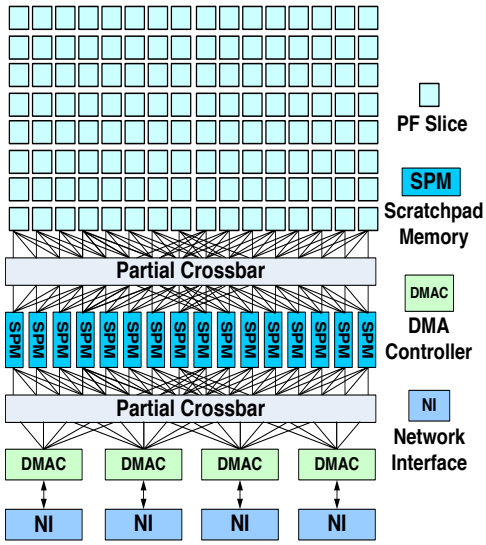


Figure 5.3: Programmable Fabric

in Figure 5.1. This figure consists of a set of cores with private L1 caches, shared L2 cache banks, and the following specialized CAMEL components: (1) ABBs grouped into a series of islands (shown as "I"); (2) accelerator block composer (ABC) responsible for accelerator composition, PF assignment, and CAMEL resource arbitration; and (3) PF (for additional ABBs).

5.1.1 ABB Islands

Figure 5.2 shows the internal structure of an ABB island; in this sample figure there are 8 ABBs, 8 scratchpad memory (SPM) banks, and 1 multi-channel DMA controller (DMAC). Each ABB has access to only 4 of the SPM banks using a partial 8x8 crossbar [34]. These SPMs are in turn connected to the multi-channel DMAC. The numbers and types of the ABBs are determined using software-driven design-space exploration, and the ABBs of a given type are distributed evenly across the islands in a round-robin fashion.

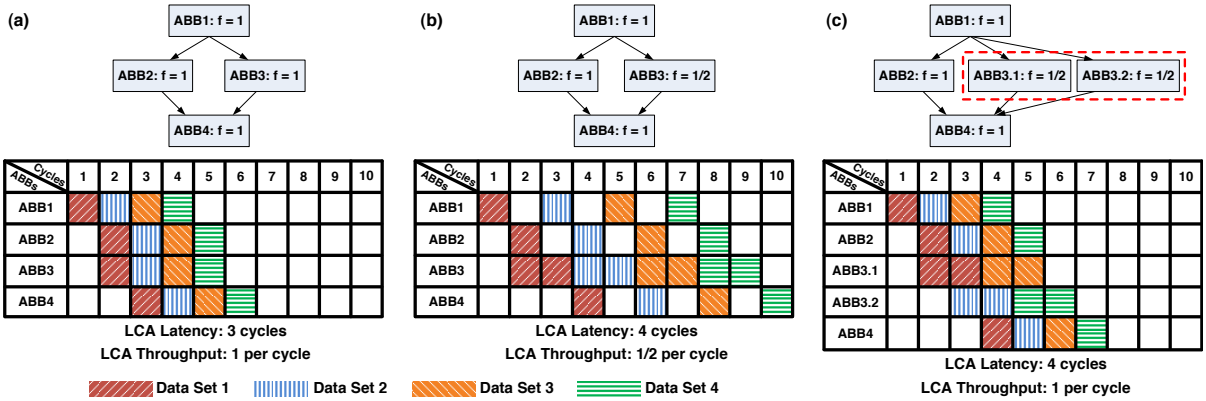


Figure 5.4: Motivational Example of Applying Rate-Matching on PF

5.1.2 Programmable Fabric

The PF is used for hosting the ABBs required by new applications (in new or existing domains). The internal design of the PF in CAMEL is shown in Figure 5.3. It consists of PF slices, 16 SPM banks, 4 DMACs, 4 network interfaces (NI), and 2 crossbars: one to connect a selected set of PF slices to SPMs and one to connect SPMs to DMACs. Although a monolithic PF presents challenges in its shared usage (i.e. ports, NoC congestion, etc.), it accommodates ABBs of any size and avoids performance hits due to static partitioning of resources. The main advantage of using a PF is its reusability and run-time reconfigurability. However, ABBs implemented on the PF are less area- and power-efficient, and have lower performance compared to ABBs implemented on ASIC. While the area and power issues are largely technology-dependent, we address energy consumption and performance using hardware techniques that compensate for the mismatch in computation speed.

When a virtual LCA is invoked, software sends to the ABC an encoded task flow graph representing the LCA's functionality. Nodes in this graph represent functionalities of individual ABBs, while edges represent data transfers. This functionality is executed in a pipelined fashion, with each ABB in the graph communicating with others by means of bulk transfers from/to its local SPM to/from remote SPMs or memory. If a PF-implemented (presumably less efficient) ABB is on the critical path, it can negatively impact the performance of the entire LCA. Figure 5.4 exemplifies this scenario and how

rate-matching helps. In this figure, the same task flow graph is instantiated for three different hardware allocation scenarios, and we see how four independent data sets (illustrated by different shading patterns) would flow through the connected ABBs. As Figure 5.4-a shows, when all ABBs are operating at the same frequency (e.g. f = 1), the LCA they compose will have that same throughput. However, as shown in Figure 5.4-b, if one of the ABBs is slower than the others (e.g. ABB3 has f = 1/2), this ABB becomes a bottleneck and the other ABBs are forced to stall. This results in the LCA as a whole progressing at the rate of this single slow component. Since the ABBs allocated in the PF typically have less throughput than ASIC ones, the inclusion of a PF-based ABB could result in such a bottleneck.

To address this, CAMEL allocates multiple copies of the slower ABB to bring the aggregate throughput of the collection of slow ABBs up to match that of the faster ABBs. This is referred to as *rate-matching*, and is shown in Figure 5.4-c. Provided there are sufficient PF resources for multiple ABB instantiations, this technique interleaves independent data sets between the duplicated PF-based ABBs and allows for the LCA to make more efficient use of the ASIC-based ABBs. As throughput is increased, the other ABBs and overall system components are left idle for a shorter period of time, thereby reducing static energy consumption. Although dynamic power is slightly increased, dynamic energy remains constant and so overall energy consumption is reduced. Thus, while rate matching not only improves performance and resource utilization, it also improves energy efficiency. The implementation of this technique is described in Section 5.1.3.

5.1.3 Runtime PF Allocation

The ABC performs PF-based ABB allocation using the algorithm shown in Figure 5.5. It receives information on the available space on the PF, along with the list of available ASIC ABBs and the LCA task flow graph. Using these it determines what ABBs to allocate in PF. To achieve the best allocation, it starts with the minimum configuration as a feasibility test; if the minimum currently cannot fit, it temporarily keeps the task

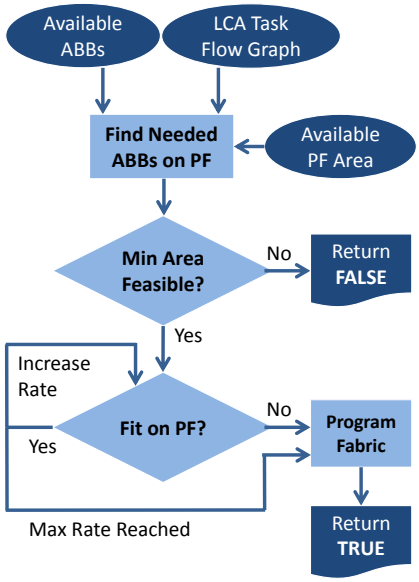


Figure 5.5: PF Allocation Algorithm

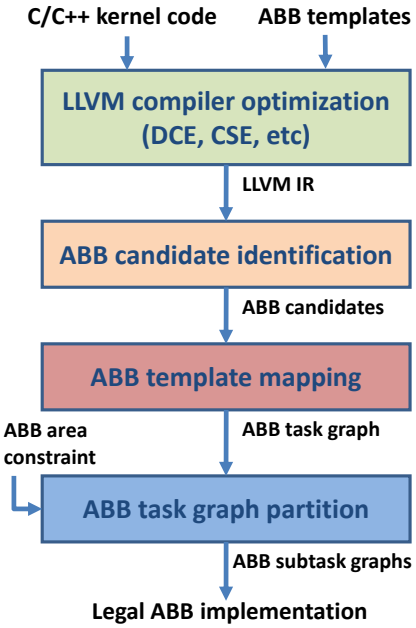


Figure 5.6: Compiler Framework

until enough space is available on PF. If the minimum cannot be implemented at all, the ABC informs the requesting core of the failure to implement. If the feasibility check passes, the ABC attempts rate-matching: it iteratively increases the PF-based allocation of critical ABBs (i.e. those on the critical path of the task flow graph) until either no space is left on the PF or the best rate-match is achieved.

5.1.4 Compiler Support

An overview of the CAMEL compiler framework is shown in Figure 5.6. Given information on ABB types to potentially use, the compiler is responsible for mapping a given program kernel to a set of those ABB types, producing a data flow graph (i.e. task flow graph) whose nodes are ABBs and whose edges are data transfers. The algorithm used is similar to that described in [35]. Provided supplemental information on the available ASIC ABBs and PF for a given platform, the compiler can also determine if a kernel being mapped is too large for the total number of ASIC ABBs combined with the total PF. In these cases, the kernel's task flow graph is partitioned into the fewest number of

regions such that allocation is possible. Partitioning is done along regions of the graph such as to minimize data transfer between partitions, and temporary storage is allocated to store intermediate data. The partitioned regions become subgraphs that can then be run sequentially. An example of this is shown in Section 5.3.4. After a mapping solution exists, addressing for the local SPM of each ABB is calculated. Part of this calculation is an optimization for graphs that feature multiple paths of different lengths (i.e. slack) between a pair of nodes. Once this slack is identified, computational correctness is ensured by allocating extra buffer space along shorter paths. By avoiding stalls, this method allows for higher ABB utilization and overall throughput along all paths.

5.2 Evaluation Methodology

5.2.1 Tool Chain

In order to evaluate this architecture, we extended Simics [19] and GEMS [20] with the cycle-accurate models needed by CAMEL. Table 5.1 shows the simulation parameters used. We also implemented a complete tool-chain for generating simulator models starting from C-based kernel code. Table 5.2 shows the additional tools used for acquiring accurate timing and power values for these models. Furthermore, the compiler framework was implemented in LLVM [36], and has an average compilation time of 6.1 seconds per kernel for our benchmarks.

5.2.2 Domains

In this work, we target the four application domains described below. These four domains not only provide coverage of real-world applications with interesting computational demands, they also represent classes of applications that are algorithmically diverse in nature. Table 5.3 shows the numbers and types of ABBs used for accelerating each domain using one set of accelerators. Note that by *one set of accelerators* we mean as many ABBs as it would take to instantiate one of each virtual LCA in the domain. In

Table 5.1: Simulation Parameters

Parameter	Value
Main Memory	Latency: 280 cycles, bandwidth: 10 B/cycle per controller
L2 Cache	8MB, 8-way set-associative, 32 banks, latency: 10 cycles
Coherence Protocol	Shared banked L2-cache, L2: MOSI, L1: MSI
Network Topology	4x8 MESH, latency: link 1 cycle & router 5 cycles, bandwidth: 72 B/cycle per link
ABB Islands (Base)	16 islands; 14 ABBs and 14 4KB SPMs per island

Table 5.2: Tools for Timing and Power Models

Tool	Purpose
Xilinx Vivado Design Suite [37]	Accelerator high-level synthesis
Synopsys Design Compiler (32nm) [22]	ASIC synthesis (power, performance)
Xilinx ISE [38]	PF synthesis (performance)
Xilinx Virtex 6 XPower Estimator [38]	PF power analysis
CACTI [28]	Cache and scratchpad modeling
Orion [29]	NoC power and area
McPat [23]	Core and cache power analysis

our experiments, we have used four sets of accelerators.

5.2.2.1 Medical Imaging (Med)

Medical imaging is an important tool for diagnosis and treatment. Because of the high volumes of data and high computational demands, the algorithms cannot be easily used in real-time clinical diagnosis, making them excellent candidates for acceleration. The medical imaging pipeline includes denoising, deblurring, fluid registration, image segmentation, and compressive sensing for reconstruction. These algorithms and their acceleration

strategies are described further in [31].

5.2.2.2 Commercial (Com)

We have selected three applications from the PARSEC [39] suite to represent the commercial domain: BlackScholes, Streamcluster, and Swaptions. These applications solve partial differential equations, online clustering problems, and probability distribution estimations.

5.2.2.3 Vision (Vis)

Computer vision is a compute-intensive domain with inherent parallelism that makes it ideal for streaming-data style of acceleration. Two main categories of applications in this domain are feature extraction, for which we include implementations of SURF from OpenCV [40] and LPCIP from MRPT [41], and image processing, for which we include the Texture Synthesis application from SD-VBS [42]. These applications provide a variety of computation including complex matrix-based, trigonometric, log-polar, and gradient histogram computations, with fluctuating memory usage.

5.2.2.4 Navigation (Nav)

Navigation is a compute-intensive, AI-related domain that aims to achieve high levels of situational awareness. We include EKF-SLAM from MRPT [41], along with Robot Localization and Disparity Map from SD-VBS [42]. These applications provide diverse computation in the form of partial derivatives, covariance, spherical coordinates, probabilistic models, particle filters, search for minimal sum of absolute differences, etc.

5.2.3 ABB Characterization

The ASIC ABBs for our system have all been synthesized with a frequency of 1GHz and an initiation interval (II) of 1. Although the PF ABBs also have II's of 1, they have

Table 5.3: ABB Types, PF Synthesis, Domain Numbers, and Func.

ABB Type	FPGA Slices	Power (mW)	Freq (GHz)	ABBs per Domain Med	Com	Vis	Nav	Functionality Description
poly	3536	571	1/4	47	95	143	167	16 I/O Polynomial (floating pt.)
sqrtf	672	176	1/3	2	1	1	3	Square root (floating pt.)
divf	255	84	1/3	6	5	6	7	Divide (floating pt.)
powf	672	176	1/3	1	3	1	0	Power function (floating pt.)
logf	672	176	1/3	0	3	0	0	Log base e (floating pt.)
rr1D	25	2	1/2	0	2	0	0	Random read in 1 dimension
rr2D	90	70	1/2	0	0	2	0	Random read in 2 dimension
rr3D	145	91	1/2	0	0	73	0	Random read in 3 dimension
rw1D	25	2	1/2	0	1	0	0	Random write in 1 dimension
selff	58	54	1/2	0	4	50	0	MUX (float inputs, float select)
selfi	57	54	1/2	0	3	4	0	MUX (float inputs, int select)
selif	27	84	1	0	4	8	0	MUX (int inputs, float select)
selii	30	85	1	0	1	0	0	MUX (int inputs, int select)
sum	134	77	1/2	0	1	8	1	Accumulate a vector
castfi	94	32	1/4	0	0	43	0	Cast float to integer
castif	108	35	1/4	0	0	1	0	Cast integer to float
mod	255	84	1/3	0	0	2	0	Modulo
min	65	54	1/2	0	0	3	0	Find minimum value in vector

Table 5.4: Power and Area for CAMEL Base Platform

Unit Type	Num. Units	Power per Unit (mW)	Area per Unit (um^2)
ABC	1	66.00	8383
poly ABB	188	6.65	362570
sqrtf ABB	8	9.49	368819
divf ABB	24	0.52	15117
powf ABB	4	9.49	368819
SPM	240	17.60	40773
DMAC	20	0.59	10071
L2 Bank *	32	148.97	881990
Core *	1	686.46	9868400
NoC *	1	4923.52	557978

* Power varies with execution; average value is shown.

Table 5.5: Number of ABBs and PF Slices in CAMEL-x%

	CAMEL-0%	CAMEL-10%	CAMEL-20%	CAMEL-30%	CAMEL-40%	CAMEL-50%
# ABBs	224	192	168	148	128	108
# PF Slices	0	2466	4935	7404	9873	12342

different operating frequencies depending on their type. Table 5.3 details the results of synthesizing the various ABB types for a Xilinx Virtex6 FPGA, along with the numbers of ABBs needed by the four domains and the functionalities of the ABBs. Note that the ABB granularities and functionalities have been determined according to a domain-space optimization primarily for Med, which is the base domain of CAMEL in our case studies (see Section 5.2.4), with additional ABB types added as needed.

5.2.4 Case Studies

For the purposes of this paper, we consider the cases of running single benchmarks, where accelerator needs are known. As such, PF reconfiguration can be done statically, and so reconfiguration time is excluded from all results. In our experiments we have considered the following cases, each representing a different class of accelerator-based architectures:

GPU is a Tesla M2075; performance measures consider computation only (not data transfer time).

LCA-ASIC is an accelerator-rich platform where all LCAs are monolithic and ASIC-based [8].

LCA-FPGA is an accelerator platform where all LCAs are monolithic and FPGA-based [8].

CHARM is a composable accelerator-rich platform with Med base domain and no PF [24].

CAMEL-x% is the CAMEL architecture with Med base domain and "x" percent of the total ABB area substituted (by removing "x" percent of ABBs of each type, maintaining even ABB distribution across islands) for equivalent area of PF; x ranges 0%-50%.

The power and area values modeled for the CAMEL-0% base platform can be found in Table 5.4, where the total area of the chip is 122 mm^2. To determine the number of PF slices that can fit in CAMEL-x%, we have used the die area size of Virtex6 (measured by taking X-ray photos) and have estimated 2955 um^2 for each slice in 32nm. Table 5.5 shows numbers of PF slices and remaining ASIC-based ABBs for each CAMEL-x% case. Note that ABB types vary in both area and quantity – the distribution shown corresponds specifically to our platform. As PF slices are linearly increased for the CAMEL-x% cases, different numbers of various types of ABBs are removed to make room for the PF area, so the total number of remaining ABBs may not decrease linearly.

5.3 Experimental Results

In this section, we present and discuss our simulated results. Although our Simics+GEMS framework simulates an Ultra-SPARC-III-i 1GHz processor (running Solaris 10), we conservatively measure our performance gains in terms of a wall-time-based comparison to fully parallelized runs on a 4-core 2GHz Intel Xeon E5405 processor. When there are insufficient accelerator resources to run a benchmark, we fall back to running on the CPU, and thus exhibit no benefit.

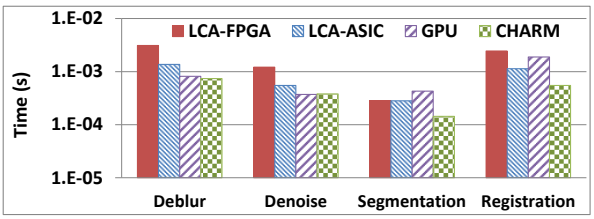


Figure 5.7: Performance Comparison between Acceleration Schemes

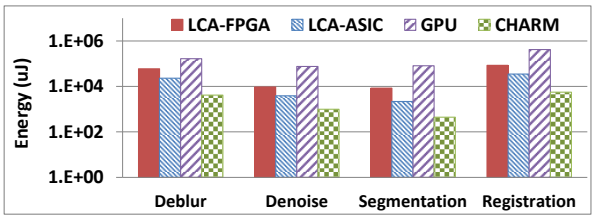


Figure 5.8: Energy Usage Comparison between Acceleration Schemes

5.3.1 Comparison Between Acceleration Schemes

Figure 5.7 and Figure 5.8 compare four accelerator-based architectures running benchmarks from the Med domain. As it features domain-specific acceleration, CHARM (i.e. CAMEL-0%) outperforms by 2.1X and saves energy by 93X compared to the power-hungry GPU. Furthermore, with its ability to load-balance and dynamically virtualize LCAs, CHARM on average outperforms LCA-FPGA by 3.5X and LCA-ASIC by 1.8X, resulting in energy savings of 14.5X and 5.1X, respectively. For an optimal design, we would want the performance and energy usage of CHARM with the adaptivity of GPUs and FPGAs. We show next how CHARM is made adaptive for greater performance and energy savings across domains.

5.3.2 Effect on Domain-Span

To evaluate CAMEL support of domain-span, we have used the Med base domain (for ASIC ABBs) and chosen three other target domains: Com, Vis, and Nav (as mentioned in Section 5.2). In all of these experiments, we have kept the overall area constant by removing 0%-50% of the ASIC ABB area in increments of 10% (maintaining even distributions of ABB types across islands) and adding PF slices equivalent to the removed area.

Figure 5.9 shows the aggregate speedup and energy savings for all four domains, while Figure 5.10 shows the average speedup and energy savings of each domain vs. software-only versions of the implementations. Since most of these new applications are unable to

run on the base without the PF (i.e. exhibit 1X as they fall back to running on the CPU), the aggregate speedup of CAMEL-0% (i.e. CHARM) across all benchmarks is relatively low. As seen in Figure 5.10-a, the Med applications, for which this base was originally optimized, see performance improvement with the addition of a small amount of PF, followed by a decrease in performance as more PF is added. This is intuitively correct, as the platform considered was provisioned with the ASIC-based ABBs designed specifically for accelerating Med applications. A small amount of PF (10%-20%) provides adaptivity for higher load balancing and resource utilization for each individual benchmark, while larger amounts of PF begin to starve the system of the improved performance efficiency of the ASIC ABBs. However, even the small performance improvement initially seen with the addition of PF is not enough to counterbalance the reduction in power-efficiency as ASIC ABBs are replaced by PF. As a result, we see an initially small decrease in energy savings for CAMEL-10% and CAMEL-20%, followed by a larger decrease for CAMEL-30% and onward.

For Com (Figure 5.10-b), no applications can be implemented without PF because they all require a variety of new ABB types that do not appear on the base platform (refer to Table 5.3). As PF is added, these ABBs can be instantiated and rate-matched, resulting in large performance gains and energy savings. With Vis (Figure 5.10-c), we see behavior similar to that of the Com applications. For Nav (Figure 5.10-d), we see an initial speedup even without the PF because this domain shares a lot of the same ABBs as Med, allowing some benchmarks to be minimally implemented on the base platform. As we initially increase PF, we are able to instantiate the missing ABBs and run all benchmarks, resulting in increased average gains in both performance and energy. However, similar to the trends we see with CAMEL-10% and -20% for Med, as more ASIC is replaced by PF for Nav, performance continues improving slightly, yet energy savings begin dropping (e.g. CAMEL-30% and onward).

In summary, as ASIC ABBs are removed and replaced by PF, more useful ABBs become available and rate-matching takes effect. This translates into better adaptivity, and often times higher performance and energy savings for new domains. While these

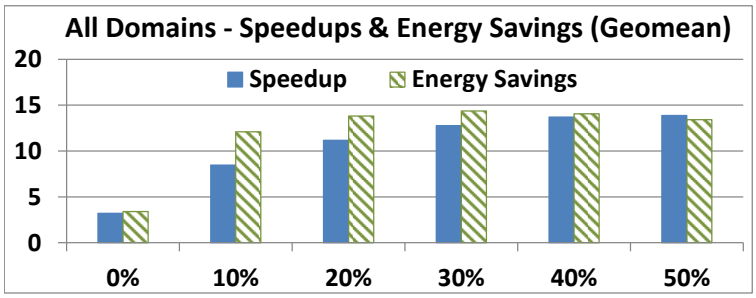


Figure 5.9: Geometric Mean of All Speedups and Energy Savings

trends depend on the specific workload you are considering, as intuitively suspected, the less similar a workload is to the base domain of the platform, the more useful the PF. As with the law of diminishing returns, however, increasing the PF past a certain point starts reducing the improvements because the system is now removing too many of the useful ASIC ABBs and replacing them with their equivalent PF-based ones. We see this turning point with ∼30% PF for domains similar to the base (e.g. Nav) and ∼50% PF for other domains (e.g. Com and Vis).

5.3.3 Effect on Domain Longevity

In order to evaluate the longevity of the base domain, we have added a new application to Med: compressive sensing magnetic resonance (CS_MR) [43]. This application needs one additional ABB, namely the "sum" ABB, which is not found on the Med base domain of CAMEL. This "sum" ABB is one that accumulates the values of a given vector, and is used to implement the internal FFT engine of CS_MR. The speedup result for CS_MR is shown in Figure 5.11. CS_MR does not need many of the ASIC-based ABBs on CAMEL, so as more PF slices are provided, it can use them to implement more "sum" ABBs, allowing it to instantiate more of its virtual LCAs and achieve more speedup.

5.3.4 Graph Partitioning for Lower-Capacity Hardware

As described in Section 5.1.4, it is sometimes the case that a benchmark demands a massive LCA for a large kernel and requires more resources than are available on CAMEL,

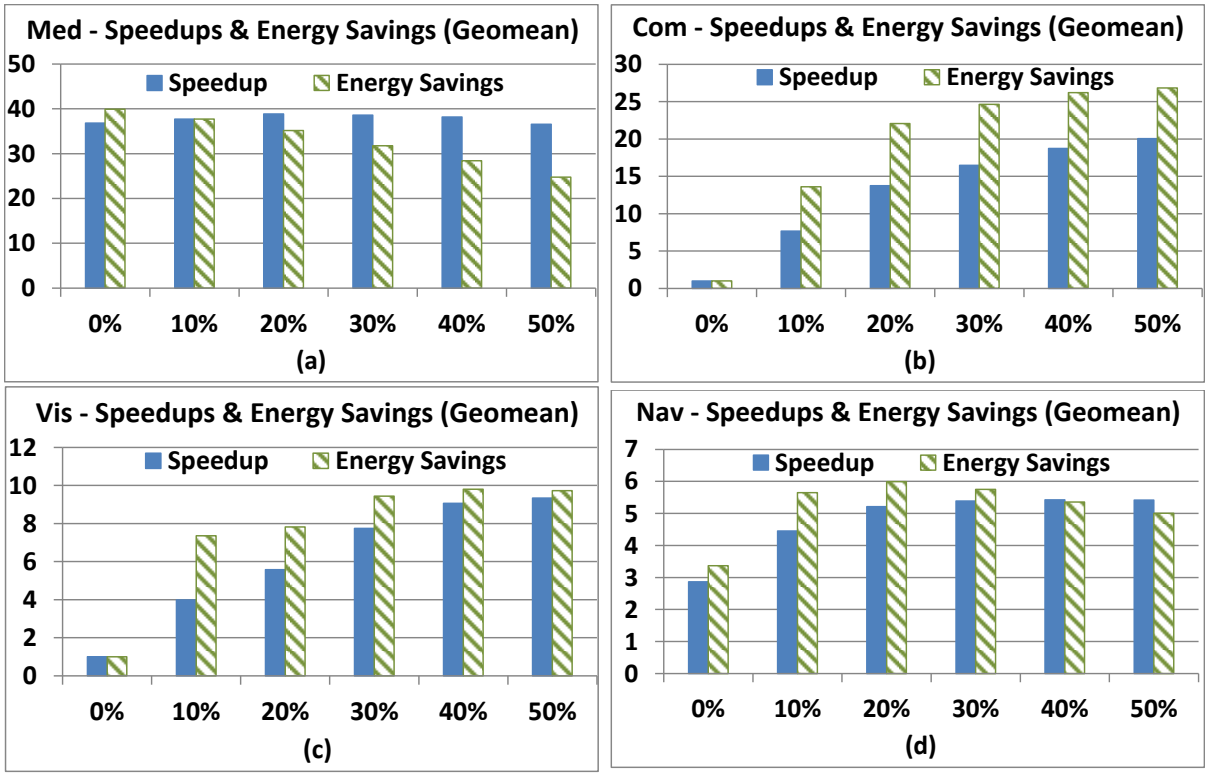


Figure 5.10: Geometric Mean of Speedup and Energy Savings for Each Domain

even with PF. Benchmarks like Texture Synthesis, Swaptions, Stream Clusters, and SURF contain kernels that can never be implemented in their original form. To overcome this, our compiler partitions the task flow graph of each of these kernels into a number of subgraphs that can each fit on CAMEL-x% (e.g. Texture Synthesis requires 6 partitions for CAMEL-50%). Figure 5.11 shows the result of accelerating Texture Synthesis as an example after applying this graph partitioning technique, where we are able to achieve up to 11.96X speedup.

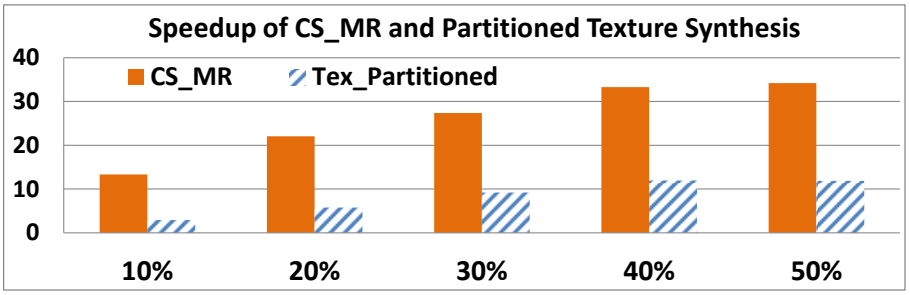


Figure 5.11: Domain Longevity and Graph Partitioning

CHAPTER 6

AIMing to Topple the Memory Wall

While accelerators are able to bring a great deal of computational power to bear on a problem, providing data to these compute engines continues to be a problem. This is particularly problematic when considering workloads that must stream over a large volume of data. The reason for this is that, while resources that can be dedicated to on-chip networks scale well with regard process technology, the pin-out of a chip improves relatively slowly. The 2012 ITRS Road Map [1] predicts a 1.48x increase in the number of pins over an 8-year time-frame and based on Moore's Law there will be at least a 16x increase in the number of transistors available to implement compute engines over the same time period. The impact that this has on the design of a system is obvious, an impact referred to as "Hitting the Memory Wall" [44], which describes the inevitable and pervasive out-pacing of memory devices by compute engines, due simply to manufacturing capability.

This shortcoming however manifests itself in very specific places: chip boundaries. Even conventional memory systems in modern machines feature memory modules collectively feature bandwidth that is many times the ability of the central processors ability to make use of this bandwidth. This observation was made by prior works as well, that distributed compute engines into memory devices directly, so as not to have to pay the cost of moving data across the die package boundary of the central processor [45–48]. While the performance of these systems are very high, they are highly invasive to memory module design, which results in both manufacturing challenges and practical challenges in the form of requiring cooperation with memory manufacturers to create.

While integrated accelerators benefit from low latency communication with conven-

tional cores, many accelerator designs circumvent the other structures on the processor die for the purposes of achieving higher performance [49]. For accelerators designed to compute over large volumes of data in a streaming fashion, caches are a hindrance. For accelerators accessing highly structured data, prefetchers have no purpose. For accelerators that exhibit well structured communication patterns, a packet-switched NoC is an unnecessary source of fluctuation in communication latency. There is also the simple and practical truth that integrating accelerators into a processor die is complex, and getting a commercial-grade highly-efficient accelerator packaged into the same die as a commercial-grade highly-efficient conventional core requires requires both a large engineering effort and cooperation between industry players. These factors together make an argument for moving accelerators that perform streaming style computation over large volumes of data into a separate die package entirely, and moving toward a more modular design, and leave integration to those accelerators that implement only the functionality that benefits from tight coupling to caches and other mechanisms that improve the performance of a conventional core.

Inspired by these observations, we propose an architecture that meets the following objectives:

- **Modularity:** The accelerator must be implemented such that it can be designed and tested in isolation.
- **Non-invasiveness:** It must be possible to introduce accelerator modules into a system that is otherwise composed of off-the-shelf components.
- **Low latency communication** Communication with the accelerators must be fast.
- **Shared Memory** The accelerator should operate over the same memory as the processor.
- **Scalable** The bandwidth accelerators are able to use must scale with the system, rather than be bound by packaging pin-out.

Our proposed architecture meets these objectives by introducing a new device to the system called an Accelerator In Memory (AIM) module. This device is an accelerator in a DIMM form factor, and features a DIMM interface on the AIM module into which conventional memory DIMMs may be seated. The AIM module with a memory DIMM seated into it would then be seated into the system motherboard. This design accomplishes the previous objectives in the following way:

- **Modularity:** The AIM itself is simple, and provides scaffolding for simplifying the authorship of customized accelerators as a separate package. The implemented accelerator also communicates with other devices using the well established protocols that the CPU uses to communicate with memory, which limits the need for testing communication protocols.

- **Non-invasiveness:** A set of AIMs can be introduced to a machine that otherwise consists entirely of off-the-shelf parts, that run off-the-shelf software.

- **Low latency communication** The needs of the CPU that drives the design of low latency and high bandwidth networks between the CPU and memory is leveraged to provide low latency and high bandwidth communication with the accelerators.

- **Shared Memory** The AIM literally uses the same memory as the CPU, and thus shared memory is automatic, without the need of costly hardware abstractions.

- **Scalable** As the memory system grows in size, the number of AIMs grows as well. Aggregate bandwidth observed by all AIMs is the peak DRAM bandwidth, rather than the artificial limit imposed by CPU pin-out.

While conceptually simple, there are a number of difficult challenges in achieving such a design. This paper discusses how we met these challenges, and shows experimentally that a system featuring AIMs can achieve substantially higher performance for a selection of benchmarks, and performs at least as well as accelerators integrated into the CPU for other highly parallel benchmarks, without incurring the engineering cost of integrating accelerators into a CPU.

6.1 Acceleration Platform

This section will describe the architecture and design of AIMs, along with how these modules interact with the rest of the system. Because the design of accelerators more broadly have been widely studied [4, 8, 24, 49–52], this section will not focus on the compute engine of the AIMs, but instead focusing on the mechanisms that are necessary to integrate AIMs into an existing system.

At a high level, a system using AIMs is structured as a hierarchy, with the CPU as the master, and AIMs serving as slaves. Each AIM and its attached memory acts as a small independent system capable of independent computation, while the CPU views all the memory in the entire system as shared memory.

While the particulars of a communication protocol between the CPU and system memory is important, the details discussed here are focused on the internals of the AIM itself, and not how these modules interact with any particular memory protocol. For this reason, this section will discuss components in a way that is agnostic to the memory protocol being used. An actual implementation of an AIM would be customized for a specific memory standard, and this customization may allow for certain components to be simplified. A more detailed discussion of how an AIM featuring system can be adapted to a specific memory protocol standard is presented in Section 6.2.

6.1.1 Accelerator Integrated DIMM Architecture

As shown in Figure 6.1, an AIM consists of an additional card that plugs into a conventional DIMM interface of a motherboard. An AIM in turn features its own DIMM interface which can seat conventional, off-the-shelf, DIMM memory. In almost every way, an AIM paired with a memory DIMM appears completely identical to a conventional memory DIMM from the perspective of the surrounding system. The sole difference between an AIM and a conventional memory DIMM is that an AIM filters out accesses to a small address range that serves the purpose of allowing communication with the accelerator. We will call this region the accelerator control memory region (ACMR). The memory

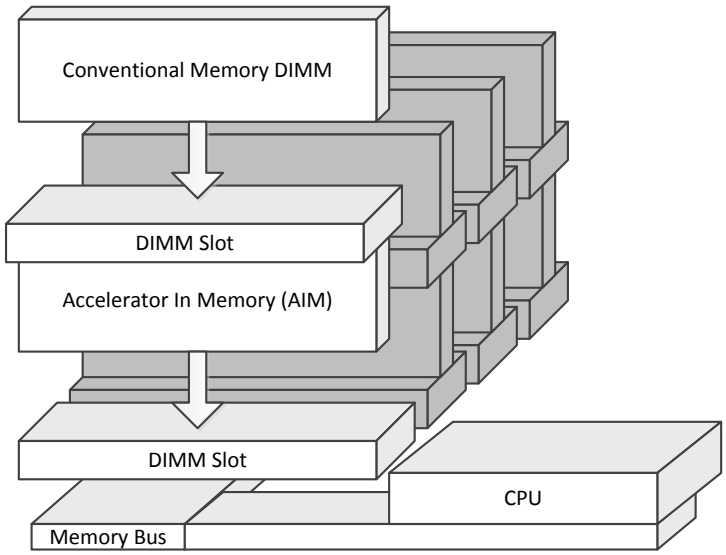


Figure 6.1: How a set of AIMs physically connect to an existing system.

DIMM seated in an AIM acts as both normal system memory from the perspective of the CPU, and also the private memory from the perspective of the attached AIM.

Figure 6.2 details the internals of an AIM. This module consists of three components, and interconnection between these components.

- **Address Filter:** The address filter decides whether or not an incoming access is an access to the ACMR, which is simply an address range check. If the access is to the ACMR region for a particular AIM, the request is serviced by reading configuration state of the compute engine, along with signaling the compute engine that the access has occurred. If the access is to the ACMR region belonging to a different AIM, the request is either ignored if the memory system uses a broadcast network, or forwarded on if the memory system features a multi-hop network. If the access is not to the ACMR, then it is a normal memory access and is forwarded to the Memory Response Filter.

- **Memory Response Filter:** When a memory access is made by the CPU, the Memory Response Filter tracks the address that was accessed. The purpose of this is to correctly route responses from the attached memory DIMM to the device

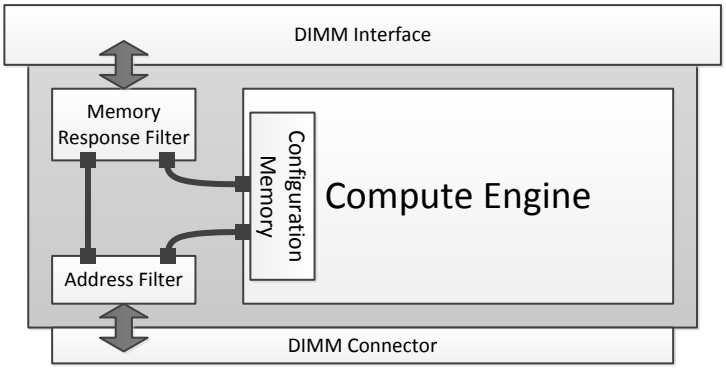


Figure 6.2: The internals of a sample AIM.

that issued the request: either the compute engine, or the CPU. When a response arrives from the attached memory DIMM, the Memory Response Filter is checked. If the address is found, the response is forwarded to the CPU and the entry in the Memory Response Filter is invalidated. Otherwise, the response is forwarded to the compute engine. The exact design of the Memory Response Filter depends on the type of memory system the AIM is being introduced into, but generally the Memory Response Filter is a small associated table, similarly to an MSHR [53], that is sized appropriately to the maximum number of concurrent accesses the attached memory module can support being simultaneously pending. If a memory format does not support multiple simultaneously pending accesses, the Memory Response Filter is simply a register and the necessary comparison logic.

- **Compute Engine:** The Compute Engine consists of both the accelerator(s), and a small amount of memory reserved for the purpose of servicing read requests from the ACMR. This does not require the entire ACMR address range associated with a particular AIM to be backed by this memory range, only that addresses not backed by this address range are write-only. These write-only addresses are used for configuring an accelerator prior to performing a task, or sending control signals to the accelerator. Any access performed on an ACMR region for the AIM receiving the access results in a signal being sent to the accelerator indicating that an access has occurred, to what address the access was made. In the case of writes, the signal

also contains the value that was written. The reason that parts of the ACMR are backed by dedicated physical memory is to allow for a guaranteed access latency, which is a common requirement in memory systems. A read to an area that is not backed by physical memory returns the value of zero for all bits requested. The exact functionality of the accelerator encapsulated within the Compute Engine is outside the scope of this section, and is expected to be application specific. In Section 6.3, we describe that our experiments assumed that the compute region is an FPGA, but can also be implemented as dedicated ASIC [8], CGRA [54] or composable accelerators [24], or any other compute mechanism.

Memory accesses to addresses in the ACMR region are routed independently from the normal memory address interleaving protocol imposed by the memory controller. This is done to assure that any specific address range can be chosen for the ACMR, and the systems memory controller does not have an opportunity to exempt certain AIMs from having adequate ACMR space allocated to them. This is done by overriding the DIMM select mechanisms that would be used by the memory controller. Because an address that is determined to be within the ACMR range by a single AIM will never be determined to not be in the ACMR range by a different AIM, this cannot result in multiple responses to a single access. Furthermore, overriding routing for accesses in the ACMR range enables broadcasting to all AIMs. This is performed by all AIMs simultaneously determining that a given access maps to their portion of the ACMR, and that it maps to the ACMR portion of another AIM.

The ACMR is split into two regions: global memory and local memory. An access to ACMR global memory is treated as a broadcast using the mechanism described above, and is received by all AIMs. This is primarily used for bulk configuration of accelerators or for communication of control signals. Because a single memory request is expected to solicit a single memory response, read accesses to ACMR global memory are serviced by the AIM indicated by the DIMM select mechanism emitted by the memory controller. ACMR global memory is not backed by compute engine register state, and thus a read from an address in this range is responded to with a zero. An access to ACMR local

memory is an access or signal to a specific AIM, such as to set control registers that differentiate a given AIM's accelerator from that found in its neighboring AIM module. While it is not necessary that ACMR local memory be backed by physical memory in the AIM, any ACMR memory that is backed by physical memory in an AIM is assured to be local memory. Also, because most AIM configuration is expected to be done via global memory, configuration time remains constant regardless of the number of AIMs in the system.

An AIM gives priority to memory accesses made by the CPU. This is due to the increased sensitivity of processing cores to memory latency. This is also frequently mandated by the memory communication protocol as well.

6.1.2 Accelerator Use

The primary mechanism for communicating with AIMs is an AIM driver. This driver also serves the purpose of reserving the memory address range that corresponds to the ACMR as being mapped to a device, as is typically done for hardware memory mapped hardware devices. Memory mapping the ACMR region is only done to forbid the operating system from allocating accelerator configuration addresses for other purposes, but does not correspond to any device listening to the CPUs address bus. For this reason, an access to an ACMR address results in an access to the conventional memory system, and thus is visible to the AIMs.

In order to send data to an AIM, a region in the ACMR is written, and then instructed by the CPU to flush the modification from the cache, such as by explicitly invalidating the block. This will result in a write of the dirty cache region back to memory, which would in turn result to a signal being sent to the appropriate AIM. The process of reading data from an AIM is similar: the CPU requests a region of memory that maps to the ACMR local memory associated with a specific AIM, and gets the response back just as would occur with a normal memory read. There is no coherence mechanism between AIMs and the CPU, so if the CPU wants to get an up-to-date value from an AIM, it

must first assure that the block associated with the ACMR local memory is not stored in the local cache.

Because the memory controller does not support unsolicited responses from memory, AIMs cannot communicate information to the CPU without having this information requested via a read from the ACMR local memory. In theory, requiring the CPU to poll the AIMs for progress updates, such as to see if a task has been completed, seems highly inefficient. In practice however, the inefficiency of polling is offset by the long-running nature of the data parallel computations over large volumes of data that AIMs are designed to perform, combined with a quite predictable run time [8]. Because computations of this kind tend to both be long running, and predictable, using the AIMs' compute engine to generate a estimated time until the task is completed was found to be a very effective method of making polling more efficient. A CPU would poll current estimates, then wait until the estimated time has elapsed before polling again. This process would be repeated until a poll resulted in a notification that the task is complete.

The exact protocol for interacting with the accelerators located on AIMs is application specific, and thus beyond the scope of this section. Our methodology focuses on extending a mechanism for communication, rather than making demands on a particular communication protocol.

6.1.3 Memory Layout

A critical requirement of this system is that memory required for the operation of an accelerator must be located entirely within a single memory DIMM. Each AIM is only capable of interacting with the memory that is directly attached to it. The primary reason for this is that unexpected traffic on the network joining the CPUs memory controller to main memory may result violate the assumptions the memory controller makes about the memory communication protocol. For this reason, all memory accesses made by an AIM must not require transmitting memory over the memory network, and thus must only be to the attached memory. Communication between AIMs must be performed as a

result of a memory access on the part of the attached CPU, or via direct memory copies performed by the CPU. A secondary reason is simply performance, as the aggregate bandwidth potential of all attached memory in a system is typically much larger than the bandwidth of the memory network.

In order to make effective use of all AIMs in the system, software must distribute data required by a given computation evenly across all memory DIMMs, and aligned such that all the inputs for a given computation are all located on a single DIMM. For linearly structured streaming computations, this is a straight forward requirement to meet: Each input stream is a sequence of elements such that the first element of each stream resides in the same memory DIMM, and padding is added so as to make all inputs to a single computation the same size. This would guarantee that for all computations, all inputs needed by that computation reside in a single memory DIMM. AIMs can work on the entire data set in parallel, with no further action on the part of a CPU.

For more complex computations, the use of AIMs can be optimized by restructuring data that will be consumed by the AIMs such as to maximize the amount of computation that can be performed by a single AIM. Various compiler passes can facilitate the process of aligning data properly, as the process of mapping data to DIMMs is identical to data tiling and for either improving cache performance or vectorization [55, 56]. Often though this process leaves regions parts of the computation that cannot be performed exclusively using data located in a single memory DIMM. These computations can either be done within the CPU, or via data replication.

6.2 Standards Compliance

To achiever the objective of using off the shelf components for every component of our proposed system, except for AIMs, we designed our system to work within the tight constraints laid out by existing communication protocols between the CPU and main memory. Because there are many such protocols, and each of them features small but critical differences, we chose as a case study the common protocol of DDR3 memory over

a multidrop bus. This design point was chosen because it is the protocol found in most modern systems, and because its specification is at least as restrictive as most modern competing technologies.

6.2.1 Timing

Because the introduction of AIMs involves inserting additional address filtering logic between a normal DRAM module and the memory network, a slight increase in the latency for operations is to be expected. Total bandwidth would not be affected, but latencies of CAS & RAS operations would be increased by a value δt in order to allow for this additional logic, and would have no impact to operations that do not solicit a response such as writes. δt is calculated as the amount of time it takes to filter the request from the CPU against the ACMR range, plus the time it takes to check whether or not the response from the attached DIMM is as a result of a request from the CPU. This change in timing would not impact potential bandwidth, but only the latency of individual accesses. δt is a small value, since the computations mentioned above are very simple. Within this work we considered δt to be 1 nanosecond, where 0.5 nanoseconds were used for filtering the request, and 0.5 nanoseconds were used for filtering the responses.

The memory DIMM itself would operate in the exact same way as before, at the exact same clock. During initialization the AIM would add a single nanosecond to the latencies returned by the memory DIMM, and send these values to the memory controller instead of the original values. From the perspective of the memory controller, the memory seated in the DIMM slot containing an AIM would feature memory with slightly longer latencies than is actually included in the system.

An additional concern is the handling of accesses from the CPU that arrive while the DRAM is busy handling a memory read that was issued from the accelerator. The DRAM cannot be interrupted, and the memory controller is expecting a response at a very specific time. In this event, it is impossible to successfully satisfy the CPUs request on schedule. As a last-resort mechanism, the accelerator will return a dummy data value

that is known to fail the memory controllers error check. This will cause the memory controller to retry the request again in the hopes that the a transient error had been encountered. The accelerator will use this window of time to stall the accelerator so as to assure that the DRAM is not busy when the memory controller retries the request.

6.2.2 Communication with CPU

As was mentioned in Section 6.1.2, AIMs act purely as slaves to the CPU and do not interact on their own with either each other or the CPU. For this reason, all communication between accelerators and the CPU must occur upon solicitation by the CPU. This occurs when the CPU accesses a memory region that is mapped to the configuration space of an accelerator, at which point the AIM may respond.

This communication pattern has implications on software design, as the CPU must assume the lack of an interrupt mechanism for notifying software of the progress of accelerators. A CPU would have to poll for completion, rather than relying on event to notify the CPU of progress. Many accelerator designs however have predictable workload run times, and as such predictions can be made regarding how to poll more intelligently. In our cases, for example, we found that the total execution time of an accelerator could be fairly accurately estimated immediately after execution. The CPU polls a region of the ACMRs local memory space that held the value of an estimated time until completion, and would not poll again until that time had elapsed. Since our accelerators typically ran for hundreds of thousands, or even many millions of cycles at a time, a slight over-approximation of the time to completion resulted in very little performance overhead while also assuring we polled only a small number of times before finding that the accelerator had completed its task.

6.2.3 Routing of Configuration Messages

Configuring each accelerator individually is time consuming if all accelerators are going to be configured identically. To eliminate this, we proposed the use of a ACMR global

memory. The implementation of the ACMR global memory is done by over-riding the DIMM select bits on messages sent by the CPU over the memory network. In a conventional system, normal memory DIMMs only respond to communication where the DIMM select bits are set to a specific value, every message sent over the multi-drop bus is received by all memory DIMMs and ignored by all except one. In a system that features AIMs, an address filter is applied regardless of the setting of the DIMM select bits. If the address corresponds to the ACMR, the AIM accepts this message and ignores the DIMM select bits entirely. ACMR global memory is implemented by having each AIM listen for accesses to the same memory address range. ACMR local memory is implemented by having each AIM filter addresses within a disjoint portion of the ACMR. This mechanism allows for communication from the CPU to the accelerators to be routed independent of the normal memory interleaving policy implemented by the memory controller.

6.2.4 Consideration of Alternative Protocols

While DDR3 over a multi-drop bus is the most pervasive memory configuration used in modern machines, there have been plenty of others that have competed for that market share. This section will briefly describe why transitioning to some other near-by protocols would still allow for the design of an AIM described in Section 6.1, though with some changes made to the specifics mentioned in this section.

The most obvious point of comparison are Fully Buffered DIMMs [FB-DIMMs], which have seen implementations as modifications on DDR2 and DDR3. FB-DIMMs abandon a multi-drop bus in exchange for a more scalable network topology consisting of a chain of linked DIMMs. To access memory in a far away DIMM, a message is passed from one DIMM to another down the chain, and the response is propagated back in a similar fashion. This protocol has the advantage of not necessarily limiting inter-DIMM communication, since the network connecting DIMMs to one another is not a broadcast mechanism. Thus this would allow for a more expressive accelerator model than can be implemented over a system featuring a multi-drop bus.

Some protocols, such as that proposed in the Hybrid Memory Cube[HMC][57] work, do not demand a specific round-trip delay on individual accesses to memory, and instead allow individual memory flexibility with regard to timing for accesses. In these cases, a back-off mechanism such as that described in Section 6.2.1 would not be needed, and it would be sufficient to simply wait for memory to service an accelerators already pending memory request prior to then servicing the request from the CPU. It would still be beneficial to give the CPU priority, but it would no longer be necessary. A memory controller that does not expect a rigid round-trip time for memory accesses would not necessarily require a new memory standard, such as HMC, but could be any memory system that lacks deterministic timing due to the topology of the memory network as well, such as that described in [58].

6.3 Evaluation Methodology

6.3.1 Evaluated Systems

It has been extensively shown in prior work that an accelerator is capable of providing substantially higher performance than a conventional processor [4, 8, 24, 49–52]. To create a fair comparison for evaluation, we compared a system featuring accelerators integrated into the processor die, to a system featuring accelerators integrated both into the processor die and distributed through the memory system in the form of AIMs. In all cases, accelerators were FPGA modules. For a given number of DIMMs, the FPGA resources in the baseline system was the same as the FPGA resources in the system featuring AIMs. We chose this point of comparison, instead of a comparison between a conventional CPU and a system featuring AIMs, because there has been extensive prior work illustrating the performance and efficiency benefits of accelerators over conventional processors [4, 8, 24, 49, 50]. We thus felt it would not be a meaningful study unless we compared a system with AIMs against another accelerator-featuring platform.

On both the baseline and the AIM featuring system, we modeled a network bus with

Table 6.1: Characteristics of the evaluated system(s)

Cache	8MB L2, 8-way set associative, 32 banks, 10 cycles latency for L2 access
Coherence Protocol	Shared banked L2 cache. MOSI protocol
FPGA region	FPGA from Xilinx Zynq 702 board. 52K LUTS, 106K FF, 140 BRAM, 220 DSPs
Memory Network	24-GBps peak bandwidth, multi-drop bus
DRAM memory	DDR3-1600 DIMMs, 12.8-GBps peak bandwidth, 11.25 ns CAS latency

the bandwidth and latency characteristics of Intel's Quick Path Interconnect (QPI). While we did not make use of the coherence or multi-socket features of QPI, we did make use of its bus standard. All systems modeled featured a single socket into which a processor is placed, along with between 1 and 16 memory DIMMs.

Details of the baseline system are presented in Table 6.1. The baseline system contains one more FPGA regions than there are DIMMs in the system integrated into the processor die. The experimental system features the exact same configuration, except for an AIM between the DIMM interface and the memory DIMM for each memory DIMM in the system. The experimental system only features a single FPGA region integrated into the processor die, along with one FPGA region acting as the computation engine in each AIM. In both systems, all FPGAs are used for all compute tasks, with the CPU primarily being used to configure the FPGAs. In the experimental system, the CPU also is used to provide explicit transfers between AIMs where necessary. FPGA regions that are integrated into the processor die access memory through a cache interface, and are thus cache coherent.

Though these two systems are relatively simple, we believe they demonstrate the potential of our approach. The baseline is still representative enough of more complex accelerator architectures like GPUs, or specialized coprocessors like the Cell [59], to

provide us with confidence in the generality of our results.

6.3.2 Benchmarks

We selected a diverse set of applications by focusing in on three distinct categories of applications, and selecting a number of benchmarks from each category for our results. These categories are:

- **Trivially data-parallel (TDP):** These workloads are the type of computations for which vector processors and stream processors are most effective. Using software assisted data alignment, all data for each individual computation that is to be performed can reside in a single memory DIMM, and thus accelerators can work completely independently from one another. The entire computation can be partitioned in this way, leaving nothing that requires accessing multiple memory DIMMs. AIMs perform the entire computation in these cases.

- **Complex data-parallel (CDP):** These workloads are computations that exhibit data parallelism, but for which dividing data cleanly into specific DIMMs is impossible for some subset of computations to be performed. Our chosen benchmarks for this category are predominantly stencil-based computations, with the data divided evenly among memory DIMMs. Computations that require data located on the edges of the dividing lines between regions require data from multiple DIMMs, and thus cannot be performed by the AIMs. In these cases, the computation is performed on the CPU by an integrated accelerator. The overwhelming majority of computations do not require input data from multiple memory DIMMs, and thus can be performed within the AIMs.

- **Phases & Filters (PF):** These workloads feature distributed computations that either perform reduction operations, or require shuffling of data between compute phases. The majority of each workload is parallel, but needs the CPU to organize data transfers between AIMs. In the case of the baseline system, these workloads

typically also feature regions that are cache friendly, such as the first several phases of calculating FFT. Because these workloads consist of different compute phases, the tasks performed by accelerators integrated into the processor are often functionally different than the functionality performed in the AIMs.

Table 6.2 describes which benchmarks we chose, in which categories they fit, and a brief description of the type of computation performed in each benchmark. For PF workloads, there is also a description of the type of computation performed by each type of accelerator. All workloads were optimized so as to maximize the amount of computation that can be performed using data residing in a single memory DIMM, while also evenly distributing the data between all memory DIMMs. For workloads running on the system featuring AIMs, the addition of padding to data was required to further increase the amount of computation that could be done within a single AIM. Because this padding reduces the efficiency of used bandwidth, by reducing the density of useful data, padding was not applied for workloads run on the baseline system.

6.3.3 System Modeling

To model our system, we used a heavily modified version of the GEMS [20] and Simics [19] full system simulator. Our modifications were primarily focused on the integration of accelerators, along with mechanisms to communicate with accelerators. We also introduced a model for memory, which was previously absent from the base GEMS simulator, and the memory network connecting the processor to main memory. In order to calculate latency of our accelerators, we synthesized them with the use of the Vivado tool suite [37], and back-annotated the derived latency values into our simulator.

6.4 Results

The primary metric that we used to evaluate our proposal is the impact that AIMs have on system performance. To show additional insight into why the performance was

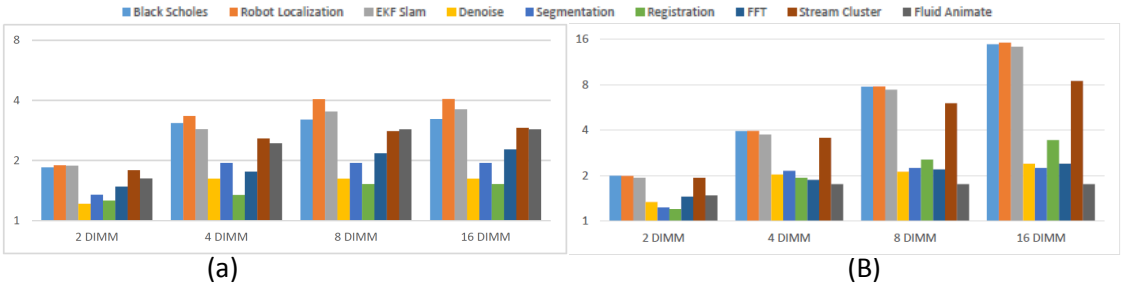


Figure 6.3: Shows the performance scalability of systems with and without AIMs as the number of DIMMs in the system is increased. (a) shows a system with AIMs normalized to the performance of a system with AIMs and only 1 memory DIMM, (b) shows the baseline system without AIMs normalized to the performance of a system without AIMs and only 1 memory DIMM.

impacted in the way that it was, we also examined memory network utilization and memory access patterns and demand by each benchmark.

6.4.1 Performance

Figure 6.3 shows the normalized performance of each benchmark for each system configuration as the number of memory DIMMs increase. The performance of our baseline system is observed to scale up smoothly for most benchmarks scale, until the system contains between two and four memory DIMMs. At this point there is a sharp decline in the rate of improvement, followed by a plateau. This plateau clearly indicates the memory wall, and is the point at which expanding the memory system serves only to improve memory capacity, but does not further benefit bandwidth and thus does not further benefit performance. While this point is reached at different points for different benchmarks, it is clear that all benchmarks examined are impacted by this problem.

In the case of our experimental system, however, it is observed that most benchmarks continue to improve as the size of the memory system enlarges. The reason for this is simple: AIMs are able to continue making use of bandwidth beyond the point at which bandwidth would be limited by the processor die packaging in system with on die

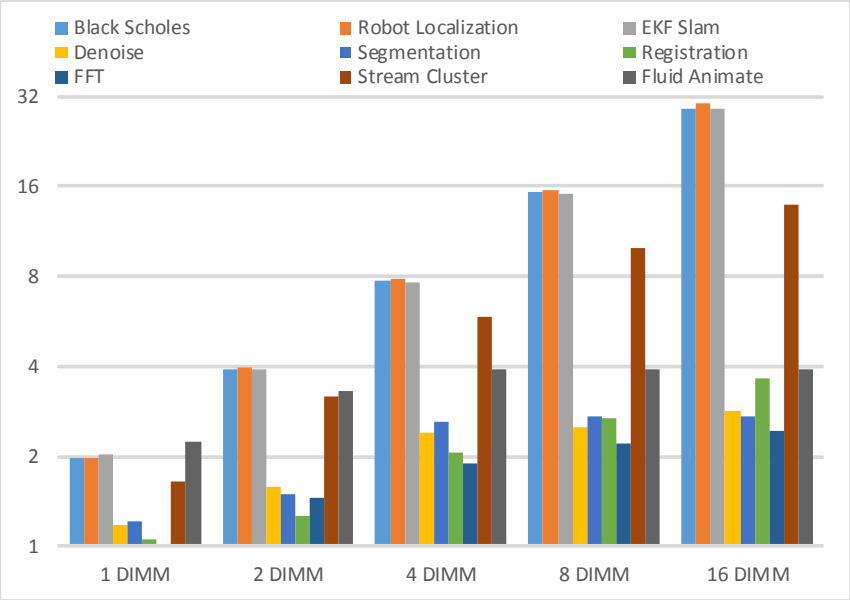


Figure 6.4: Shows performance of a system featuring AIMs compared normalized to the performance of the baseline system with 1 memory DIMM.

accelerators. In addition to the performance scalability of a system featuring AIMs, we also observed a direct performance impact associated with bypassing the CPUs memory system and bus arbitration. Not only was the system featuring AIMs able to fully leverage the bandwidth of each memory DIMM, but the AIMs saw a reduction in memory latency as would have been observed by an accelerator integrated into the processor die. This is attributed both to bypassing the memory network and the memory controller used by the CPU, but also because of being able to bypass caches in the instances in which accelerator data exhibited poor cache performance. Figure 6.4 illustrates this effect by showing the performance of our experimental system relative to our baseline system with 1 memory DIMM. Figure 6.4 also helps to put Figure 6.3, discussed above, in context.

6.4.1.1 Trivial Data Parallel [TDP]

TDP benchmarks, as described in Section 6.3.2, are those for which input data for computations can be completely divided evenly among memory DIMMs. For this reason, for

AIM featuring systems, the AIMs can perform the entire computation with out assistance from the controlling CPU, except for configuring and initiating the accelerators. Our results predictably show that we see near-linear performance improvement in our experimental system as additional memory DIMMs are added. This is not surprising, as this is the most optimal type of workload for the use of AIMs. Our baseline system benefits from additional DIMMs so long as the aggregate bandwidth of all DIMMs in the system is less than the bandwidth of the network connecting the CPU to the memory. While not experimentally shown, it is intuitively clear that the performance of a system featuring AIMs will continue to scale near-linearly with the addition of DIMMs to the system.

6.4.1.2 Complex Data Parallel [CDP]

CDP benchmarks exhibit improvement with the addition of DIMMs. As is discussed in Section 6.3.2, these benchmarks feature stencil computations. Data is partitioned among DIMMs such as to maximize the portion of the total computation that can be performed with the data contained in a single memory DIMM, while still distributing the data evenly among memory DIMMs. Even with an optimal division of labor, some of the computations are going to use data stored in multiple DIMMs, and thus a set of the computations require data from multiple DIMMs. We chose to implement these benchmarks by using the accelerator integrated into the processing die to perform the computations that require data from multiple memory DIMMs.

In the case of AIM featuring systems, the portion of the work that must be performed on the CPU increases with each memory DIMM that is added. This is because the number of regions the input data gets partitioned into increases with each added memory DIMM. For our experiments, the CPU became the system bottleneck in a system with 8 memory DIMMs. There is a marginal performance benefit moving to 16 memory DIMMs that is derived from the fact that the CPU sees a reduction in competition for memory DIMM bandwidth with the AIM. The AIMs continue to scale performance linearly, and

thus complete their task earlier, thus freeing up memory bandwidth to be used by the accelerator integrated into the processor.

As the number of memory DIMMs becomes larger, the portion of the input data that is on the border of divided regions grows, and the amount of work that can be off loaded to the AIMs shrinks. For this reason, it can be concluded that there is a break-even point, beyond which adding memory DIMMs would reduce performance. This can be resolved by limiting the number of AIMs used in a computation, by inserting padding into input data to resulting in the memory controller skipping certain DIMMs when mapping the useful regions of the input data. This would exempt certain specific AIMs from having useful input data mapped to their attached memory DIMM, which would in turn exempt them from the participating in the computation.

It should be noted however that, because platform is a simulator, we are constrained in the size of data that we can evaluate. A real workload would feature a substantially larger input size over which the stencil computation is being performed, and thus would feature a much smaller relationship between border regions and center regions of the divided up input data. As a result, a full-scale system can be expect to reach this break-even point at a substantially larger number of memory DIMMs than was experimentally shown here.

6.4.1.3 Phases and Filters[PF]

PF workloads were the most complex workloads that are evaluated within this work. As is discussed in Section 6.3.2, these benchmarks consist of multiple phases of computations. AIMs are used either to filter out the volume of data that is needed by the centralized accelerator, or to pre-compute data so as to reduce the complexity of the centralized accelerator.

FFT consisted of multiple levels of the Cooley-Tukey algorithm, which is very cache friendly. For this reason, the cache coherent accelerators located in our baseline system performed very well. Bandwidth was not so much a bottleneck as much as access latency

with the cache. Cache behavior was only problematic when computing the last iterations of the Cooley-Tukey algorithm. Even in this situation though, which was expected to be the poorest performing of our benchmarks, the AIMs were able to remain competitive, and effectively match the performance of the baseline system. It should be noted however that the AIMs were able to perform as well as centralized accelerators, but would not require the level of invasive redesign that would be necessary to create a system that featured centralized accelerators.

Stream Cluster was the most parallelizable benchmark that was examined in this category. In systems featuring AIMs, no computation was actually performed by the centralized accelerator. AIMs are used to perform vector arithmetic and produce partial results. Software would then read each partial result stored in the ACMR local memory, perform a quick computation, and broadcast the result back out to all AIMs via a write to the ACMR global memory. Software was occasionally required to perform slight rearranging of data. In systems without AIMs, the entire computation was done using centralized accelerators with no software interleaving. This resulted in a highly scalable implementation.

Fluid Animate featured computation similar to the CDP workloads, except also including a phase which used a centralized accelerator to move copy data elements between partitioned regions. For this reason, Fluid animate exhibited the same scalability issues that were seen in CDP workloads, but was further bound by the data copy step. Fluid animate was however also the most compute intensive of the examined workloads. These factors combined meant that the centralized accelerator performed quite well on this benchmark, as the system bandwidth was not the main limiting factor, but rather a constraint on compute capacity. As was mentioned in Section 6.3.1, we scaled the number of FPGA resources on the baseline system as the number of memory DIMMs increased, to make up for the compute advantage that would be added by increasing the number of AIMs in the system. This difference allowed for our baseline system to hit the memory wall very late, and thus hid many of the advantages that can be gained by introducing AIMs to a system.

6.4.2 Memory Activity

Because the AIMs lack a cache to filter out redundant memory accesses, we observed an increase in the aggregate number of DRAM accesses in a system featuring AIMs as compared to our baseline system. In our CDP benchmarks, this difference comes in the form of redundant accesses for inputs to the stencil computations in instances where the accelerator was not able to internally buffer the reused data. While a stencil computation often also benefits from caches in general, our accelerator design internalized nearly all of the sharing between neighboring computations, and thus caches were not needed to fill that roll in either the baseline system or the system featuring AIMs.

In the case of FFT, the impact of caches was more significant. Caches in the base system were able to eliminate the majority of memory accesses for the first few phases of the Cooley-Tukey algorithm. Later phases suffered from a high cache miss rate however. For this reason, as is shown in Figure 6.4, FFT does not show any benefit in the 1 memory DIMM case.

6.4.3 Memory Network Utilization

As discussed in Section 6.4.1, our baseline system encountered the memory wall after adding 4 DIMMs in most cases. This is further illustrated in Figure 6.5 which shows the utilization of the memory network for a chosen workload running on the base system. Figure 6.6 shows a similar measurement of the memory network utilization for systems featuring AIMs. As is clearly shown, nearly all of the memory network traffic is eliminated while executing the benchmark. While not shown, other benchmarks exhibited similar traffic patterns.

Even though the accelerator resources are scaled up in the baseline system as the number of DIMMs are scaled up, no performance improvement is observed once the memory network is fully saturated.

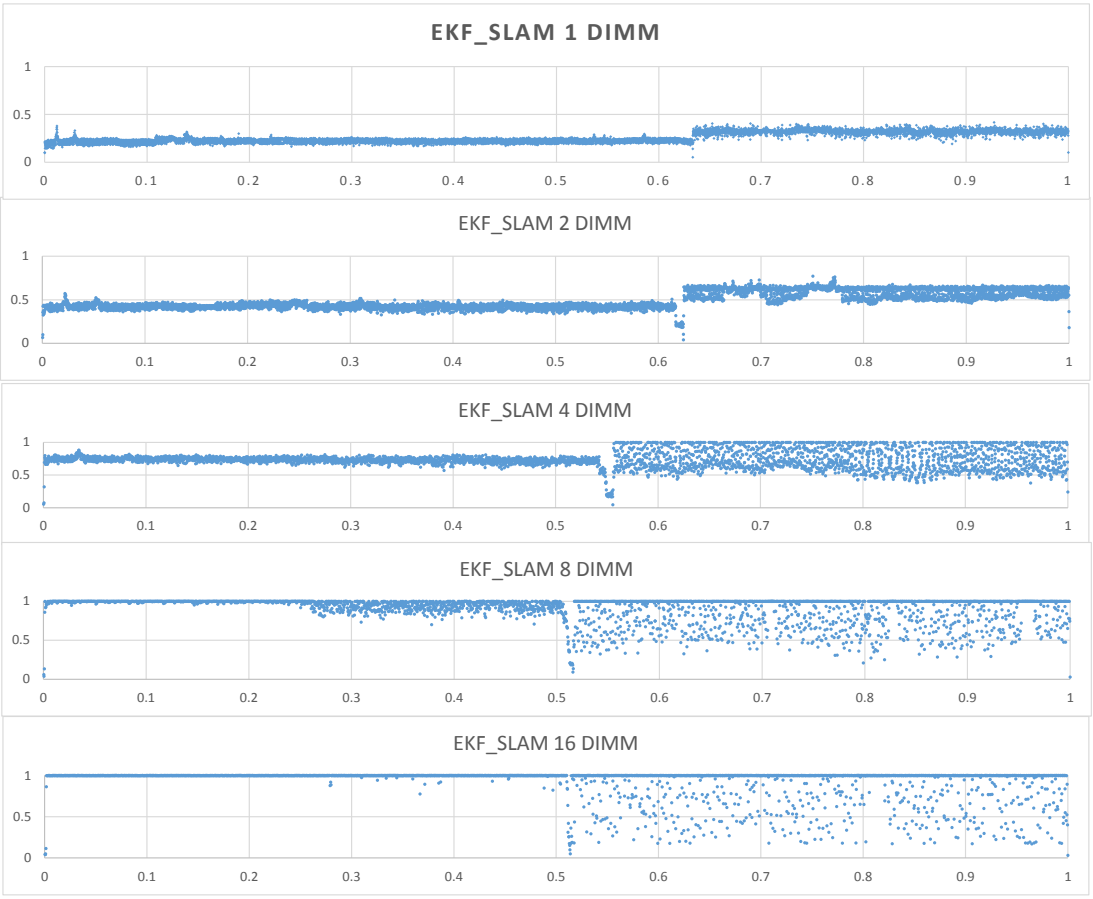


Figure 6.5: Shows the bandwidth consumption on the memory network for the baseline system for EKF_SLAM.

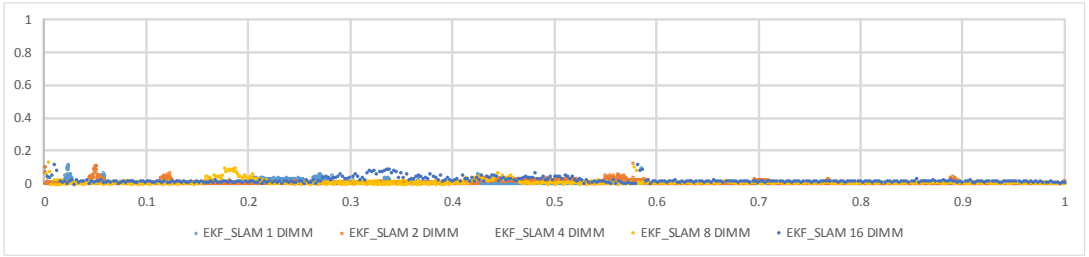


Figure 6.6: Shows the bandwidth consumption on the memory network for all systems featuring DIACs for EKF_SLAM

Table 6.2: Description of evaluated benchmarks

Benchmark	Type	Description
Black Scholes	TDP	Stock pricing simulation, part of PARSEC [39]
EKF_SLAM	TDP	Performs simultaneous localization and mapping [17], from the Mobile Robot Programming Toolkit [41]
Robot Localization	TDP	Performs Monte Carlo Localization using probabilistic models of motion [42]
Segmentation	CDP	Stencil computation of 3-dimensional image segmentation in medical imaging [31]
Denoising	CDP	Stencil computation on 3-dimensional image to eliminate imaging noise in medical images [31]
Registration	CDP	Iterative computation of fluid registration [31]
FFT	PF	A 16 radix Cooley-Tukey [60] iterative FFT calculation on a single dimension AIMs perform all but the last phase of the Cooley-Tukey algorithm. Centralized accelerator performs the last phase of the algorithm
Stream Clusters	PF	Phase-based reductions and accumulate operations [39] Compute intensive vector operations on long streams AIMs perform partial reductions. Centralized accelerator performs final reductions & results broadcast
Fluid Animate	PF	Fluid dynamics simulation [39] AIMs perform fluid simulation within a region. Centralized accelerators migrate particles between regions

CHAPTER 7

BiN: Buffer-in-NUCA for Accelerator-Rich CMPs

In order to work efficiently, accelerators need some degree of assurance regarding latency to memory accesses. This typically takes the shape of private memory that is local to the accelerator, such as the SPM that is described in Chapter 2 and Chapter 3. This design choice works well if the primary purpose of the SPM is to provide buffering, and perhaps a small degree of reuse, and is thus relatively small in size. For computations that would benefit from large SPMs, such as to provide a larger degree of reuse within a single transfer into local memory, providing a private memory for each accelerator becomes problematic due to the size of memory that would be most efficient for the computation being performed. In these cases, there have been works that look at allocation of shared buffers [61], or buffers in caches [62]. These designs however assume a uniform cost associated with accessing memory, which is not reasonable in the context of many-core systems.

Many core systems typically feature Non Uniform Cache Architectures (NUCA) for last-level caches (LLC). These systems exhibit performance characteristics that differ greatly relative to where data is placed, and where the consumer of the data is located on the network. To operate in this environment, we proposed a Buffer in NUCA (BiN) [63] protocol, that performs buffer allocation in shared cache in a way that is locality aware. This also has the added benefit of potentially reducing off-chip memory accesses for accelerators that are capable of subsuming a greater quantity of reuse when provided with a larger buffer. We capture this bandwidth savings using a representation that we referred to as the Buffer size vs Bandwidth Curve(BB-Curve), which communicates bandwidth expectations as a function of provided buffer size. Because a programmer

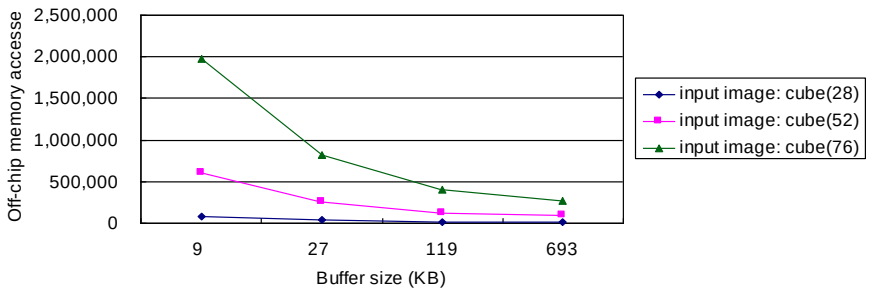


Figure 7.1: BB-Curve of a denoise accelerator

will not necessarily have information regarding system utilization at any given point, the BiN system takes as input a BB-Curve, rather than an individual buffer size request, and delegates to a hardware management component to allocate an optimal buffer size given the current system state. An example BB-Curve can be seen in Figure 7.1.

Prior work allocates a contiguous space to each buffer to simplify buffer access [62], since the address of a buffer block is calculated only as a relative position with respect to the buffer starting address. This may lead to space fragmentation when requested buffers have unpredictable space demand and come in dynamically. As shown in Figure 7.2, at Cycle 1K, there is 10KB available space in the shared buffer. But since this space is not contiguous, Buffer3 cannot be allocated. NUCA complicates buffer allocations in cache. In addition to the buffer size, the distance of the cache bank in relation to the accelerator also matters. It is possible that the only contiguous space that can satisfy a buffer is quite far from the accelerator and a better choice may be to aggregate several smaller available space segments in the cache banks around the accelerator. One approach to leveraging these fragmented resources is to make use of a paged scheme that adds a level of indirection. This makes physically non-contiguous spaces in NUCA appear to be contiguous, analogous to a typical OS-managed virtual memory. However, accelerators can not afford a large private page table in terms of energy and area; nor can they, for performance reasons, afford a multi-hop scheme to access a centralized shared page table.

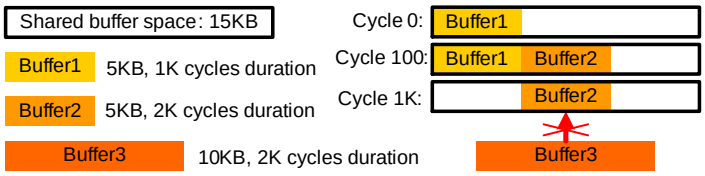


Figure 7.2: Buffer space fragmentation in shared buffer.

7.1 BiN Architecture

7.1.1 Overall Infrastructure

We construct BiN upon the ARC [8]a hardware-managed accelerator-rich CMP. Figure 7.3(a) shows the overall architecture of our evaluated ARC, which is composed of cores (with private L1 caches), accelerators, the accelerator and BiN manager (ABM), NUCA (shared L2 cache) banks, and NoC routers. The off-chip memory controllers (not shown) are attached to the routers on the four corners. ABM manages accelerator sharing (the same functionality as that of the GAM in [8]) and allocates buffers in NUCA.

Figure 7.3(b) shows the communications between a core, ABM, an accelerator and NUCA. The numbered arrows show the order of steps taken during a single accelerator invocation by a core. Buffer allocation (Step 2) is described in the following subsections.

7.1.2 Dynamic Interval-based Global (DIG) Buffer Allocation

In BiN, the space allocated to a particular buffer is dynamically determined at runtime. To avoid greedily allocating buffer space for each buffer allocation request, we propose a dynamic interval-based global (DIG) buffer allocation scheme. The key point is that ABM will collect the buffer allocation requests in a short fixed-time interval and then perform the global allocation for the collected requests to achieve short-time global optimality. By keeping the interval appropriately short, we limit the impact on performance of idle waiting in the interval. In this work, unless otherwise specified, we set the interval to be 10K cycles. Moreover, to avoid having too many accumulated buffer allocation requests,

once up to eight buffer requests are collected, the DIG allocation will be immediately triggered.

7.1.2.1 Problem Formulation

Given:

- The batch of buffer allocation requests with a set of points in the BB-Curve of request i as $\{(b_ij,\ s_ij)$ — 0 ln j ¡ $N_i\}$ (in increasing order of buffer size, where N_i is the number of points in the BB-Curve of request i, b_ij and s_ij are the bandwidth requirement and buffer size of the j^th point of the BB-Curve of request i, respectively);
- The total available buffer size W

Goal: Find one and only one node n_i for each buffer allocation request i, so that the total bandwidth $\sum_{i=0}^{N-1} b_n$ is minimized and the sum of the buffer size $\sum_{i=0}^{N-1} S_n$ is less than or equal to W.

7.1.2.2 Optimal Solution

This problem can be solved optimally through dynamic programming. Define a N+1 dimension array M, each element M[n_0, n_1, ... $n_N - 1$, w] denotes the minimum total bandwidth that can be attained with a buffer size less than or equal to w, when the buffer request i uses its curve nodes up to the node n_i (not that this curve does not necessarily need to use node n_i it may use any node in its curve up to n_i). For the first N dimensions, each dimension i has a size of N_i, where N_i is the number of points in the BB-Curve of the buffer allocation request i. The last dimension has a size of W.

Then we can have:

$$M[n_0,...,n_i,...,n_N-1,W] = \begin{cases} M[n_0,...,n_i,...,n_N-1,W] & \text{if } s_i n \text{ ¿ w} \\ min(M[n_0,...,n_i,...,n_N-1,W]+b, & \text{if } s_i n \text{ ln w} \\ M[n_0,...,n_i,...,n_N-1,W]) & \end{cases}$$

Note that if one of the first N dimensions of the M array is 0, it means that none of the BB-Curve points of this buffer allocation request can be used. The solution can then be found by calculating $M[n_0,...,n_i,...,n_N-1,W]$. And the complexity for both space and time is $O(N_0 * ... * N_i * ...N_N - 1 * W)$.

7.1.2.3 Online Greedy Heuristic

The aforementioned dynamic programming approach can solve the problem optimally, but it incurs relatively high timing and space overhead. Moreover, this optimal solution assumes that the buffer can always be allocated if there is enough space, regardless of the space fragmentation problem. Thus we developed an online greedy heuristic to solve the problem fast and efficiently, with consideration of fragmentation.

The algorithm first allocates the minimum buffer size for each request and tries the paged allocation to see whether this allocation is valid. If so, it then checks the next point of each curve, and selects the one with the maximum $(b_i j - b_i(j-1))/(s_i j - s_i(j-1))$, i.e., the request that gives the maximum reduction of bandwidth with unit increase of buffer size. Again, the paged allocation is tried to validate this new allocation. This process will go on, until the next point of each BB-Curve makes the resulting allocation not valid. An example of this scheme is shown in Figure 7.4. To save the computation of the DIG allocation, the $(b_i j - b_i(j-1))/(s_i j - s_i(j-1))$ (buffer utilization efficiency) is pre-computed, so that the curves sent to ABM are actually $((b_i j - b_i(j-1))/(s_i j - s_i(j-1)), s_i j)|0 \ln j < N_i$, where $b_i(-1) = 0, s_i(-1) = 0$. This heuristic only has a linear time and space complexity. Experiment results show that, for our evaluated benchmarks, this heuristic can provide the same buffer allocation solution as the optimal solution, if paged buffer allocation is not considered (the optimal solution

can not take the paged allocation into consideration since the transition in the dynamic programming assumes that the buffer can always be allocated if there is enough space, regardless of the space fragmentation problem)

Once the allocation decision is successfully made, ABM follows the allocation solution to allocate the buffers in the NUCA banks. As in BiC [62], each L2 cache line has a bit to indicate if it is allocated to a buffer. Allocating a cache line as buffer may result in L1 sharer invalidations and dirty block write-back. If the DIG allocation fails, ABM will leave out the last request and invoke the DIG allocation again on the reduced set of requests. The removed requests are put in an outstanding queue. Once a buffer free event happens, ABM will allocate them. Otherwise, they will be allocated with the requests accumulated in the next interval (as the earliest requests in that interval). If the paged allocation of the minimum buffer size of a buffer allocation request fails at the upper bound of BiN space (discussed in Section 7.1.4), the core needs to perform the task without calling accelerators. We use the algorithm in [64] to obtain the accelerator BB-Curves for various input sizes.

7.1.3 Flexible Paged Buffer Allocation

In BiN, we compose non-contiguous spaces to satisfy the buffer requests. We propose that once a buffer is allocated, the accelerator will use a small local page table to translate buffer addresses into absolute addresses that can be found in NUCA. The key point to achieving this is to set the page granularity for each buffer according to the buffer size; i.e., a larger buffer may have a larger page size so that the total number of pages for this buffer is still a fixed number (32 in our evaluated system). The allowed page size is always a power of 2 to simplify translation. In our evaluated system it ranges from 4KB to half of the L2 cache bank size (32KB). A page must start at an address that is a multiple of the min-page and should not span cache banks. Since all of the buffer allocation and free operations are performed by ABM, it locally keeps the information about the current contiguous buffer spaces of each cache bank. To allocate a buffer with size S, ABM uses

the smallest page that is no smaller than S/32 to try the allocation, starting from the cache bank nearest to this accelerator, to the farthest cache bank. The amount of cache lines that can be used in each cache bank will be discussed in Section 7.1.4. To try the allocation of a set of buffers, ABM processes the buffers in a decreasing order of the buffer size, since a larger buffer may be more difficult to fit. If any buffer in this set fails to allocate, the paged allocation for this set fails. An example of the flexible paged buffer allocation is shown in Figure 7.5, where the min-page is 1 way of an 8-way set-associative cache bank.

To reduce the page fragments, we allow the last page (source of page fragments) of a buffer to be smaller than the other pages of this buffer, since this does not affect the page table lookup. For example, in Figure 7.5, the last page of Accelerator 0 is only half the size of its other pages. Therefore, the max page fragment for any buffer is smaller than the min-page. Note that the page fragments do not waste capacity since the cache lines in these fragments will remain as cache blocks and be used by the cache. For example, in Figure 7.5, the shaded blocks denote the page fragments. Then Sets 2 and 3 of Cache bank 0 actually have five cache lines.

7.1.4 Buffer Allocation in NUCA

In this work we assume a static NUCA design (statically-mapped addresses to banks). Since the buffers are allocated on-demand, the boundary between the cache and accelerator buffers is floating. When BiN allocates buffers in the cache, it could easily consume all or most of the cache space to maximize accelerator gains. To limit the impact on cache performance, we impose an upper bound on the total buffer size that can be allocated. In our implementation where each cache bank is 8-way set-associative, BiN may vary the upper bound from 1/8 to 7/8 of the NUCA size with a 1/8 increment at each step. The upper bound can be controlled by existing cache partitioning schemes (e.g. [65]) where the BiN upper bound is simply one of the partitions competing for space. The accelerators BB-Curves collected in each partitioning interval can be used to estimate the

potential variation of off-chip access counts when making partition decisions. BiN can definitely benefit from dynamic upper bound tuning through smart cache partitioning, but our experimental results show that an upper bound set to half of the NUCA size achieves the best or close to the best performance for most of our evaluated benchmarks. Since the focus of this work is to show the gain of the DIG buffer allocation and the flexible paged buffer allocation over the prior work, we always set the upper bound to be half of the NUCA size unless otherwise declared, in order to make fair comparisons.

To avoid creating high contention in a particular cache bank, the upper bound is uniformly distributed to each cache bank; i.e., if the upper bound is half of the NUCA size, then in each cache bank at most half of the cache ways can be allocated as buffers. If high contention still occurs in some cache banks, we use the page re-coloring scheme [66] to remap the OS pages originally mapped to the cache banks to other underutilized banks in order to reduce contention. Other cache bank utilization balancing techniques (such as [67]) can also be used. We would like to emphasize that BiN is orthogonal to and compatible with most state-of-the-art NUCA management schemes since BiN only considers cache bank adjacency while allocating buffer pages in NUCA.

7.1.5 Hardware Overhead

The hardware overhead introduced by BiN on the evaluated ARC is mainly in ABM (buffer allocation) and the accelerators (look-up of buffer page locations in NUCA).

The block diagram of ABMs buffer allocation module is shown in Figure 7.6(a). The SRAM table to store the contiguous spaces information for each cache bank is 7-entry. This is because there can be at most 7 contiguous spaces in a 64KB cache bank with a min-page of 4KB, as shown in Figure 7.6(b). Each entry has 10 bits for the starting block address and 4 bits for the space length in terms of min-page. There are also 8 SRAM tables to store the BB-Curves of the buffer requests (DIG scheme processes at most 8 requests in a batch). We limit BB-Curves to have at most 8 points. Each point uses 2B for the buffer size and 3B for the buffer utilization efficiency. Thus, the total storage

overhead is 768B. According to the Synopsys Design Compiler (SAED library, 32nm), the buffer allocation logic has an area of 9,725um2 under a cycle time of 0.5ns, and the storage area is 3,282um^2 based on Cacti [28] (32nm). Thus, the total area of the buffer allocation module is less than 0.001% for a medium size 1cm2 chip. An average latency of 0.6us (1.2K clock cycles at 2GHz) is required to perform the buffer allocations. The initial 10K (waiting time in the DIG allocation) + 1.2K cycles of allocation time is only 1% of the typical accelerator runtime (in order of million cycles).

Figure 7.7 depicts the local page table of the accelerator and the mechanism to generate the block addresses in NUCA. Since we have 32 banks (5-bit bank ID) and each bank has 1024 blocks (10-bit block ID), each table entry is 15-bit and the page table is 64B. This table has an area of 373um2 based on Cacti [28], which is less than 1% of the area of our evaluated smallest accelerator (denoise: 496,908um2). The table access latency is 0.14ns. Since the clock cycle of the evaluated accelerators is 2ns, this latency can easily fit in the pipeline of block address generation. Note that the 15-bit adder in Figure 7.7 also exists in a shared buffer design with contiguous space allocation. This is not an overhead of BiN.

BiN does not introduce any more micro-architectural complexity to the L2 cache controller than BiC [62] does. For buffer allocation in a cache bank, BiN uses the same way-based approach as BiC (it uses cache lines uniformly from one way, and then uses the next). Moreover, the overhead of the cache partition scheme and the cache-bank-utilization balancing scheme are not considered as the overhead of BiN, since BiN are orthogonal to these techniques.

7.2 Compiler-Based BB-Curve Analysis

This section describes the generation of the BB-Curve for synthesized accelerator modules based on the compiler-based static analysis. The input is the C-code of the accelerators which is used to generate RTL accelerators through a high-level synthesis design flow. According to the access pattern analyzed by the compiler infrastructure, the tradeoffs

between the off-chip bandwidth requirements and on-chip buffer size are explored to generate the BB-curve for the SPM allocation optimizations in BiN.

7.2.1 Analysis Flow Overview

Data-intensive applications always have repeated accesses to the same array element in the program: this is one of the major causes of the critical off-chip memory bandwidth utilization. Data reuse is the efficient technique that is widely used to reduce the use of off-chip bandwidth usage by using an on-chip reuse buffer. The possibility of data reuse can be statically analyzed by a compiler-based design flow shown as Figure 7.8.

The input to the BB-Curve analysis is the C-code program which can be synthesized into hardwired accelerators. We primarily care about the off-chip memory access references and their surrounding loops in the input program for data reuse optimizations. A traditional compiler infrastructure, for example ROSE [68] is used to parse this information from the source code, and is expressed into the polyhedral model. The polyhedral model represents the loops and array access references into a linear form, and also provides a set of functionalities to perform transformations for the loops and array references in an efficient way: this makes it easy to statically calculate the bandwidth saving and buffer size for data reuse.

Data reuse tradeoffs are exploited and evaluated based on this information in the polyhedral model. First a data reuse graph is built to express the data reuse candidates in the program. Second, the access count saving and buffer size is calculated based on the polyhedral model. Third, for each given bandwidth requirement, the minimum buffer size for the requirement is calculated by an optimization process. And after that, the BB-Curve for the accelerator module is generated. To simplify the evaluation, we assume that the computation core has a fixed throughput so that the off-chip bandwidth is proportional to the off-chip access count.

7.2.2 Data Reuse Modeling

The data reuse graph (DRG) is widely used to represent data reuse candidates [69–72], as Figure 7.9(a) shows. Nodes of the graph are array references, and edges are the data reuse between the nodes. Nodes are weighted by the access count (AC) of the reference, and edges are weighted by the reuse buffer size (BS). A data reuse buffer is allocated to store the data accessed by the source node of a reuse edge, and then is accessed by the target node for the reused data. Each edge in a DRG represents a data reuse buffer candidate which can be allocated in on-chip memory to save off-chip bandwidth.

For a read node, its bandwidth is saved if there is a reuse edge allocated whose target node is this read node. Bandwidth of the write node can even be saved if all the read nodes of the same array are reused and the data is not the primary output of the design. This means that the data of this array will be always in on-chip memory. We simplify the DRG by pruning sub-optimal buffer allocations by only considering the reuse from the temporal nearest neighboring node, as shown in Figure 7.9(b), which has the minimal buffer size for each specific node.

Figure 7.10 shows an example of data reuse between two array access references from A[i,j] to A[i-2,j]. The on-chip reuse buffer size is 2N, because the data fetched by reference A[i,j] will be used by reference A[i-2,j] after two loop i iterations, and the 2N new data elements accessed during this period (two loop i iterations) need to be stored in the reuse buffer for continuous data reuse. After the data in the buffer is reused, it can be replaced to store new reusable data. The modulo operation in the reuse buffer addressing indicates that the buffer is accessed and updated in a cyclic way. By allocating the reuse buffer, off-chip memory accesses by reference A[i-2,j] are saved.

Table 7.1: System configuration of Simics/GEMS simulation

Core	4 Ultra-SPARC III-i cores @ 2GHz
L1 data & instruction cache	32KB for each core, 4-way set-associative, 64B cache block, 3-cycle access latency, pseudo-LRU, MESI directory coherence by L2 cache
L2 Cache (NUCA)	2MB, 32 banks, each bank is 64KB, 8-way set-associative, 64B cache block, 6-cycle access latency, pseudo-LRU
Network on Chip	4x8 mesh, XY routing, wormhole switching, 3-cycle router latency, 1-cycle link latency
Main Memory	4GB, 1000-cycle access latency

7.3 Evaluation Methodology

7.3.1 Simulation Infrastructure

We use the HSI simulator described in Section 7.1.1 to evaluate the system performance. Table 7.1 shows our simulated system configuration.

7.3.2 Benchmarks and Accelerators

We chose applications from one domain (medical imaging) to accelerate, since accelerator-rich architectures (such as ARC) are most suitable for domain-specific computing [8, 73]. We chose the medical imaging domain because it consists of applications that are both compute- and memory-intensive, and the highly regular computation of the domain makes it an ideal target for hardware acceleration. Furthermore, improved performance in medical imaging has a tremendous potential to transform health care. The chosen applications are: denoise, deblur, segmentation, and registration. These applications are explained in detail in [74] and form a medical imaging pipeline.

We use the methodology of [8] to extract the computation- intensive tasks of these applications, synthesize these tasks as ASIC accelerators by using a high level synthesis tool Vivado [37] from Xilinx, and obtain their cycle-accurate modules. We then plug these modules into our simulator. Each of these accelerators has at least 4 copies on the chip to allow threads calling the same type of accelerator to run simultaneously. All of the accelerators work at a frequency of 500MHz. The non-computation-intensive and control tasks of these applications and the Solaris-10 OS are running on the general-purpose cores.

7.3.3 Reference Designs

We compare BiN to the following representative schemes from prior work. All are evaluated under the same area constraint. To make a fair comparison, we set a fixed upper bound of BiN as half of the NUCA size when comparing to the prior work. We will discuss the impact of dynamic upper bound tuning in Section 1.4.4.

- **Accelerator Store (AS) [61]:** In AS the shared buffer and NUCA are two separate units. We set the 32-bank NUCA size as 1MB (since we use an upper bound as half of the NUCA size in BiN). Because buffers in cache have some area overhead compared to separate buffers, we set the capacity of the shared buffer in AS as 1.32MB (32% larger than the maximum buffer size in BiN under the same area constraint) based on Cacti [28]. We also partition the shared buffer into 32 banks to increase the buffer access ports and these banks are distributed to the 32 NoC nodes.

- **BiC [62]:** BiC dynamically allocates contiguous cache space to a buffer. Here we set an upper bound for BiC by limiting buffer allocation to at most half of each cache bank. To allow a buffer to span multiple cache banks, the end of the BiC space in one cache bank is considered to be contiguous to the beginning of the BiC space in the next cache bank. The system configurations of BiN and BiC are the same except that the DIG allocation and the flexible paged allocation are

not available in BiC. This scheme is used to show that simply allocating buffers in NUCA will result in space fragmentation and underutilization.

We use the off-line compiler analysis in Section 1.2 to generate the desired buffer space of a set of potential input sizes for both AS and BiC to achieve their best performance. In addition, to demonstrate the innovations of BiN step by step, we also construct the following schemes. 1) **BiN-Paged**. It only uses the proposed flexible paged allocation scheme (32-entry page table). 2) **BiN-Dyn**. Based on BiN-Paged, it performs dynamic allocation without consideration of near future buffer allocation requests; i.e., it just responds to a buffer allocation request immediately by greedily satisfying the request with the current available spaces. Thus, a new buffer allocation request may not be satisfied since a preceding buffer has consumed most of the spaceeven if the new requesting accelerator can use the space more efficiently. 3) **BiN-Full**. This is the entire proposed BiN scheme.

7.4 Results

We conduct experiments for different degrees of buffer pressure by varying the number of medical imaging pipelines that we run in parallel (1, 2, and 4 pipelines). Each pipeline consists of one thread sequentially executing the four medical imaging applications on a different set of imaging data (i.e., no data dependencies between pipelines), where the image sizes also vary. Our benchmark naming convention indicates both the number of concurrent pipelines and the image size for that particular run. For example, benchmark 4P-28 means that there are 4 copies of the pipeline running in parallel and the input to each is a unique image that is 28x28x28 pixels. Benchmarks from 1P-28 to 4P-100 feature pipelines that are running with the same input image sizes. Thus there is no buffer fragmentation problem. These benchmarks are mainly used to demonstrate the gain of the DIG allocation. The 4P-mix label indicates that the various pipelines in the benchmark have randomly selected input image sizes. Computation over varied image sizes exhibits variation in the buffer demand and duration of buffer use, which in turn

results in fragmentation; thus they are used to demonstrate the gain from both the flexible paged allocation and the DIG allocation.

7.4.1 Impact of DIG Buffer Allocation

Figure 7.11 and Figure 7.12 show the comparison results of runtime and off-chip memory access counts. All of the results in Section 1.4.1 to 1.4.3 are normalized to that of AS (Accelerator Store). In the first 12 benchmarks, since there is no buffer fragmentation, AS, BiC, and BiN-Paged behave similarly (the impact of buffer-to-accelerator distance will be discussed in Section 1.4.2).

By greedily satisfying the buffer requests with currently available resources, BiN-Dyn outperforms AS, BiC, and BiN-Paged in 1P cases and 2P/4P cases with small inputs, because the shared space can accommodate these small buffers even if they are greedily satisfied. This gain can also be confirmed by the bandwidth reduction in these cases (shown in Figure 7.12). However, in the cases where greedily satisfying the first buffer requests will severely reduce the available space for subsequent requests that may more efficiently use the buffer space, BiN-Dyn behaves considerably worse compared to the first three schemes, as can be seen in the 2P/4P cases with large input sizes. Interestingly, the off-chip memory access counts do not increase correspondingly. The reason is that, when BiN-Dyn allocates a large buffer for the initial requests, it will delay subsequent requests from accelerators with large input sizes. Eventually these subsequent requests will be assigned to a large buffer (for reducing off-chip bandwidth), but their execution is serialized and thus the performance is impacted. Therefore, a reduction of the total off-chip memory accesses may not necessarily result in an increase in performance.

BiN-Full consistently outperforms the other schemes because of the DIG allocation (and also the flexible paged allocation in the 4P-mix cases). The only exception is in 4P-mix3, where the 1.32x larger capacity of AS is just large enough to accommodate all buffer requests; whereas BiN-Full needs to allocate a smaller buffer size to the accelerator that has the smallest buffer utilization efficiency. But it still outperforms the other three

schemes that have the same total capacity. Overall, compared to AS and BiC, BiN-Full reduces the runtime by 32% and 35%, respectively.

7.4.2 Impact of Paged Buffer Allocation

In the 4P-mix cases where there are buffer fragmentation problems, BiN-Paged and BiN-Full improves the runtime (up to 24% and 56%, respectively) in 4P-mix1,4,5,6. However, the bandwidth is not reduced by BiN-Paged and BiN-Full correspondingly, because although AS, BiC and BiN can allocate similar buffer sizes (thus similar bandwidth), BiN can allocate the buffer much earlier than BiC and AS, since BiN can aggregate non-contiguous space to satisfy the parallel buffer requests while BiC and AS can not. In 4P-mix2,3, the 1.32x larger capacity of AS can accommodate the buffers even when there is fragmentation. Thus BiN-Paged behaves similar to or even worse than AS. But in both cases, BiN-Paged outperforms BiC, which has the same capacity.

Since the first 12 benchmarks do not suffer buffer fragmentation, BiN-Paged improves their runtime by allocating buffers closer to the accelerator. However, the improvement is small. The reason is as follows. The buffer accesses from our evaluated accelerators are mainly reading in large amounts of input data, performing some calculation on the data and then writing the transformed data out to the buffers. These accesses do not have inter-dependencies. They are pipelined so well that buffer access latency is completely hidden. Therefore, as shown in Figure 7.13, even though BiN-Paged can improve the average buffer access latency by 19%-32%, the runtime gain is only 2%-9%. We expect that accelerators from other domains which may have dependencies among the buffer accesses can obtain more benefit from adjacent buffer allocation.

7.4.3 Impact on Energy

We obtain the dynamic and leakage energy data of NUCA and the main memory through Cacti [28] and McPAT [23], and the power data of the ABM module via the Synopsys Design Compiler [22]. We back-annotate these numbers into our simulator to obtain the

memory energy results, as shown in Figure 7.14. By separating cache and buffer units, AS consumes less energy for each cache/buffer access and also consume less unit standby leakage than the other schemes. Therefore, BiC and BiN-Paged consume more energy than AS. BiN-Dyn can save energy in cases where it can reduce the off-chip memory accesses and runtime. However, in the cases where BiN-Dyn significantly increases the runtime, it consumes considerably more energy (more standby energy). By performing DIG and flexible paged buffer allocation, BiN-Full can reduce both the number of off-chip memory accesses and the runtime. When compared to AS, it sees a 12% reduction in energy on average. In cases where the 1.32x capacity of AS can better satisfy buffer requests (4P-mix3), BiN-Full consumes more energy due to more off-chip memory accesses and longer runtime. BiN-Full reduces the energy by 29% on average compared to BiC.

7.4.4 Impact of DIG Allocation Interval Length

The DIG buffer allocation is triggered 1) once an interval end is achieved, or 2) once the number of collected buffer allocation requests achieves a predetermined value (8 in this work). If the allocation interval length is too short, BiN will behaves like BiN-Dyn which greedily satisfies each buffer allocation request using currently available resources, without considering near-future requests. If the allocation interval is too long, the buffer allocation requests that arrived earlier will wait for a long time to get their requested buffers. In the previous experiment subsections, we always use an empirical value of 10K-cycle DIG allocation interval. In this subsection, we will discuss the sensitivity of the system performance to this interval length.

Unlike the previous experiment subsections that strictly follow the dependencies among the medical imaging pipeline stages, here we assume that each stage of the medical imaging pipeline can be issued independently. Thus we can flexibly tune the buffer allocation request arrival rate to ABM. Moreover, we also extract some kernels (such as the polynomial computations) from these medical imaging pipeline stages, which run much faster than these pipeline stages, to make our case that the waiting time of the

10K interval length is non-trivial to these kernels.

We characterize the tasks that run on accelerators as small, medium and large, based on their estimated runtime. Note that here small tasks mean the aforementioned kernels, whereas the small input size that we used in the real benchmarks in the previous subsections falls into the medium and large categories. Basically, the small tasks can be finished within 20K-50K cycles; the medium tasks can be finished within 400K-1M cycles; and the large tasks can be finished within 3M-10M cycles. Moreover, we characterized the task arrival rates to ABM as fast, moderate, and slow, which stands for the average arrival rates of one task per 500 cycles, 5K cycles, and 50K cycles, respectively. Each task will issue a buffer allocation request to ABM. In sum, we have nice benchmarks of all 3x3 combinations, ranging from small-fast to large-slow. We compare the system performance (reverse of the average task runtime) of these benchmarks with the DIG interval length as 1K-cycle, 10K-cycle and 100K-cycle. The results are shown in Figure 7.15.

For small tasks, when the arrival rate is moderate and slow, a 1K-cycle interval length can significantly reduce the waiting time in an interval; thus, it has the best performance (27% better than the 10K-cycle and 67% better than the 100K-cycle). The 1K-cycle interval only has, on average, two buffer allocation requests served at each DIG allocation, which has less global optimality than the 10K-cycle and 100K-cycle intervals. However, the tasks finish quickly, and thus they do not hold up the following tasks too much. If the arrival rate is fast, the waiting times of 10K-cycle and 100K-cycle intervals are reduced remarkably since once the number of collected buffer allocation requests achieves a predetermined value (8 here), DIG allocation will be triggered. Thus the 10K-cycle and 100K-cycle intervals perform 13% better than the 1K-cycle interval because of more global optimality.

For medium tasks, the impact of waiting time begins to diminish, and the global optimality of DIG allocation becomes more important. When the task arrival rate is fast, 10K-cycle and 100K-cycle intervals outperform the 1K-cycle interval by 16%; the reason for this is similar to the small-fast case. When task arrival rate decreases to moderate, the average waiting time increases, but still cannot offset the benefits of more global

optimality; thus, the 10K-cycle and 100K-cycle intervals still outperforms the 1K-cycle interval. However, when the arrival rate is slow, the importance of an average waiting time reduction begin to outweigh the benefits of a more global optimality; thus, the 1K-cycle interval performs slightly better than the 10K-cycle and 100K-cycle intervals.

For large tasks, the global optimality of DIG allocation is much more critical than the waiting time reduction. Therefore, the 1K-cycle interval always performs the worst. At the relatively slower arrival rates, the 100K-cycle interval can collect a large enough number of allocation requests in one interval to perform global optimization. Therefore, it performs 15% and 11% better than the 10K-cycle interval in the moderate and slow arrival rate, respectively.

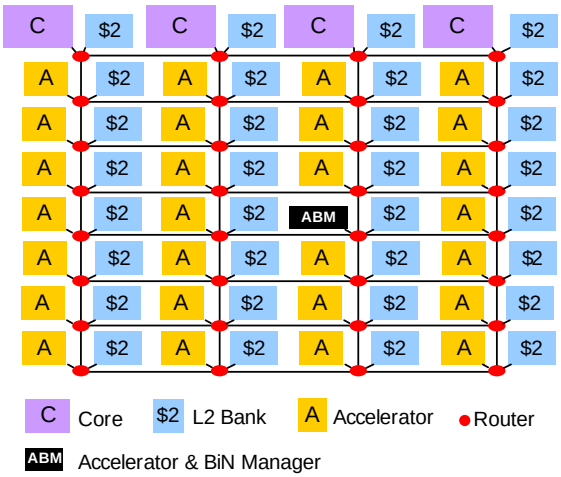
Overall, it can be seen that the short interval performs well in cases where task size is small and arrival rate is slow, while the long interval performs well in cases where task size is large. The applications in our evaluation are mostly medium and large tasks; therefore, in this work we choose an interval length of 10K cycles.

7.4.5 Impact on Cache

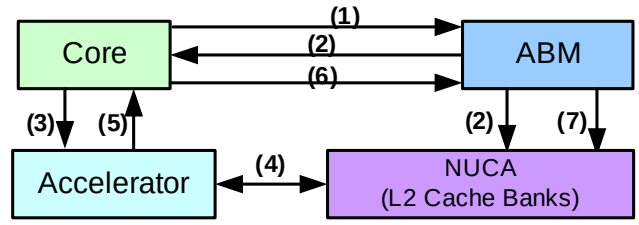
We further quantify the impact of the effective cache capacity reduction due to buffer allocation by running a set of general-purpose applications (SPEC CPU2006 benchmarks) on the cores concurrently. In the previous experiments we use a fixed BiN upper bound which is half of the NUCA size. Thus, we first evaluate the impact on SPEC benchmark runtime when half of the NUCA space is used by BiN, as shown in Figure 7.16. The results are normalized to full cache capacity performance (additional separate buffers are required for the accelerators). In most cases the runtime increase is within 10%, except milc which has a large working set. For milc, although a full-capacity NUCA with separate buffers has a remarkably small runtime, it almost doubles the on-chip memory energy and area, and has less flexibility.

The impact of BiN on the cache can be improved via dynamic upper bound tuning. We use the cache partitioning scheme proposed in [65] to dynamically tune the upper bound.

We run each SPEC benchmark simultaneously with two medical imaging pipelines for each of the 4 input sizes, respectively. The dynamic tuned BiN upper bounds for each case are shown in Figure 7.17. In most cases an upper bound of half of the NUCA size is selected. In the cases where the SPEC benchmarks can give more space to BiN without considerably impacting the performance, an upper bound of 5/8 NUCA size is selected for BiN. For these cases, the runtime of both SPEC benchmarks and the medical imaging pipelines (normalized to the runtime with an upper bound as half of the NUCA size) and the product of the two are shown in Figure 7.18. All of the runtime variations are within the range of 5%, which suggests that impact of the upper bound tuning is limited.



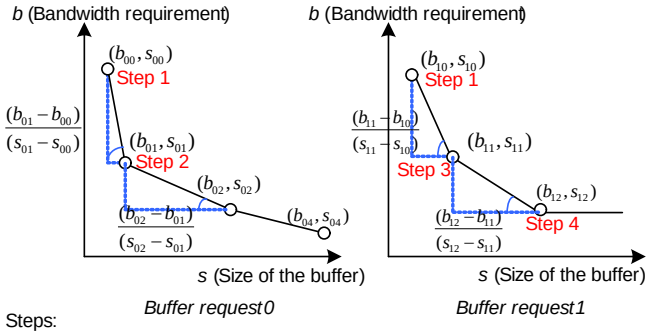
(a)



(1) The core sends the accelerator and buffer allocation request with the BB-Curve to ABM.
(2) ABM performs accelerator allocation , buffer allocation in NUCA, and acknowledges the core .
(3) The core sends the control structure to the accelerator .
(4) The accelerator starts working with its allocated buffer .
(5) The accelerator signals to the core when it finishes .
(6) The core sends the free -resource message to ABM .
(7) ABM frees the accelerator and buffer in NUCA .

(b)

Figure 7.3: (a) Overall architecture of AXR-CMP with BiN. (b) Communications between core, ABM, accelerator, and NUCA.



Steps:

1. Allocate s_{00} and s_{10}.
2. Increase buffer 0 to s_{01}, since $\frac{(b_{01}-b_{00})}{(s_{01}-s_{00})} > \frac{(b_{11}-b_{10})}{(s_{11}-s_{10})}$.
3. Increase buffer 1 to s_{11}, since $\frac{(b_{11}-b_{10})}{(s_{11}-s_{10})} > \frac{(b_{02}-b_{01})}{(s_{02}-s_{01})}$.
4. Increase buffer 1 to s_{12}, since $\frac{(b_{12}-b_{11})}{(s_{12}-s_{11})} > \frac{(b_{02}-b_{01})}{(s_{02}-s_{01})}$.
5. Try to increase buffer 0 to s_{02}, but the paged allocation failed, terminated

Figure 7.4: An example of the DIG buffer allocation scheme

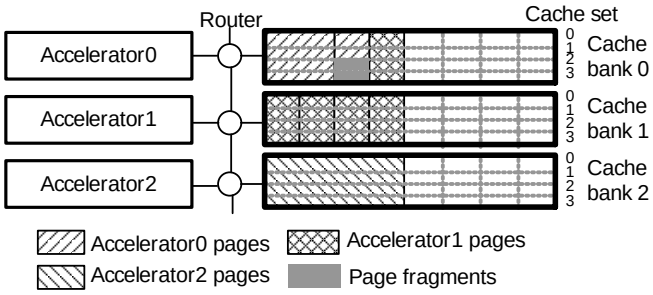


Figure 7.5: An example of the flexible paged buffer allocation

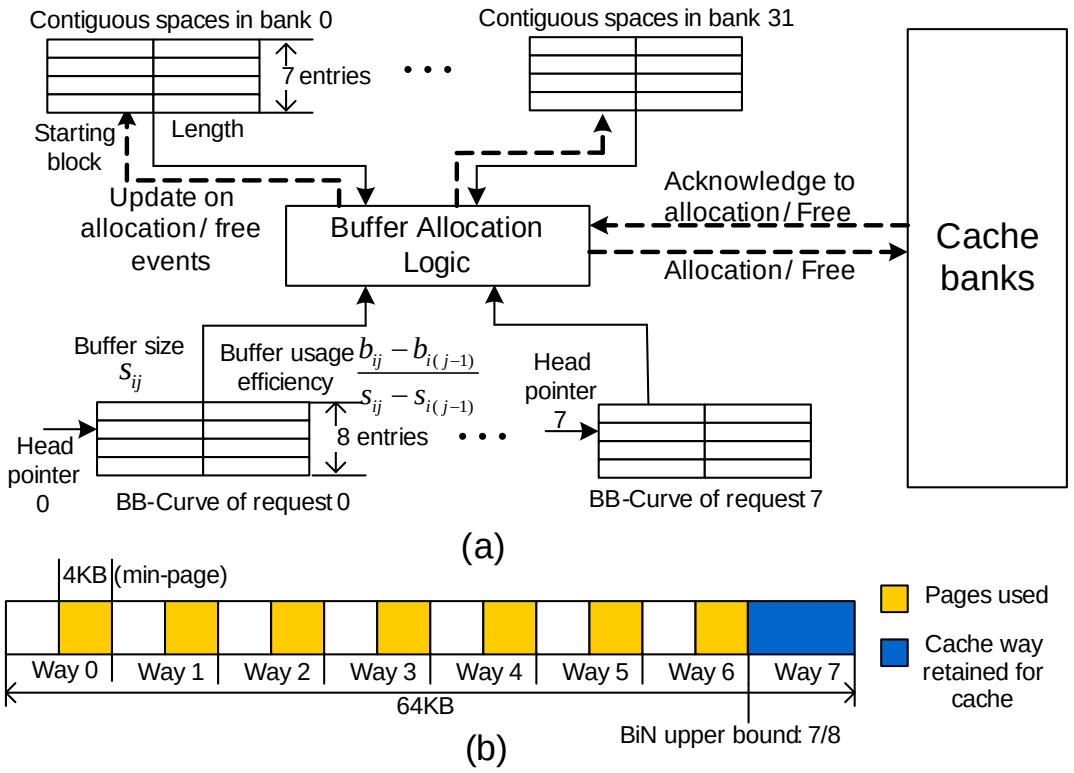


Figure 7.6: (a) The buffer allocation module in ABM. (b) Worst-case buffer fragmentation in a cache bank.

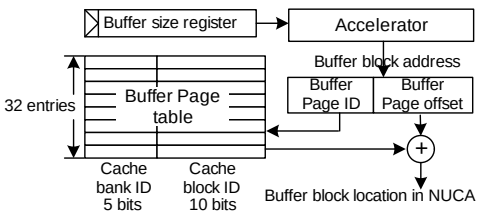


Figure 7.7: Block addresses generation with the page table

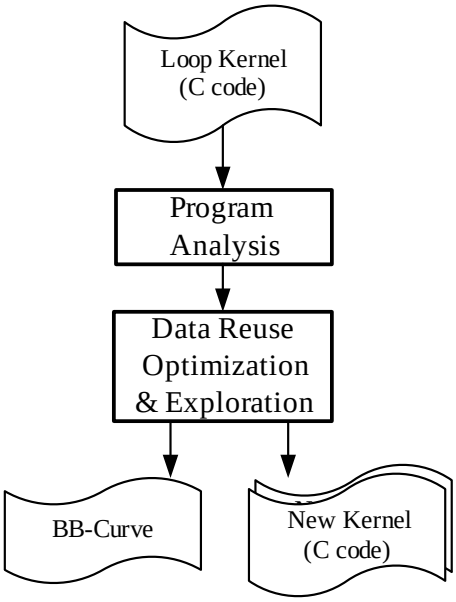


Figure 7.8: Data reuse analysis for BB-Curve

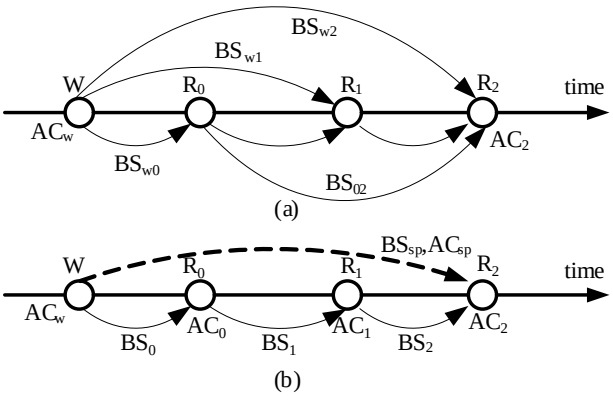


Figure 7.9: (a) Full data reuse graph. (b) Simplified data reuse graph

```
// original
for i=0 to N
   for j=0 to N
S1:   B[i,j]=f0(A[i, j], A[i-2, j]
                  A[i-3, j]);
for i=0 to N
   for j=0 to N
S2:   C[i,j]=f1(B[i, j],B[i, j-2],
                  B[i,j-3]);
```

```
data_type buf[2, N];
for i=0 to N
   for j=0 to N {
S1:   B[i,j]=f0(A[i, j],
                  buf[i%2, j], A[i-3, j]);
      buf[i%2, j] = A[i, j];
   }
for i=0 to N
   for j=0 to N
S2:   C[i,j]=f1(B[i, j],B[i, j-2],
                  B[i,j-3]);
```

Figure 7.10: (a) Full data reuse graph. (b) Simplified data reuse graph

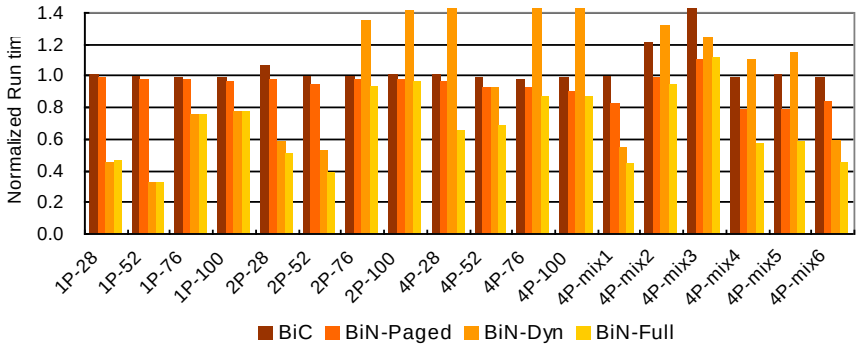


Figure 7.11: Comparison results of runtime

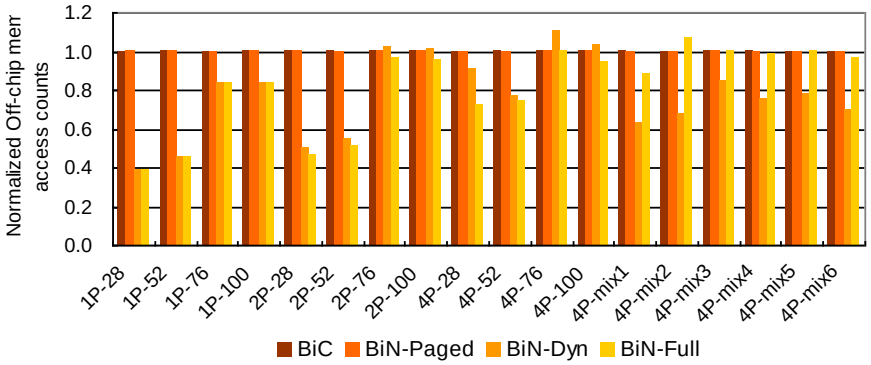


Figure 7.12: Comparison results of off-chip memory accesses

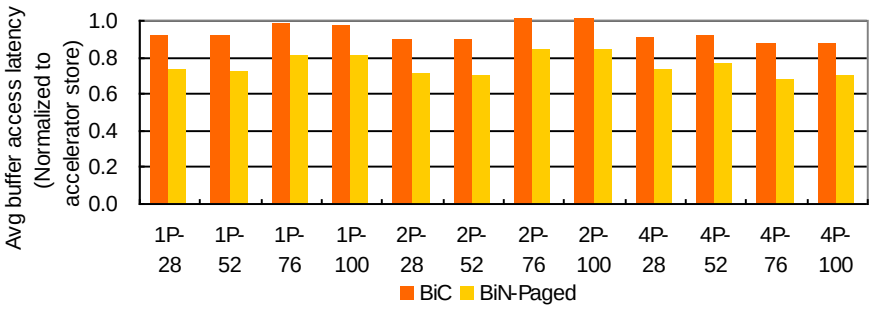


Figure 7.13: Comparison results of buffer access latency

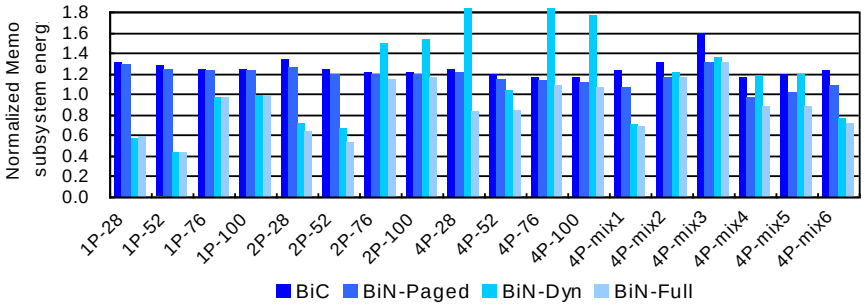


Figure 7.14: Energy comparison of the memory subsystem

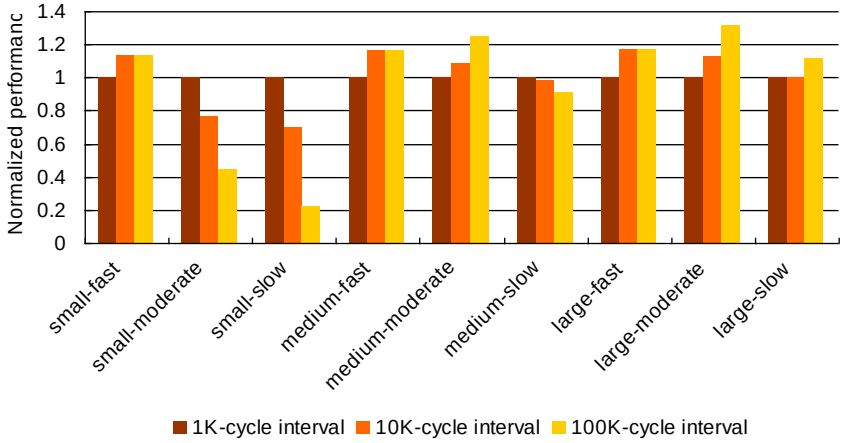


Figure 7.15: Impact of DIG allocation interval length

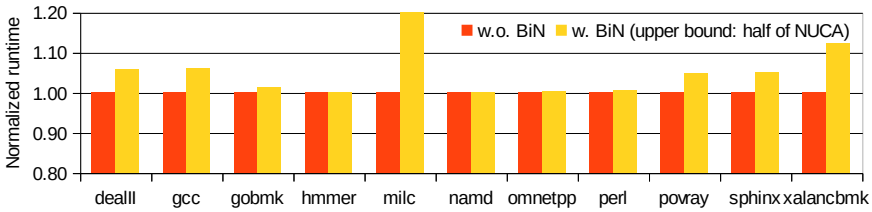


Figure 7.16: Impact of BiN with a fixed upper bound (half of the NUCA size) on the runtime of SPEC benchmarks.

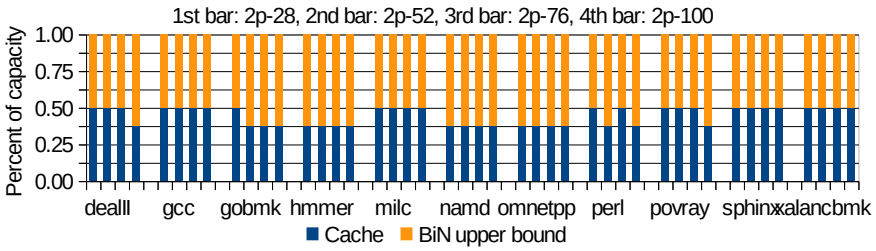


Figure 7.17: Partitions via dynamic upper bound tuning

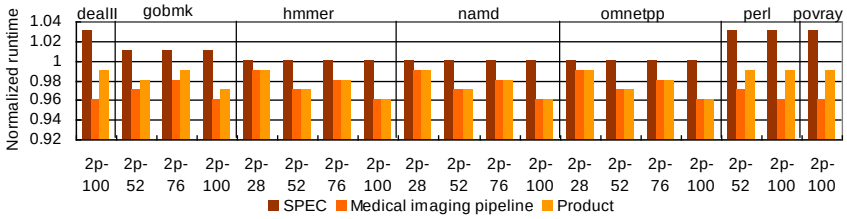


Figure 7.18: Runtime of dynamic partitioning compared to a fixed upper bound (half of NUCA)

CHAPTER 8

Stream Arbitration: Towards Efficient Bandwidth Utilization for Emerging On-Chip Interconnects

Emerging interconnect technologies, such as optical network [75, 76] and radio frequency interconnection(RF-I) [77] networks, have been considered recently as potential replacements for traditional copper wires for long-haul NoC lines. These technologies offer enormous bandwidth potential, low energy cost per bit transmitted, and low latency, when compared to traditional networks. Drawbacks of these technologies are two-fold: 1) These are analog networks, and thus require digital to analog conversion to transmit and receive data, and 2) the bandwidth capacity of these networks is so large as to make in-flight inspection of packet content, and thus traditional packet-switched routing, impossible. For this reason, we introduced STREAM [78], which is a communication protocol designed to operate efficiently within these design constraints. As will be shown, STREAM features a chip-wide communication network that allows for high-bandwidth transmission between source-destination pairs. STREAM was the first protocol and design considering emerging network technology resources that allowed for the high performance offered by these technologies to be exploited successfully.

This technique is particularly attractive in the context of accelerators, which were used as a driver for this study. The reason for this is that, while it is relatively simple to scale up the capability of an accelerators compute engine, feeding data to this compute engine has been consistently identified as a challenge. STREAM assists in addressing this by offering massive amounts of burst bandwidth while an accelerator is in use, while not exhibiting many of the short-comings typically associated with NoC hotspots in large scale NoC topologies.

In this work, we used RF-I as a driver, but the methodologies discussed in this chapter are applicable to optical networks as well. RF-I was chosen due to the existence of publications on the topic of RF-I networks, which allowed us to model our system at a level of detail that would have been impossible if we assumed an optical network instead, which had yet to have been successfully integrated into CMOS technology as of the time that this work was done.

8.1 Introduction

As we enter the era of many-core and beyond, the number of cores, co-processors, and on-chip accelerators grows rapidly. The dramatic increase of these processing elements (PE) imposes a tremendous bandwidth requirement on the communication to the memory/cache [79]. This communication is typically accommodated via an on-chip interconnection network (or NoC: Network on Chip). It has been observed that electronic networks cannot efficiently supply the dramatically increased PE-memory communication bandwidth due to unacceptable power and area consumption [79]. Therefore, alternative interconnectssuch as radio-frequency interconnect (or RF-I) [77], and optical interconnect [75, 80]have become more attractive as a means to scale bandwidth and latency in a power efficient manner. However, efficient utilization of the on-chip communication bandwidth provided by these emerging interconnects still remains an open problem due to nonuniform and temporally irregular traffic patterns in current and future CMPs.

There are two main approaches to dealing with the allocation of the on-chip communication bandwidth for emerging interconnects. Both of them partition the aggregate bandwidth into a set of communication channels. The first approach allocates these channels as application-specific shortcuts [81] that are overlaid on the baseline traditional NoC to facilitate critical and heavy-loaded communication paths. This addresses the concern of spatial nonuniformity in NoC traffic. Off-line compiler optimization or profiling methods are used to predict the communication pattern of program phases. Since the shortcuts are realized by tuning the frequency of the transmitter-receiver pairs,

they can be reconfigured for each application or each phase of an application to match the changes in program characteristics, and can accommodate a degree of change in traffic patterns effectively. This is based on the assumption that the dynamic communication pattern changes can be predicted a priori, and are sufficiently rare to justify the cost of reconfiguring routing tables. However, in modern CMP designs, the data layout in the last level cache (typically designed as NUCA, non-uniformed cache architecture) is dynamically determined by OS through virtual-to-physical page translation [82]. Moreover, threads may be dynamically migrated to fully utilize the on-chip core resource [83] and cache blocks may also be dynamically migrated [84] or replicated to reduce cache access latency. In a recently investigated composable accelerator-rich CMP [24], a virtual accelerator called by a thread is dynamically composed by the available on-chip accelerator building blocks, which cannot be pre-determined. Moreover, in an accelerator-rich CMP, buffers may be dynamically allocated in any L2 cache bank based on the distance to the accelerator [63]. All of these complications make it unrealistic to enable an accurate prediction of the on-chip communication pattern. Therefore, although this shortcut approach can efficiently address the spatial communication heterogeneity, it cannot effectively handle temporal communication heterogeneity.

The second approach uses token-based dynamic arbitration [75, 80] to allocate the channels at the real-time to communicating pairs on demand. Each receiver node in the NoC has its own channel which serves as its home node. The home node injects a token into its channel and any senders that want to communicate to this home node can perform a destructive read of the token to acquire the communication channel, and then reinject the token once it finishes this one-time communication. Unlike the aforementioned shortcut approach, this approach allocates the bandwidth on-demand at runtime with low arbitration latency, power, and hardware cost, satisfying the real-time communication requirement. If all of the receivers are uniformly receiving packets, it also tends to high utilization and fair sharing. However, this is not the case for modern and future CMPs due to the spatial communication heterogeneity.

Figure 8.1 depicts the high variation in number of received network flits per node

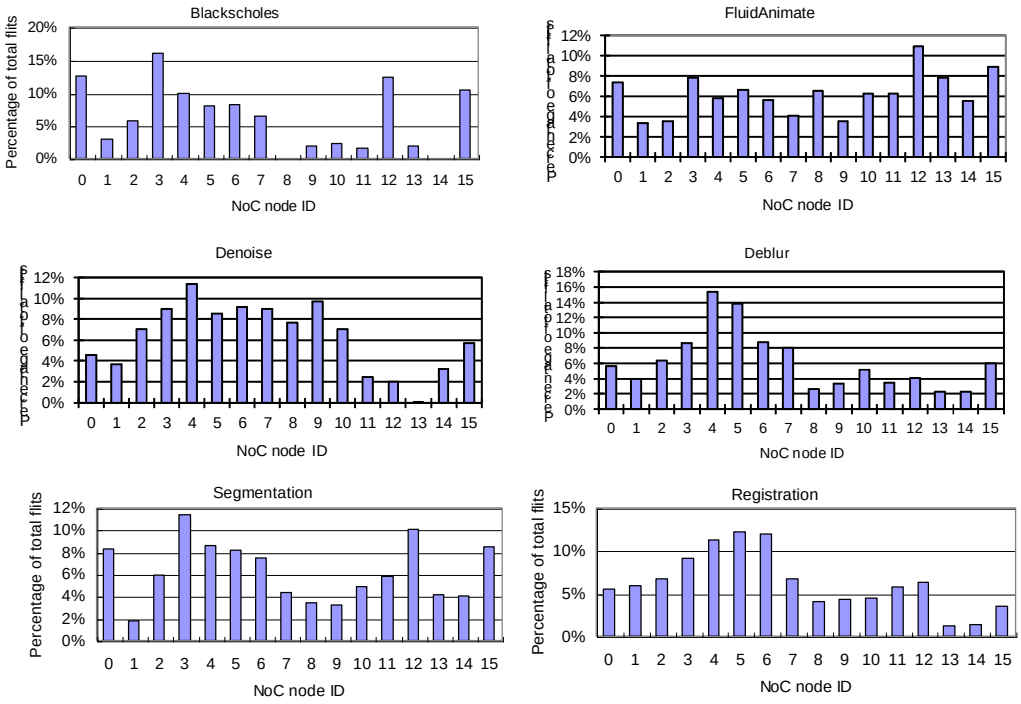


Figure 8.1: Percent of received flits per node in a Token-based RF-I NoC

for a NoC using RF-I with a token-based arbitration scheme. The results shown are complete executions of each benchmark (a detailed system description is in Section 8.4). Each histogram denotes the received flits for one node over the total number of flits of the application (i.e. the percent of total flits received by a given node). The coefficient of variation for these applications ranges from 33% to 80%. This variation is even more significant if instantaneous demand is examined instead of aggregate demand.

In this situation, the channel bandwidth of the nodes that receive few packets is wasted, while the channels associated with the nodes receiving a large amount of packets are bandwidth-hungry. It may be argued that the bandwidth allocated to each channel can be pre-determined based on the home nodes communication workload, but as mentioned earlier in discussing the shortcut approach, the temporal communication workload is difficult to predict a priori due to the dynamic uncertainties. Therefore, although this

token-based dynamic arbitration scheme efficiently solves the temporal communication heterogeneity, it cannot effectively handle spatial communication heterogeneity.

Therefore, it is necessary to find an efficient bandwidth utilization scheme which can deal with both spatial and temporal communication heterogeneity. In this paper, we propose a dynamic arbitration scheme called Stream Arbitration. Unlike token arbitration where channels are coupled to receivers, a channel in stream arbitration can be used to send packets from any sender to any receiver, which efficiently addresses the problem of spatial communication heterogeneity. Since stream arbitration is inherently a dynamic arbitration scheme, it also efficiently handles temporal communication heterogeneity. In this paper we choose RF-I to evaluate stream arbitration. RF-I is one promising alternative interconnect due to its compatibility with existing CMOS design process, thermal insensitivity, low latency, and low energy consumption [77]. But it should be noted that stream arbitration is not constrained to only RF-I: it can also be applied to other emerging interconnects such as optical interconnect. The main contributions of this work can be summarized as follows:

- To the best of our knowledge, steam arbitration is the first dynamic arbitration scheme which targeting emerging interconnect technologies can efficiently deal with both spatial and temporal communication heterogeneity. Starvation avoidance and flow control are also carefully considered.

- We propose a detailed circuit level design to realize stream arbitration in a radio frequency interconnect, with accurate modeling of area and power overhead.

- We develop a full-system cycle-accurate simulation infrastructure to evaluate the system performance and power impact of the stream arbitration scheme on various benchmarks and system configurations. Compared to one representative bandwidth allocation schemes for emerging interconnects from prior work: token based arbitration [80] (applied to RF-I), stream arbitration can provide an average 20% performance improvement and 12% power reduction.

8.2 Stream Arbitration: Scheme and Example

In this section we describe stream arbitration from an algorithmic and architectural point of view. We will discuss the circuit support of the operations required by stream arbitration, such as stream augmentation and circulation in RF-I in Section 8.3.

We partition the aggregate bandwidth provided by the RF-I or waveguide into several logical communication channels. One of them is used for arbitration, which is called the arbitration channel. The remaining channels are used for PE-memory data requests and responses, which are called data channels. Each RF node has one transmitter and receiver pair to access both the arbitration channel and data channels. Active sources (nodes that want to send flits) compete for the data channels in the arbitration channel to talk to their desired destination nodes. Arbitration is done for each flit that is transmitted.

8.2.1 Stream Arbitration Scheme

The key component of our approach is the arbitration stream that travels across the arbitration channel. Conceptually, the arbitration stream starts at a single node, which is called the stream origin. The arbitration stream starts out logically empty and will travel in a unidirectional manner across all the nodes on the chip, this is called **Trip 1**. In this trip, when the stream passes each node, the node logically augments a number of bits (referred to as substream) in the arbitration stream to specify whether or not this node is attempting to send to another node, and whether or not this node is capable of receiving packets. It is important to note that these two pieces of information (desire to send and availability to receive) do not require any parsing of the stream they only rely on information known a priori at the node. So there is no dependence where the stream must be read first and then modified such a dependence would impair the arbitration latency by bringing slower logic on the critical path of the stream propagation.

To ensure this decoupling and that nodes need only modify the stream without first reading it, each node has a specified region of bits that make up that nodes substream, and substreams are disjoint within the arbitration stream. Collectively, these disjoint

Algorithm 1 Stream Arbitration

INPUT: Stream:*flowControl*[1..N], *interested*[1..N], *destination*[1..N], where N is the number of RF nodes; the total number of channels M; this nodes ID *node_id*.

OUTPUT: *Transmitting_channel_ID*, *Receiving_channel_ID*.

Transmitting_channel_ID = INVALID;

Receiving_channel_ID = INVALID;

for i = 1..N **do**

  if interested[i] **and not** flowControl[destination[i]]) **then**

    flowControl[destination[i]] = TRUE;

    channel_ID++;

    if destination[i] = node_id **then**

      Transmitting_channel_ID = channel_ID;

    end if

    if channel_ID = M-1 **then**

      break;

    end if

  end if

end for

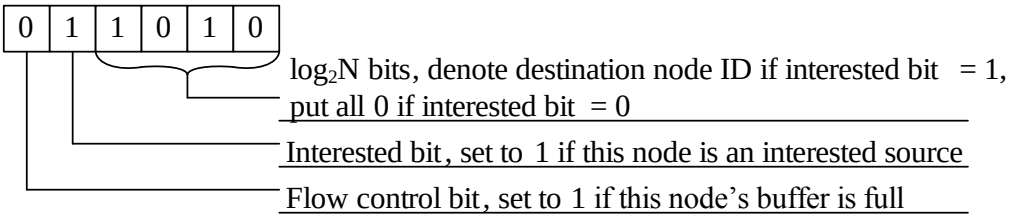


Figure 8.2: The substream augmented by each node as the stream passed by

substreams will represent each nodes interest in sending over a data channel and availability to receive from a data channel. The layout of a single substream element is shown in Figure 8.2. A node that wants to send a flit and is therefore contending for data channels is referred to as a source node. The destination ID is the label of the node to which a source node intends to send a flit (referred to as a destination node). The flow control bit indicates the whether or not there is sufficient buffer space in this node to accept a flit. N is the number of RF nodes.

After the arbitration stream passes the last RF node in Trip 1, it circulates over all nodes a second time, which we refer to as **Trip 2**. In this trip, when the stream passes each node, the node receives the arbitration stream but does not modify the stream. The purpose of Trip 2 is to parse the stream in order to check:

- *Ability to Send:* If this node is attempting to send a flit, information from the stream will be used to indicate whether this node can acquire a data channel, and if so, the data channel ID.

- *Receive Channel:* Determine whether this node will be receiving a flit, and if so, the data channel ID where this data will be arriving is computed from the stream.

The algorithm is very simple and straightforward. It parses the stream in the order of the augmentation of the bits. A source node can acquire a data channel to a destination node if:

- The flow control bit of the desired destination node is zero.
- There is no upstream node already sending to the desired destination node. In this context, upstream means that an earlier node in the unidirectional flow of the stream.
- There are still available data channels.

After the arbitration stream is parsed, the transmitter of a source node that has successfully acquired a data channel will be tuned to send on this channel, and the receiver of the intended destination node will be tuned to listen to the same data channel. A node can be a source and a destination simultaneously, using different channels. After Trip 2, the sources that successfully acquired data channels begin to use these data channels to communicate with their corresponding destinations. A channel is used for a single flit, and is surrendered. This does not incur a performance penalty because arbitration can be initiated every cycle, so a pair of nodes is allowed to communicate so long as the source continues to win arbitration. This requires a pipelined stream arbitration and data transferring. The latency of the two trip arbitration and the pipelining of the arbitration and data transferring in the physical design are detailed in Section 8.3.2.

It can be seen that the upstream nodes always have higher priority in the arbitration than the downstream nodes (nodes encountered later in the unidirectional flow of the stream). In order to introduce fairness into stream arbitration, we use a rotating prioritization scheme, where each node is gradually reduced in priority each cycle, until it reaches the lowest priority. Each cycle, the lowest priority node from the previous cycle becomes the highest priority node. This prevents nodes that are lowest priority from being starved during periods of high system load. Moreover, each node gradually reduces priority to reduce the likelihood of burst data transfers dropping suddenly from highest priority in one arbitration cycle to lowest priority in the next. We found this method allowed for flits associated with multi-flit messages to arrive at the destination subsequently without high transmission latency deviation. From an architectural point of view, this gradually priority reduction is achieved by rotating the stream origin in the

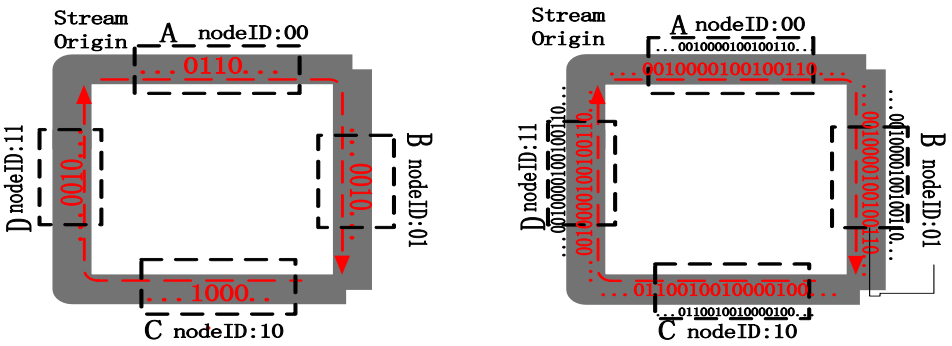


Figure 8.3: An example of the stream arbitration scheme

reverse direction of the stream traversal in the transmission line. However, we have a smart scheme detailed in Section 8.3.3 to support this without really rotating the stream origin.

8.2.2 Example of Stream Arbitration

To illustrate our approach, this section presents a single example arbitration attempt (Figure 8.3). This example consists of only 4 nodes participating in arbitration, one arbitration channel, one data channel, and assumes the stream origin is at the node A. Node A is attempting to send to node C, while node B and D are attempting to send to node A. C has no flits waiting for transmission, but has a fully occupied buffer and cannot receive any flits.

In Trip 1, each node augments its substream as described in Section 8.2.1 to the stream as showed in Figure 8.3(a) (The stream vectors follow the direction of arrow in the figure). Nodes A, B and D are source nodes in this arbitration cycle. They modulate the interested bit 1 and their respective destination node IDs when the stream travels by. The node C has no requirement for data transmission, and its receiver buffer is unavailable for new messages in this arbitration. So node C only sets the flow control

bit to 1 and leaves the interested bit at 0 when the stream travels by. Therefore the substreams to be modulated by the node A, B, C and D are 0110, 0100, 1000 and 0100, respectively.

In Trip 2 as shown in Figure 8.3(b), each node receives the full arbitration stream, and executes the algorithm presented in Section 8.2.1. Node A is the first node to modulate the stream and has the highest priority to acquire the data channel, but it cannot send because its destination, node C, has declared that it does not have available buffer space. Thus, node A loses in arbitration and does not receive a data channel. Then the next node in priority order, node B, wins the data channel in this arbitration since its destination, node A, has an available buffer. Node C does nothing, since its receiving buffer is full and it is not an active node. From parsing the stream, Node D knows the upstream node B gets the data channel, and there are no more channels left. So node D also loses in this arbitration.

After the arbitration, the winner, node B, will transmit the message through data channel, and others will retry in the next arbitration.

8.3 Stream Arbitration in RF-I

8.3.1 RF-Interconnect

Radio Frequency Interconnect (RF-I) was proposed in [85, 86] as a high bandwidth, low latency alternative to traditional interconnect. Its benefits have been demonstrated for off-chip, on-board communication [87] as well as for on-chip interconnection networks [88]. The two most distinct advantages of RF-I compared to traditional interconnects are that 1) instead of charging and dis-charging the whole wire to 1 or 0 as is done in a traditional electrical interconnect, which consumes substantial time and energy, RF-I modulates information on an electro-magnetic carrier wave which is continuously sent along the transmission line, and 2) instead of trying to aggressively expand baseband bandwidth, which often involves power-hungry compensation technique to achieve a flat channel

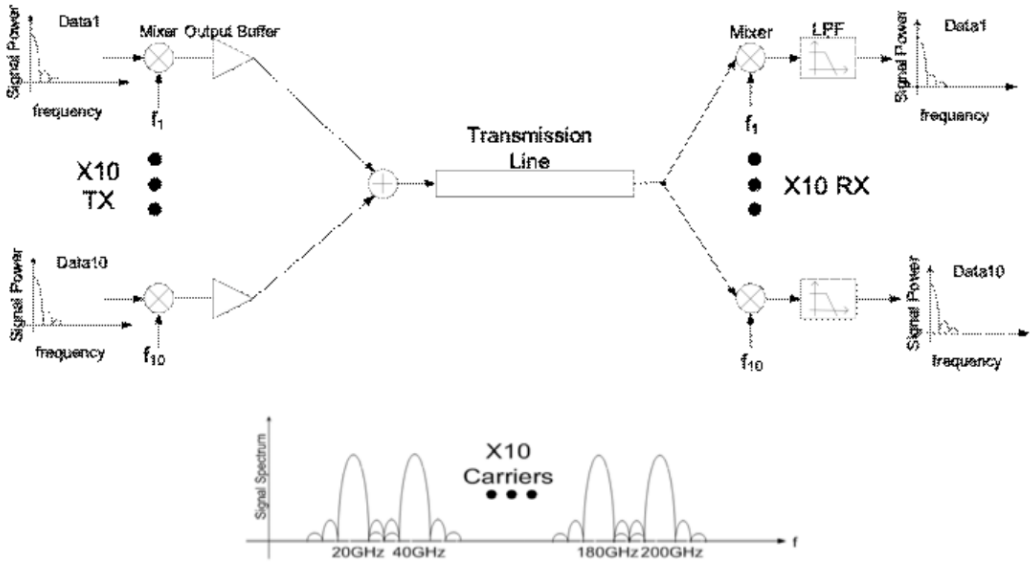


Figure 8.4: A ten carrier RF-Interconnect and corresponding waveform at the transmission line

frequency response, RF-I divide bandwidth in frequency domain, each of which becomes a narrow-band signal which saves power. By doing this, RF-I also improves bandwidth efficiency by sending many simultaneous streams of data over a single transmission line. This particular technique is referred to as multi-band RF-I. Also, RF-I has been projected to scale better than traditional RC wires in terms of delay and power consumption, and unlike traditional wires, it can allow signal transmission across a 400mm2 die in 0.3ns via propagation at the effective speed of light. Figure 8.4 shows an exemplary RF-Interconnect link with 10 bands transmitting on the same physical transmission line.

One key advantage of RF-I over traditional interconnects is its capability of multi-cast with on-chip directional couplers [88]. Impedance matched directional couplers eliminate signal reflection that has inhibited multi-cast on traditional interconnects. Figure 8.5 shows an exemplary RF-Interconnect multi-cast link of one band. In the case that higher aggregate data rate is desired for this arbitration channel, more RF channel can be added into this multi-cast with a similar fashion as shown in Figure 8.4, with multi-band

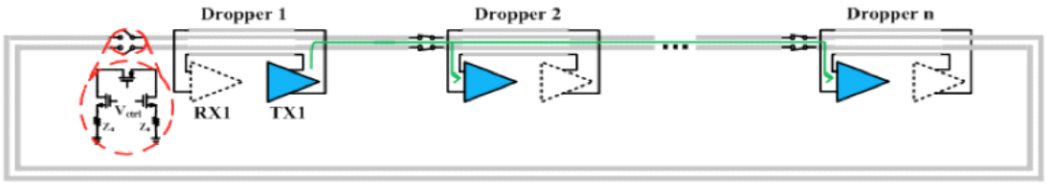


Figure 8.5: Multi-cast RF-Interconnect system

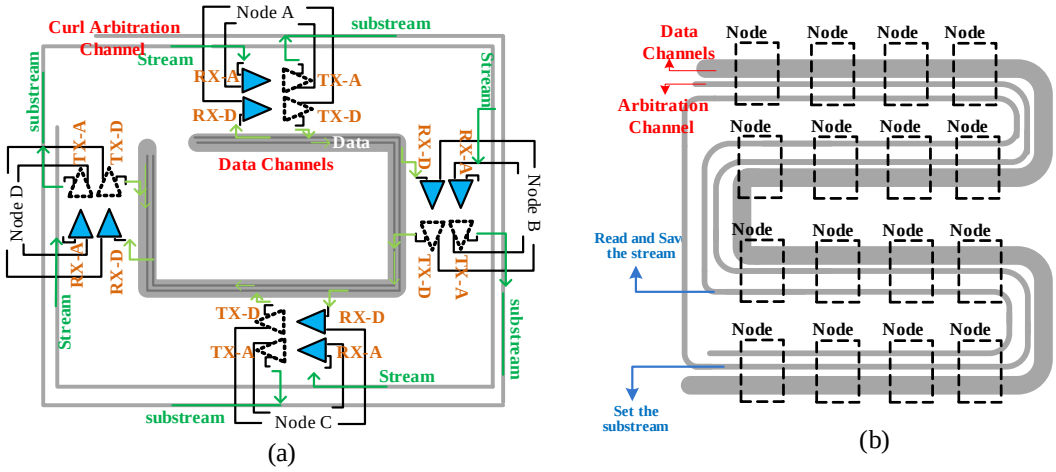


Figure 8.6: The curl transmission line

directional couplers. Since required signal power scales with the number of drops along a multi-cast link, such as is shown in Figure 8.5, a larger amount of power is required to transmit a signal on a multi-cast link as compared to required to transmit on a point-to-point link [88, 89]. Such effects are taken into consideration in our power estimation shown in Section 8.3.4.

8.3.2 Curled Transmission Line for Stream Circulation

A conventional RF-I transmission line is unidirectional and acyclic, i.e., the starting point and the end point of the transmission line are two different points. This prohibits the stream circulation in the arbitration channel. To enable the two-trip stream arbitration,

we propose a curled transmission line. Figure 8.6(a) shows the curled transmission line for the arbitration channel and the normal RF-I transmission line for the data channel. This curl starts from the stream origin at the outside loop and ends at the last RF node on the trip in the inner loop.

The outer loop of the arbitration channel is Trip 1 for transmitting only, while the inner loop is Trip 2 for receiving only. The transmitters to the arbitration channel (TX-A) of all the nodes are attached to the outer loop, while the receivers (RX-A) attached to the inner loop. There is also a frequency-tunable transceiver pair (TX-D and RX-D) at each node, which is attached to the data channels. Although we presented a rectangular style transmission line in Figure 8.6(a) for better illustration in the real physical design, all the transmission lines should go through each node, as shown in Figure 8.6(b). The reflection and discontinuity effect of sharp 90 degree turns in the curl transmission lines can be mitigated by careful designs for impedance matching. For example, [88] used 450 diagonal routing at each corner of the channel to eliminate sharp turns. As a result this did not impact the interconnect performance. Depending on different CMOS fabrication technology, rounded turns can also be implemented to better avoid the reflection issue caused by the sharp turns. In our evaluated system, which is a 1cm2 chip with 16 PE clusters (each cluster has one RF node), the total distance for 1 trip in the arbitration channel, or the longest distance in the data channel, is 5cm. The speed of light in silicon is 8ps/mm, thus each trip of the arbitration can be finished in 400ps. Our evaluated system has a working frequency of 2GHz. Therefore, each trip only takes one cycle, and any flit transfers on the data channel can reach its destination in one cycle.

At any particular cycle, the TX-As of the nodes are augmenting their substreams for the arbitration initiated during cycle x; the RX-As of the nodes are receiving the entire stream for the arbitration initiated during cycle x-1; the local stream parsing unit is parsing stream for the arbitration initiated during cycle x-2; the RX-Ds and TX-Ds of the winning sources of the arbitration initiated during cycle x-3 are using data channels to transfer data. In this way, stream arbitration can be initiated at every cycle.

8.3.3 Time Division Modulation Multicast for Stream Augmentation

We propose a time division modulation multicast (TDMM) approach in the arbitration channel to achieve the stream augmentation with priority rotation. The latency T for one stream trip can be divided into N slots, where N is the number of RF nodes, and the length of each slot is λ = T/N. Let d(v) denotes the RF node hops from the stream origin to node v. Let p(v) denotes the priority of node n in a particular arbitration, in which p=0 means highest priority. Then, the slot for a node to modulate its substream to the arbitration channel is (p+d) λ.

An example of TDMM is shown in Figure 8.7. In this example, there are 4 RF nodes. Node A is the stream origin where the curl starts. Assume the current highest priority is rotated to node C and the priority order is in the reverse of the stream travel direction. Then we have: node A: d=0, p=2; node B: d=1, p=1; node C: d=2, p=0; node D: d=3, p=3. Therefore, node A, B, C, and D will modulate their substreams at slot 2λ, 2λ, 2λ, and 6λ, respectively. These modulated substreams will finally form a stream that is in the order of the priority of their nodes when the stream makes the second pass to be read.

The proposed TDMM approach can support any arbitrary priority assigned to these nodes using the formula described above, provided all nodes have unique priorities. Here we adopt the gradual priority reduction scheme discussed in Section 8.2.1. Each node locally keeps a small counter to record its current priority p. Initially, each node v is assigned with a priority p(v) = d(v). After each arbitration, it increases its p(v) by 1. When p(v) reaches N, where N is the number of RF nodes, indicating that it is at the lowest priority,the node will reset p(v) to 0, which is the highest priority in the next arbitration.

In Trip 2, each node also receives one substream per slot. A node does not wait for all the substreams to begin parsing the stream (using the algorithm in Section 8.2.1). Instead, each time when a node receives a substream, it begins to process the substream by perform one iteration in the algorithm. The only difference is that the flow control

bits are buffered but not checked, and are used at the end to invalidate the winning source nodes with destinations that have a full buffer. In this way, we parallelize the stream receiving and stream parsing.

8.3.4 Power and Area

Table 8.1: Power Parameters of Point-to-Point RF Transceiver in 32nm Technology

	Power (mW)	Power Efficiency (pJ/b)	Active Area	Passive Area
TX Mixer	1		5um x 5um	0
TX PA	2.5		10um x 10um	50um x 50um
Total TX	3.5	0.44	$125um^2$	$2500um^2$
RX Mixer	0.5		10um x 10um	50um x 50um
RX PA	2		20um x 20um	0
Total RX	2.5	0.31	$500um^2$	$2500um^2$

Table 8.2: Power Parameters of Arbitration RF Transceiver in 32nm Technology

	Power (mW)	Power Efficiency (pJ/b)	Active Area	Passive Area
TX Mixer	1		5um x 5um	0
TX PA	5		15um x 15um	50um x 50um
Total TX	6	0.6	$250um^2$	$2500um^2$
RX Mixer	3.5		20um x 20um	50um x 50um
RX PA	2		20um x 20um	0
Total RX	5.5	0.55	$800um^2$	$2500um^2$

For power estimation, the predicted power parameters are different between the arbitration channel and data channels. Although data channels are broadcast links, the

arbitration strategy allows them to be treated as point-to-point links in any data communication cycle. On the other hand, due to large signal attenuation for the multicast link in the arbitration channel, increased power is needed to meet the signal-noise-ratio (SNR) requirement of desired bit-error-rate (BER, 10-12).

The power and area modeling values of point-to-point RF-Interconnect RF Transceivers used in this paper implemented with 32nm CMOS Technology are shown in Table 8.1. The TX and RX power consumptions are predicted (scaled) from our implementation of a multi-band RF-I at 90nm CMOS technology [88]. For the scaling from 90nm to 32nm CMOS process performance, it is assumed that the average power consumption per transceiver channel is expected to stay constant at about 6mW. The logic behind the assumption is that although RF circuits at higher carrier frequencies require more power, this additional power is compensated by the power saved at the lower carrier frequencies due to higher fT transistors available with scaling. In addition to increased number of channels, the modulation speed of each carrier would also increase, allowing a higher data rate per channel. As a result, the data rate per channel per wire is predicted as 8Gbps, which results in a power efficiency of 0.75pJ/b. A behavioral model simulation shows that 15GHz channel spacing is sufficient to carry 8Gb/s data with a low BER. Therefore it is projected that 12 carriers can be sent simultaneously on each wire (transmission line) given the 350GHz fT of 32nm technology, which indicates 96Gbps aggregate data rate on each wire. The active area and passive area are also predicted from our 90nm multi-band RF-I prototype.

Table 8.2 shows our power and area modeling of arbitration RF-Interconnect RF Transceivers in 32nm CMOS Technology. The power consumption is estimated by scaling our implementation of a multi-cast RF-I at 65nm CMOS technology in the similar fashion of the scaling of point-to-point RF transceivers. The power efficiency is predicted to be 1.15pJ/b. The number is higher than that of point-to-point RF transceivers mainly because of the larger channel loss of multi-cast data links. The data rate per channel per wire is predicted as 8Gbps. Therefore 12 carriers provide an aggregate bandwidth of 96Gbps on each wire for the arbitration channels. The active devices area is also less than

2x larger than the point-to-point link because of the higher gain required for arbitration links (larger devices implemented).

Due to the power overhead is basically the charging and discharging of the bias transistors gate capacitance, we configure the RF transmitter and receivers with simple logic gates which control their bias stage, so the RF transmitter and receivers can be turned off to save power when they are not in use. For example, a 50fF gate capacitance of the receiver bias transistor indicates 25fF energy consumption of turning it on and off, which is substantially smaller than demodulating one bit from the arbitration channel (¿1pJ/b). The speed of this power switching depends on the driving strength of the controlling logic gates. In 32nm technology, this speed is expected to be well below 0.05ns

8.4 Evaluation Methodology

8.4.1 Simulation Infrastructure

We evaluated stream arbitration using a composable heterogeneous accelerator-rich multiprocessor (CHARM) [24], where the on-chip accelerator building blocks (Custom Functional Units -CFUs) can be dynamically composed to virtual accelerators based on the applications requirement. While this architecture only features a small number of cores, the network and memory demand of the various accelerator components is substantially larger than that of a system featuring only conventional cores. We chose this architecture because we recognized that NoC design will not advance in isolation, and in order to evaluate a future NoC, we should evaluate it in combination with a system that would place a demand on the NoC representative of a HPC-aimed architecture, which is anticipated characterized as high raw throughput at low power of compute-intensive architectures in the near term future. As is shown in the Figure 8.8, our modeled system consisted of multiple cores, an accelerator block composer (ABC), a set of accelerator building blocks, memory controllers and L2 cache bank nodes. These nodes are arranged, in all

configurations, in an 8-by-8 grid, resulting in 64 nodes, and accelerators are uniformly distributed. Table 8.3 describes details of the system configuration. These details are consistent across all configurations.

Table 8.3: Evaluated system configuration

Parameter	Value
Processor	8x 2.0GHz Ultra-SPARC-III
Operating System	Solaris 10
Private L1 Cache	32-KB 4-way set-associative, 1-cycle access latency for each core
Shared L2 Cache	4-MB 8-way set-associative, 10-cycle access latency in 64 banks
Coherence Protocol	Shared banked uniformly distributed MOSI L2-Cache, private MSI L1-cache for cores, Distributed directory.
Memory	4 Memory controllers, 450 cycle latency, 30GBPS bandwidth each [120 GBPS aggregate]

We have extended the Simics [19] and GEMS [20] simulation platform to model a RF-I network, along with the stream arbitration scheme. As a point of comparison, we have also implemented a recently proposed token-based arbitration for performing arbitration over optical interconnects [80], which we will refer to as token arbitration. Our implementation of token arbitration was adapted to use a RF-I network so as to make a fair comparison. Brief details of the operations of token arbitration are described in Section 4. Timing, area, and power consumption associated with the RF-I network was measured using the methodology discussed in Section 3, and was incorporated into our cycle accurate simulation platform. Timing, area, and power consumption associated with the digital circuitry required to perform arbitration was obtained by compiling code implementing our arbitration scheme into RTL using the AutoPilot [21] behavioral

synthesis tool, which was then synthesized using the Synopsys Design Compiler [22]. For network communication along traditional wires, such as that between multiple nodes and shared RF-I transmission points, we used Orion 2.0 [29] to estimate power consumption in 32nm technology. Accelerators and accelerator arbitration mechanisms used in this work were modeled after those presented in [8]. Accelerators were included due both to the expected increase in their use in future processors, and as an effective way at stressing the NoC in a way that cores cannot. The accelerator design methodology and accelerator arbitration mechanism we adopted for this work supports hardware load-balancing.

To more effectively examine the performance of stream arbitration, and how it performs relative to token arbitration, we examined a number of different system configurations and RF-I configurations. We modeled cases in which nodes on our network were bunched into clusters of varying sizes, with RF-I serving as the only communication mechanism between clusters. We also scaled the capability of our examined RF-I network in terms of number of channels and bandwidth per channel. A clustered architecture means several routers share one RF transceiver. Only the communications between different clusters will go through the RF-I, with traffic between nodes in the same cluster instead using a fully connected traditional network. We considered the impact on latency and power of different cluster sizes of 4, 2, and 1, and found the effect is minor due to marginal reduction in the amount of bandwidth dedicated to servicing data channels when reducing the cluster size. The another reason is that, from the perspective of each individual cluster, the overwhelming majority of network traffic was inter-cluster traffic, irrespective of the number of nodes in each cluster. Therefore we chose a cluster size of 4 as our baseline configuration in this paper if not special specified.

8.4.2 Benchmarks

To evaluate our work, we examined a number of accelerator using benchmarks. These benchmarks consisted of those evaluated in the CHARM design [24], along with two PARSEC benchmarks [39], Fluid Animate and Black Scholes, that were amenable to

being mapped to accelerators. The examined region was restricted to the program kernel, which was either entirely, or nearly entirely, covered by hardware acceleration.

Since our selected accelerator arbitration scheme features hardware load balancing among accelerators, there is not a substantive difference in system load when adding software threads. For this reason, our benchmarks were primarily executed sequentially, with the accelerated regions relying on load-balancing hardware to achieve concurrency.

8.4.3 Reference Scheme

Token arbitration, presented in [80], is a multiple-writer single-reader arbitration mechanism designed for use in optical NoCs. Token arbitration makes an effort toward efficiently solving the problem of arbitrating a shared communication channel to assure that multiple writers on a single channel do not interfere with one another. In order to assure that only a single writer is writing to a given destination, the destination node transmits a send token around a ring in the NoC. A node interested in sending will halt the progress of this token by reading it, and hold it until it finishes sending, at which point it will reemit the token. This requires that there is a single communication channel for each potential destination node.

While token arbitration makes guarantees of liveness and fairness, restricting each communication channel to service only a single destination potentially results in poor channel utilization and difficulty scaling with number of nodes, since the number of channels must also scale accordingly. We chose token arbitration as our point of comparison because it is recent work, and it targets the specific problem of sharing communication channels on emerging interconnect technologies.

While static shortcuts [81], as mentioned in Section 1, are also designed to utilize emerging interconnect technologies, we chose not to compare against this design point due to the fundamentally different nature of the problem static shortcuts addressed. Static shortcuts also rely on an underlying mesh network as a fallback option if a shortcut isnt present, which our approach does not. As a result, it would not be possible to construct

a fair comparison between static shortcuts and stream arbitration.

8.5 Results and Discussions

8.5.1 Performance as Bandwidth Scales

Figure 8.9 shows a comparison of average network flit latency between stream arbitration and token arbitration, for various network configurations. Each system consists of 16 clusters of 4 nodes each, laid out in a grid, with a single RF-I access point for each cluster. Aggregate bandwidth is calculated by the number of channels multiplied by the bandwidth of each channel. Token arbitration requires a single channel for each possible destination, and thus aggregate bandwidth is adjusted by scaling the bandwidth of each channel from 2 to 16 bytes per cycle. Stream arbitration decouples the number of nodes from the number of channels, so instead aggregate bandwidth is adjusted by fixing the bandwidth of a single channel at 16 bytes per cycle, and scaling the number of channels from 2 to 16. As expected, the performance of both systems becomes near equal as both the bandwidth per channel and number of channels becomes 16 for both arbitration schemes. Our experiment also shows this clearly.

As is shown, there is an advantage of having few channels of high bandwidth and allow them to be flexibly used by any communication pairs at runtime, over having many channels of uniformly low bandwidth. This is intuitively sound, as each cycle only a fraction of nodes will be communicating with one another. Having a large number of channels is only an advantage when the traffic pattern is highly uniform, at which point all channels can be utilized continually. Since each destination can only have a single channel associated with it at a time, any non-uniformity in the traffic pattern, even if just instantaneously, results in favoring a few high bandwidth channels. While both systems were able to reduce average flit transmission latency by upwards of 60-70% for most benchmarks in systems that featured an abundance of RF-I resources, we found that stream arbitration was able to approach this figure much more quickly when RF-I

resources were reduced.

For most of our examined benchmarks, the benefit of adding channels diminished quickly, highlighting that only a small handful of nodes at any given instant are responsible for the majority of the NoC traffic. While cache activity was approximately balanced, clusters in the corners of the network which featured memory controllers saw additional traffic in the form of memory requests, and clusters featuring highly utilized accelerators saw additional traffic in the form of cache responses.

Figure 8.10 shows the impact of this latency reduction on benchmark runtime. NoC latency reduction doesnt correlate directly with performance primarily due to all of the other factors contributing to performance that are not NoC related. Chiefly among these, for both accelerator centric architectures examined in this work and conventional CMP designs, is the contribution of memory latency. Reducing NoC latency primarily contributes to performance by reducing the latency associated with coherence between the private and shared cache layers. While there is a correlation between reduction of NoC latency and improvement of overall performance, the 60-70% NoC latency reduction we observed contributed only 15-25% runtime reduction in most cases. Some benchmarks were outliers, such as Deblur, which is a memory bound benchmark that featured a working set that fit in shared cache. In these cases, we observed a much more pronounced relationship between NoC flit latency reduction and performance improvement.

8.5.2 Energy Consumption

Energy in emerging NoC designs, such as RF-I, comes primarily from two sources. First, modulation and demodulation of the signal, which occurs when a one bit of data is sent on the network or received from the network respectively. Second is the preparation of the communication medium for carrying data. For RF-I this involves supplying receivers with energy to listen to the attached channel, which must be done perpetually on any channel on which data may arrive, whether the given channel is being used to carry data or not. These two sources are analogous to dynamic and static power on traditional

electric circuits. Results shown in Figure 8.11 were gained from the experiments for which performance results were shown in Section 5.1. Energy consumption for CPUs, caches, and accelerators is not shown in order to highlight the contrast between the two arbitration algorithms.

The RF transmitter and receivers can be turned off to save power when there is no RF signal modulation and demodulation as discussed in Section 3.4. Therefore there is power saving when there is no data being sent to/from a RX-D/TX-D through the data channels, or there is no arbitration in the arbitration channel. Token arbitration on the other hand continually circulates the arbitration tokens, even when the NoC is otherwise unused. The power required to do this scales linearly with the number of potential destination nodes, but is generally small relative to the power required to actually send data. A considerable portion of the energy difference is attributed to the difference in arbitration success rate. Stream arbitration features a very high arbitration success rate in general due to the short utilization period of a given channel. In comparison, token has a relatively poor arbitration success rate due to sending nodes holding the token for long durations when link bandwidth is small.

As shown in Figure 8.11, the difference in energy consumption between stream arbitration and token arbitration is mostly related to the difference in performance provided at a given design point. But Token performs better than Stream at points that provide similar or identical performance, the reason is that Stream has more modulation power than Token, which needs modulate 6 bits each time compared with only 1 bit for token.

8.5.3 Bandwidth Allocation for Channels

To further examine the point discussed in the previous section, we conducted an experiment with a fixed aggregate bandwidth, and adjusted the number of channels and channel bandwidth to achieve that aggregate bandwidth. Figure 8.12(a) shows the impact on average flit latency of adjusting the division of bandwidth into multiple channels, with a fixed aggregate RF-I bandwidth budget of 108 bytes per cycle. The optimal configura-

tion varies by benchmark. While Section 5.1 showed that adding additional bandwidth was never observed to result in a degradation of performance, it is clear that there is a compromise to be found between high bandwidth channels and additional channels. Figure 8.12(b) shows the corresponding impact on execution time for each design point.

While in this work we focused on static partitioning of bandwidth into multiple channels, this figure shows the potential for dynamically partitioning this bandwidth as well.

8.5.4 Data Channel Utilization

Figure 8.13 shows the percent of total RF-I traffic that occupies each data channel using stream arbitration for several selected benchmarks, for a system with 64 nodes and the clusters of size 2, which is allocated 6 RF data channels and 16 bytes per cycle bandwidth per channel. As described in Section 2, stream arbitration allocates data channels to a communicating source-destination pair ordered by winning arbitration. Thus, if any data at all is being sent over RF-I, it is guaranteed that data channel 0 is in use, followed by channel 1 if two channels are in use, and so on. This figure shows that even a single channel can accommodate between 27% and 52% of the total bandwidth demand. The utilization of channels drops steadily. While data could be shown for systems with larger numbers of channels, the utilization for later channels becomes extremely low, indicating that the physical RF-I medium is being wasted. This clearly indicates that dynamic channel allocation is critical to effective resource utilization of emerging networks, such as RF-I, as static allocation results in a large amount of waste.

8.6 Scalability

8.6.1 Hierarchical Stream Arbitration

As the chip size scales up or the number of RF nodes increases, the length of the transmission line increases and one stream trip may need P cycles, where P is larger than 1, to finish. In this case, the latency to complete arbitration would increase to 2P+1. As the

number of RF nodes N increases, the amount of data sent to perform a single arbitration increases in O(NlogN). If we keep using a single curl to perform flat stream arbitration for all RF nodes, we will encounter serious challenges in terms of performance, power, and area:

- **Performance:** Each arbitration needs 2P+1 cycles to finish, therefore, it will take a much longer time for a network message to claim a data channel successfully when P scales up to a large number. This results in a significant increase in average network latency. Although we can still use pipelining to initiate stream arbitration every cycle in order to guarantee throughput, but for applications with message dependencies, average network latency has a significant impact on system performance. Under this circumstance, the long 2P+1 arbitration latency limits the scalability of the base stream design.

- **Power:** The length of the arbitration stream increases in O(NlogN), so that the arbitration channel power also increases in O(NlogN). If N is a relatively small number, then compared to the data transfer power in the data channels, the arbitration power is negligible. This is how stream arbitration can win over token arbitration in terms of power. However, as N scales up to a large number, then this arbitration overhead becomes non-trivial compared to data transfer power.

- **Area:** As the number of data channels increases as N scales up to maintain network performance, the total number of channels that the RF transceiver on the data channels should support also increases linearly with N. For this reason, the area of the communication substrate increases quadratically with N.

In the large-scale CMPs, it is not the case anymore that all the network elements are communicating with all other elements uniformly. The state-of-the-art non-uniform cache architecture (NUCA) management schemes, such as the RNUCA [82] and page-recoloring scheme [90], intend to place cache blocks near to its most frequently requesters by smart initial placement, dynamic migration, and replication. Moreover, cache partitioning schemes [65, 91, 92] can be used so that a cluster of cores are only going to access

a locally allocated cache partition. Communication between these partitions is only required when there are coherence invalidations and fills. Therefore, it is expected that in large-scale CMPs, the local communication in a core-cluster dominates the overall communication. Under this circumstance, a flat stream arbitration scheme that uses a single curl to go across all nodes is wasting resources and unnecessarily limiting performance. Here we propose a hierarchical stream arbitration scheme to make use of the communication pattern expected to be found in future large-scale CMPs. In hierarchical stream we have a local transmission line (TL) for each core-cluster, and a global TL to connect these local TLs. Each TL consists of a curled arbitration channel and a set of data channels, as shown in Figure 8.14. The stream arbitration is performed independently in each level of the hierarchy.

The local TL is connected to the global TL through a relay node which consists of two RF interfaces and two buffers, as shown in Fig 14. Each RF interface is similar to the RF nodes described in Section 3.2. One RF interface consists of a pair of transceiver (TX-A and RX-A) connected to the local curl and a pair of transceiver (TX-D and RX-D) connected to local data channels. Another RF interface also has two sets of transceivers connected to the global curl and global data channels. Buffers are used to temporarily buffer the network messages which have already arrived at the relay node but are still waiting for arbitration success to move forwardeither from local TL to the global TL, or vice versa. Each of the buffers will attach its flow control signal to the substream in the corresponding arbitration channel in order to indicate the fullness of this buffer, as a normal RF node does in Section 2.

The proposed hierarchical stream arbitration RF-I NoC is deadlock-free as long as we only allow minimal routing. Figure 8.15 shows the topology graph (TG) and the corresponding channel dependency graph (CDG) of an example hierarchical stream arbitration RF-I NoC (the definitions of both TG and CDG are detailed in [93]). In this example there are four local transmission lines (TL) and 1 global TL, each local TL contains 4 RF nodes (one of them is the relay node). Since the NoC is built upon RF medium, in the TG, when we say that there is a channel (the unidirectional arrows in

Figure 8.15(a)) from RF node A to RF node B, we mean that a packet can go directly from RF node A to RF node B through the RF medium. Due to the space limitation, in the CDG (Figure 8.15(b)), we only details the channel dependencies in the global TL and also one of the four local TLs, since the CDGs of the four local TL are identical. The purple solid arrows in Figure 8.15(b) denote the channel dependencies among the channels in TG, and the red dashed arrows denote the channel dependencies between the channels in the TG and the network injection queue (Si) and ejection queue (Ti). As it can be seen in Figure 8.15(b), the CDG of the network is acyclic. According to the theory in [94], the routing of the proposed hierarchical stream arbitration RF-I NoC is deadlock-free.

The example shown in Fig 14 has 8x8 RF nodes (actually 256 network nodes and each 4 network nodes share one RF transceiver). We let 16 RF nodes share one local TL, so that we can have enough slack to enable one stream trip to finish in one cycle, as discussed in Section 3.2. The total distance for one stream trip in global TL is 5cm with this 8x8 RF nodes topology, and so can be finished in one cycle also in this topology. This two-level hierarchical design contains 4 local TLs and 1 global TL. Since there is also a relay node in the local TL, actually, there are 17 RF nodes competing for the data channel resources of the local TL. In the global TL, there are 4 RF nodes (the 4 relay) competing for the data channels resources of global TL. In the example shown in Fig 14, if node A wants to send a message to node B, it only needs to perform arbitration within its local TL, where each arbitration takes 3 cycles (one for the first stream trip, one for the second stream trip, and one for the stream parsing). Then the message can be sent to B using the granted data channel from this local arbitration. However, if node A wants to send a message to node C, it first attempts local stream arbitration in Local TL 1, and then uses the granted local data channel to send the message to Relay Node 1. In Relay Node 1, the message is buffered and waiting for the global arbitration to allocate it a data channel in Global TL. Upon success, the message is sent by Relay Node 1 to Relay Node 3 using the granted global data channel. Then in Relay Node 3, the message is buffered again and waits for the success of arbitration in Local TL 3, at which point

it is sent to node C by Relay Node 3. The entire process takes at least 9 cycles.

8.6.2 Trace-Driven Evaulation Methodology

To verify the performance of the hierarchical architecture, we scaled the NoC size (in terms of number of routers) to 16x16, 24x24 and 32x32. In these topologies, each 2x2 routers share 1 RF node. Then the three evaluated NoC designs have 8x8, 12x12, and 16x16 RF nodes, respectively.

- For hierarchical stream arbitration, every 4x4 RF nodes share 1 local TL, so that in each local TL there are 17 RF nodes in total (the additional one node is the relay node). One stream trip in curl of the local TL takes one cycle. The hierarchical architecture of 8x8 RF nodes is already shown in Figure 8.14, where the global TL has 4 RF relay nodes. The hierarchical architecture of 12x12 and 16x16 RF nodes are shown in Figure 8.16, where the RF relay nodes in their global TL are 9 and 16, respectively. One stream trip in the curl of the global TL for these three designs takes 1, 2, and 4 cycles, respectively (depends on the length of TL).

- For flat stream arbitration, there will be a single TL across all of the RF nodes for each NoC designs. With the assumption and discussion in Section 3.2, one stream trip in the curl of the single TL for the three NoC designs takes 3, 7, and 11 cycles, respectively (depends on the length of the TL).

We assign two 16B data channels for each local TL for the hierarchical stream arbitration. To make a fair comparison, we make the number of data channels in the TL of flat stream arbitration as the total number of local TL data channels of the corresponding hierarchical design.

As we scale the number of components on chip to such a large level, full-system simulation becomes intractable due to its extremely long runtime (in order of weeks to months). Therefore, in this section, we used a trace-driven cycle-accurate network simulation method to evaluate performance and power consumption. We extended Garnet [95]

to support the proposed hierarchical stream arbitration scheme. Moreover, as a means of exploring the interconnect demand of future applications, we made use of the probabilistic trace methodology developed in [81] to represent five communication patterns for multi-threaded applications uniform, uni-dataflow, bi-dataflow, 1hotspot and 2hotspot. To mimic the local-communication-dominated commutation patterns expected in large scale CMPs, our probabilistic traces have 60% of the communication as local traffic (inside the local TL). The remaining traffic follows the specific communication pattern as the trace name indicated: 1) uniformthe routers are equally likely to communicate with all other routers in different curls; 2) 1hotspot/2hotspotthere is one router in one/two TLs sending/receiving a disproportionate amount of traffic. 3) uni-dataflow/bi-dataflowdataflow pattern simulates pipelined communication flow such as medical imaging decomposition or a cryptographic algorithm, routers are biased to communicate with routers in groups that neighbor them on either one side (unidirectional dataflow) or two sides (bidirectional dataflow). The message sizes were either 8 bytes (a cache block request or control signal) or 64 bytes (a cache block response). Each probabilistic trace is executed on Garnet for 1 million network cycles.

8.6.3 Results

Figure 8.17 shows the average network latency reduction through hierarchical stream arbitration (denoted as HStream) compared to flat stream arbitration (denoted as FStream). By serving the local data channel requests in the local TL with much less latency, HStream arbitration can reduce the average network latency by 28%-55% (40% on average) compared to FStream. The latency reduction in arbitration actually is similar for all of the five patterns (they all have the similar percentage of local traffic), however, there is still observable differences between the overall latency reduction of them. The reason for the considerable difference in performance observed is less to do with the arbitration latency, or even the latency of actual data transmission, but of the increased success of arbitration in the case of HStream as compared to FStream. The reason for this is intuitive: As the network increases, HStream maintains a constant number of competing

nodes for the messages within the local TL. Communication occurring in disjoint regions of the NoC does not compete for a shared resource, where as in FStream it does. Even though in FStream this shared resource is of greater capacity, being the sum of the local TL and the global TL in terms of bandwidth, this doesn't make up for the large increase in competing nodes in a network that is not segregated into regions.. Although in HStream there are two additional buffering steps in the relay nodes, the experimental results show that most of these long distance messages are granted data channels in the next arbitration immediately following the time it enters the relay node. We observe a reduction in benefit of HStream as compared to FStream in the hot-spot experiments. The primary source of this reduced benefit is attributable to availability of buffer space in the node relay in transition point from the local TL containing a hot spot to the global TL. FStream does not exhibit this bottleneck, and thus performs similarly as it performed in other experiments.

Figure 8.18 shows the power comparison results of HStream and FStream, which is broken down into arbitration power and data transfer power. The data transfer power also includes the power between the network interface and the RF node. The reduced data transfer required to perform arbitration, as described in Section 6.1, allows HStream to reduce energy required by arbitration by 40% - 70% compared to FStream. The larger the network size is, the more significant of this arbitration reduction to the overall power reduction is. However, in HStrean, all of the communications that go across two local TLs will have does the modulation/demodulations due to the need to buffer during transitions between the local TL and the global TL. As mentioned, these global communication only accounts for a small percentage of the overall communication in the large-scale CMPs, thus HStream only incurs a 3%-16% additional data transfer power. The overall power of HStream and FStream is similar, within 5%.

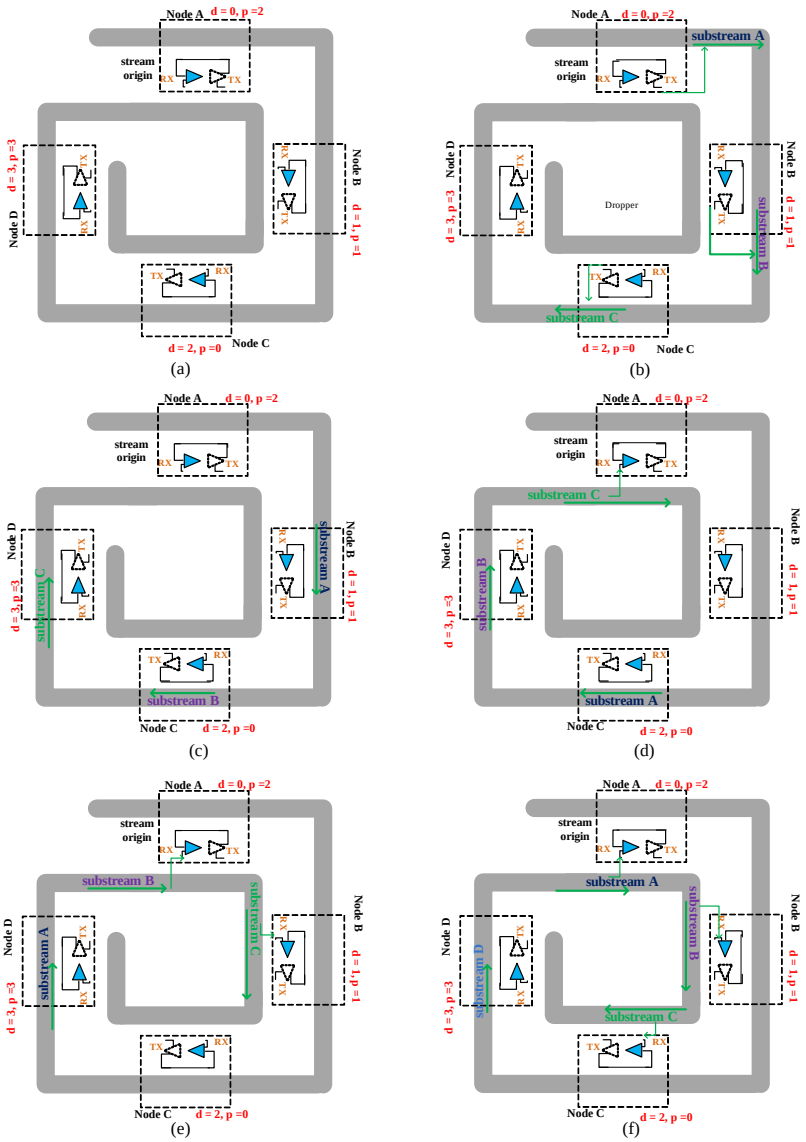


Figure 8.7: An example of TDM multicast for stream augmentation with priority rotation: (a) t=0 and t=λ: no substream is modulated. (b) t=2λ: node C, B, A modulate their substreams simultaneously. (c) t=3λ: substream C, B, A achieves node D, C, B, respectively. (d) t=4λ: substream C, B, A achieves node A (inner), D, C, respectively. Substream C is received by node A. (e) t=5λ: substream C, B, A achieves node B (inner), A (inner), D, respectively. Substream C and B is received by node B and A, respectively. (f) t=6λ: node D modulates its substream. Substream C, B, A achieves node C (inner), B (inner), A (inner), respectively.

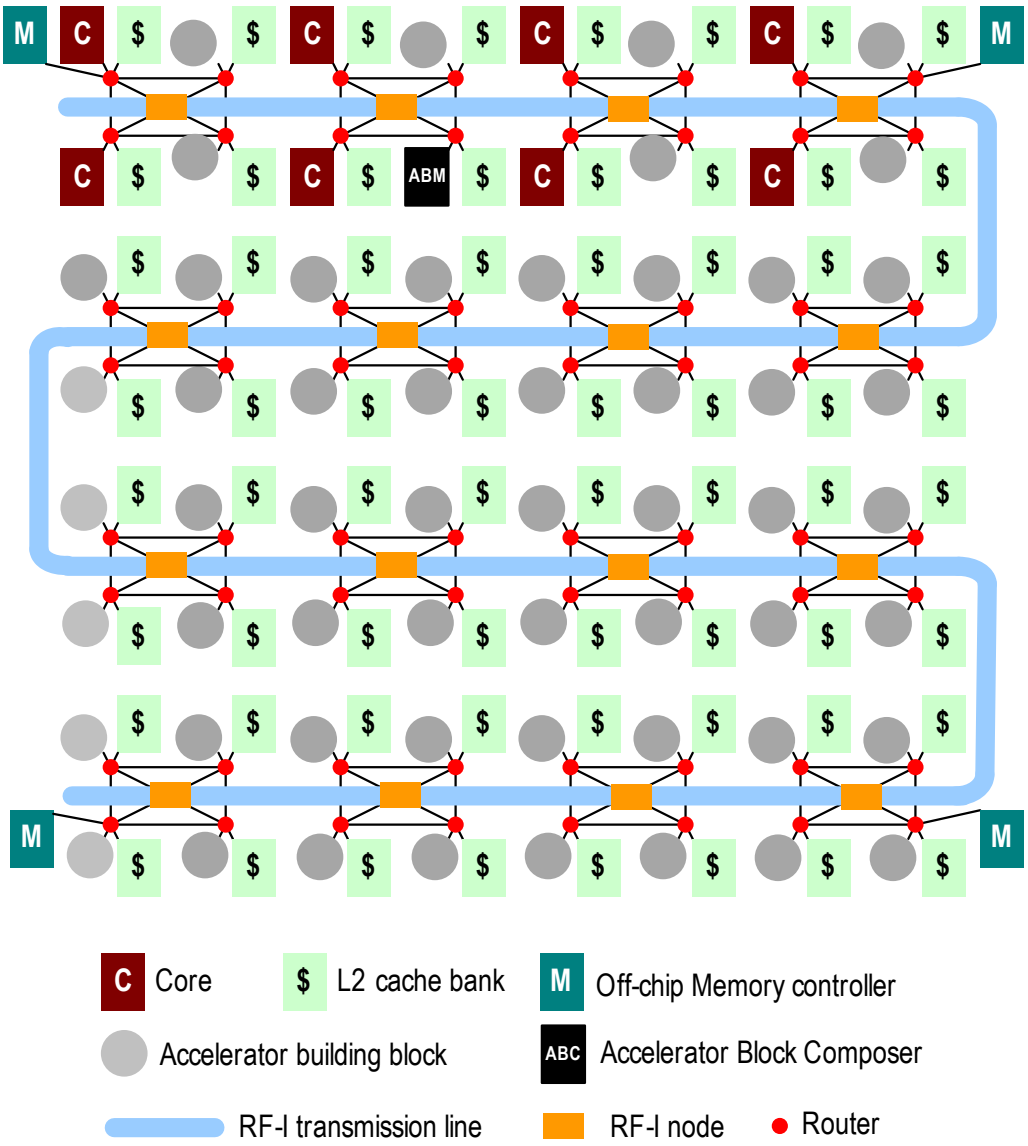


Figure 8.8: Overall diagram of the evaluated CHARM architecture with RFI overlaid NoC

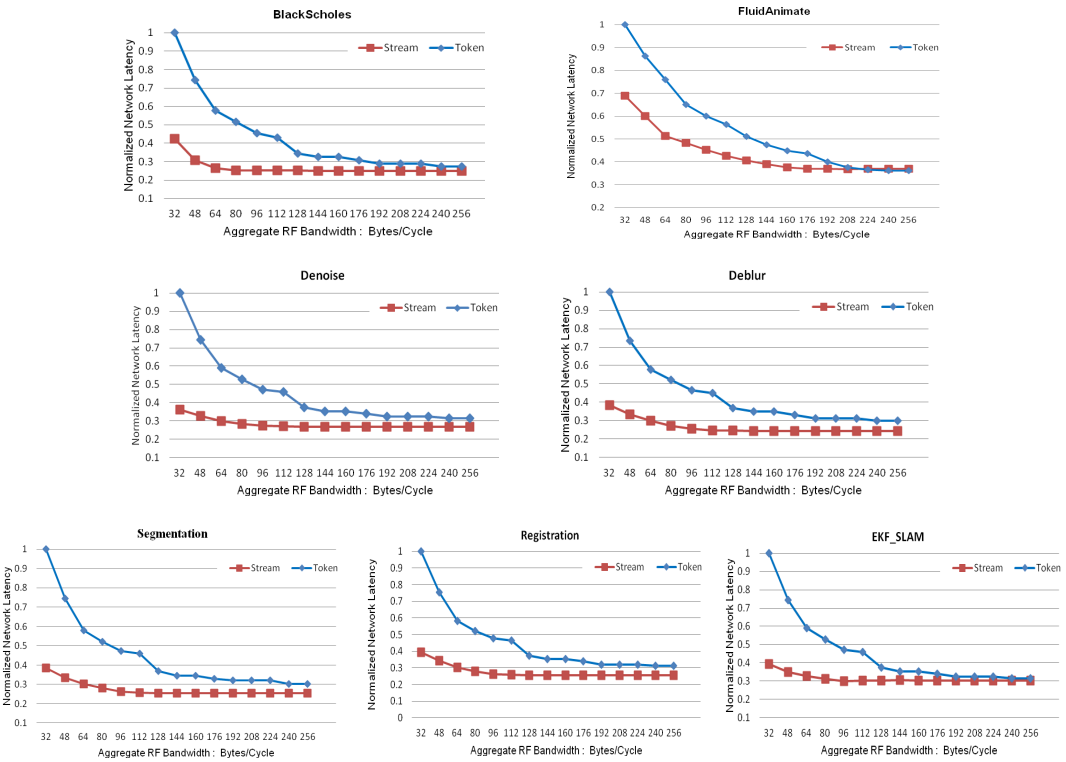


Figure 8.9: Comparison results of average network flit latency at various aggregate bandwidths (normalized to token arbitration with 32 Byte/Cycle aggregate bandwidth)

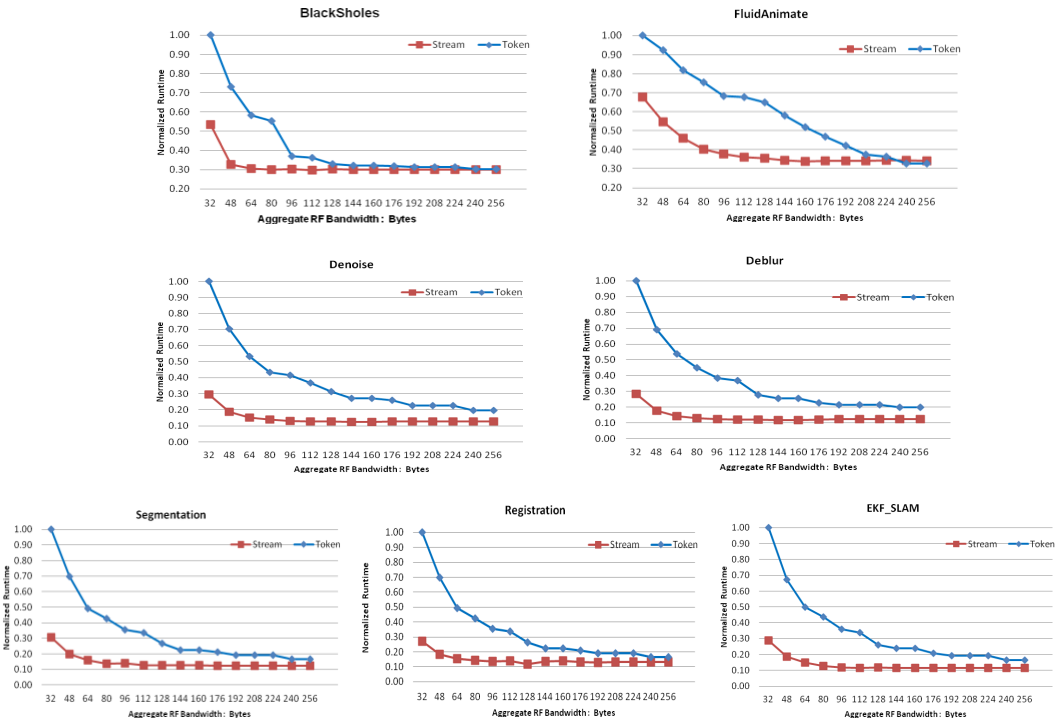


Figure 8.10: Comparison results of application runtime at various aggregate bandwidths (normalized to token arbitration with 32 Byte/Cycle aggregate bandwidth)

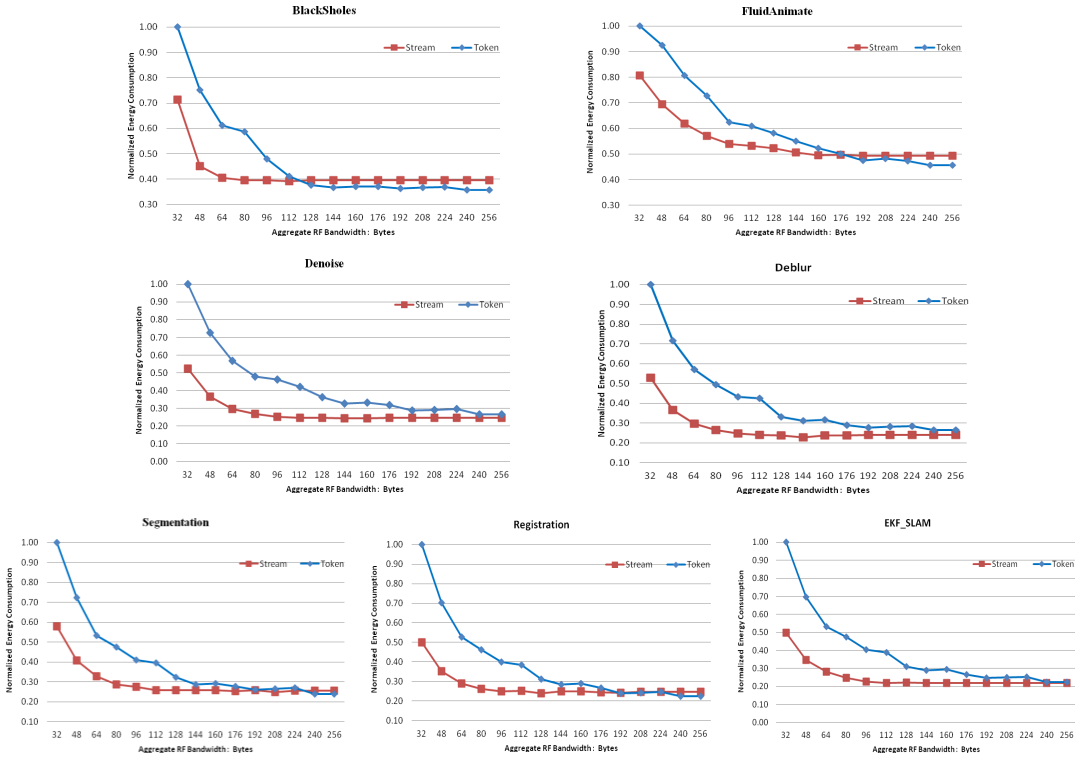


Figure 8.11: Comparison results of application Network power consumption at various aggregate bandwidths (normalized to token arbitration with 32 Byte/Cycle aggregate bandwidth)

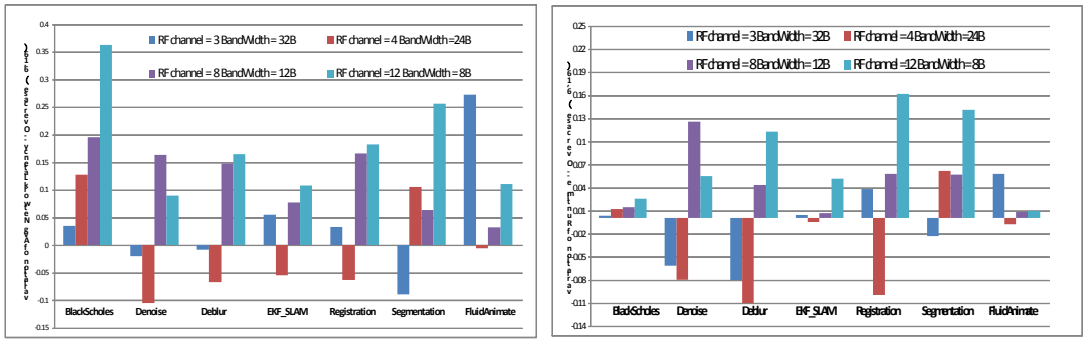


Figure 8.12: Impact of channel bandwidth allocation: (a) Average network flit latency. (b) Application runtime (The results is the variation over the case of RF channel =6, Bandwidth = 16 Byte/Cycle)

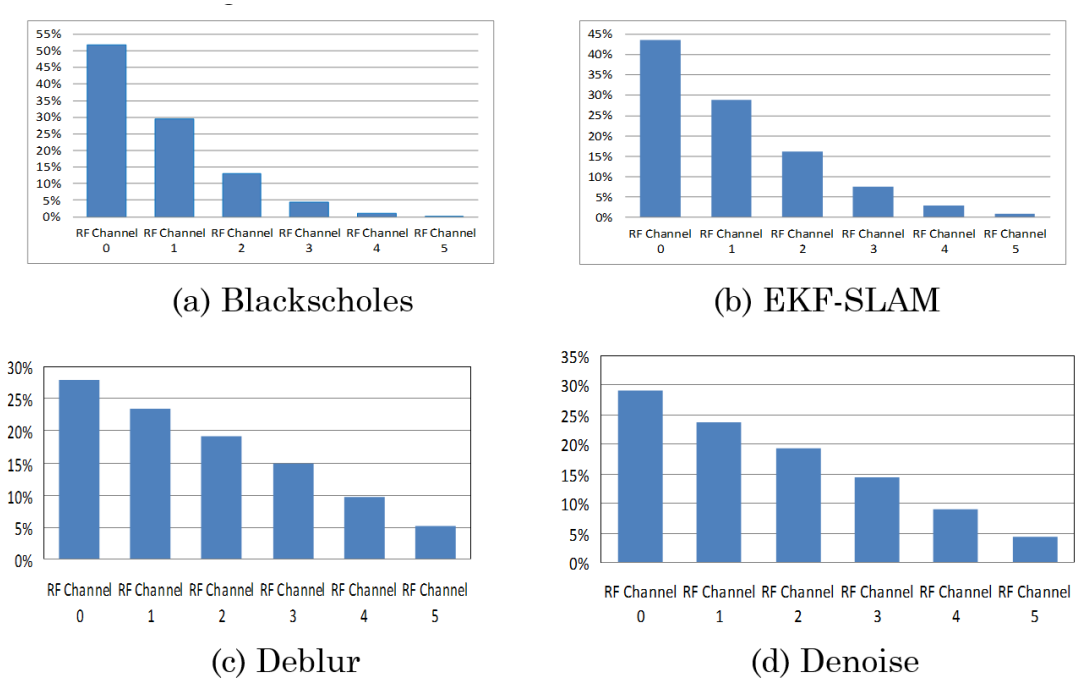


Figure 8.13: RFI data channel utilization

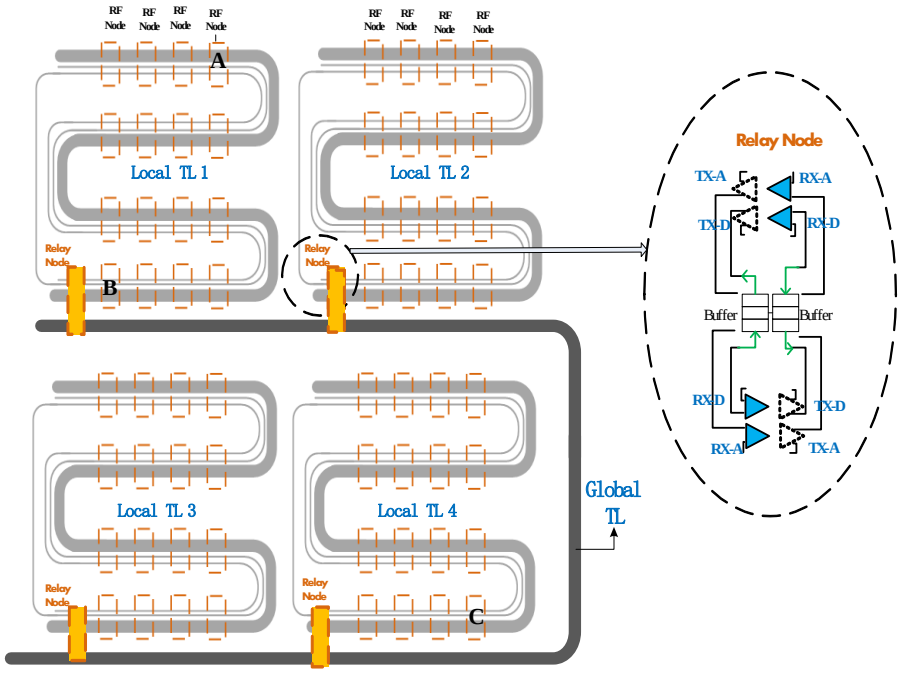


Figure 8.14: Hierarchical stream arbitration

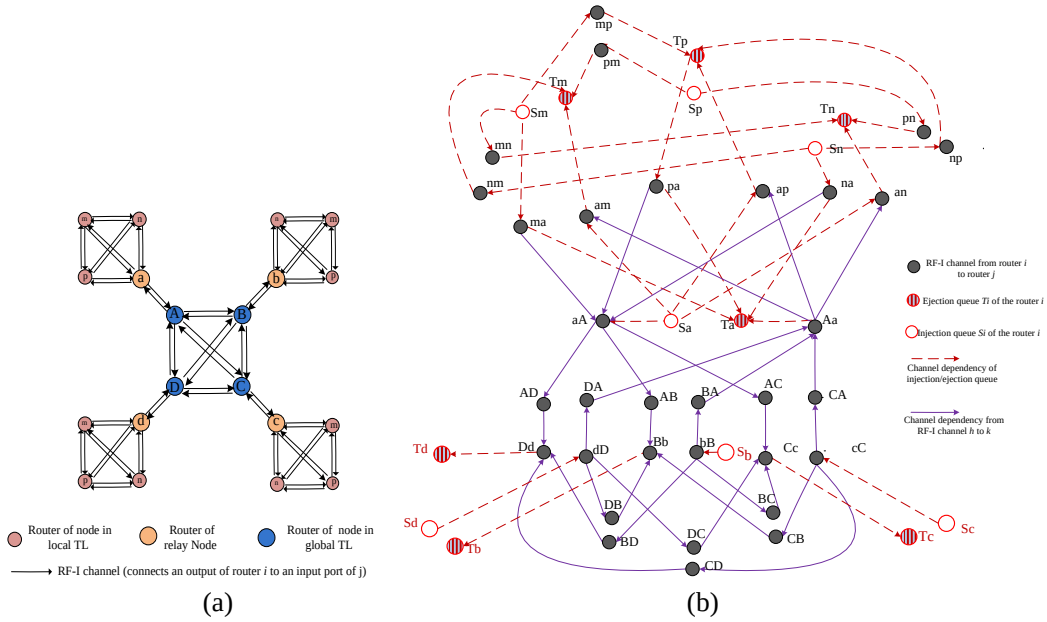


Figure 8.15: Hierarchical NoC topology characterization graph of (a) TG, (b) CDG

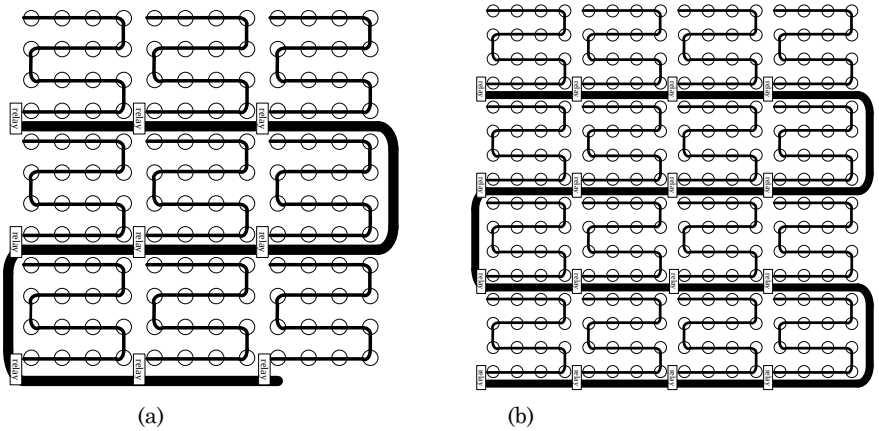


Figure 8.16: Hierarchical architecture of (a) 12x12 RF nodes for a 24x24 routers NoC; (b) 16x16 RF nodes for a 32x32 router NoC. Each 2x2 routers share 1 RF node

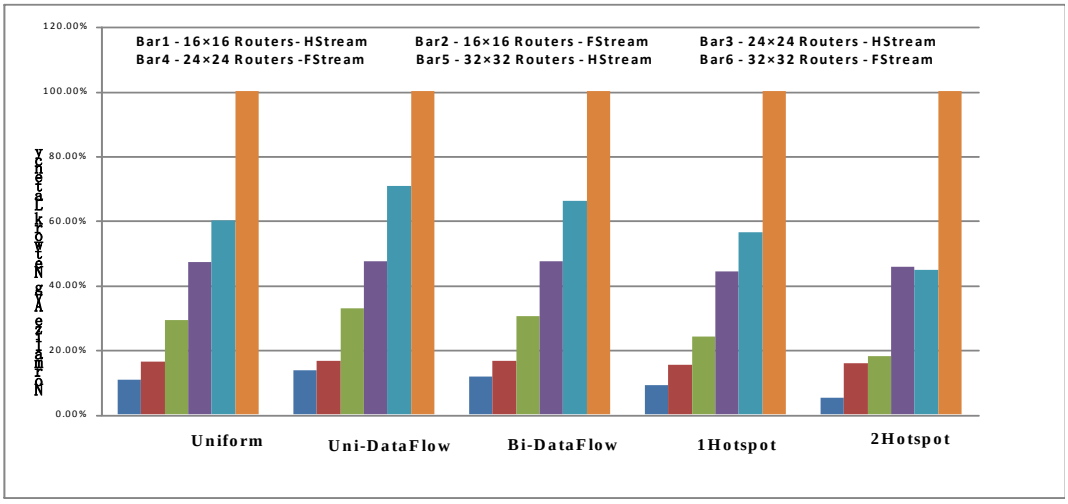


Figure 8.17: Performance (average network flit latency) gain through hierarchical stream arbitration (normalized to Flat Stream of 32*32 topology for each application)

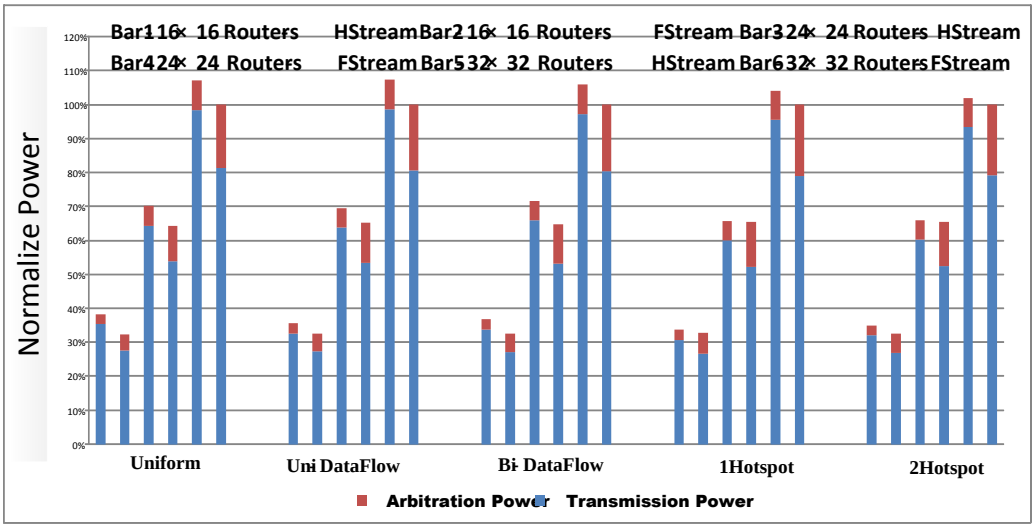


Figure 8.18: Power savings through hierarchical stream arbitration (normalized to Flat Stream of 32*32 topology for each application)

CHAPTER 9

Core Design After Acceleration

Introducing accelerators to a platform bifurcates code that runs on the system into two components: 1) code that runs on the accelerator, and 2) code that cannot run on an accelerator. Accelerators typically offer such an advantage in terms of both performance and energy efficiency as to leave no reason to use a conventional processing core to perform a task that can otherwise be performed by an accelerator. Because of this, the code that ends up running on a conventional core in an accelerator featuring system is considerably different in composition as compared to the code that runs on a conventional core in a system that lacks accelerators.

Accelerators are most appropriate for code that is highly structured, small, and compute intensive. While these code kernels generally constitute only a tiny portion of the total program size, they are greatly overrepresented in the total program execution time. For this reason, much of the research that has gone into developing high performance conventional cores has gone into introducing microarchitectural enhancements that target these compute intensive kernels. Because introducing accelerators to a system eliminates the cores responsibility to execute these highly structured code regions, many components of a conventional core are no longer utilized in the same way, and no longer contribute to a cores performance in the same way or to the same extent.

This chapter re-examines a number of common components that are critical to the performance of aggressive out-of-order processors in the context of a system with accelerators. The reasoning behind this is that the benefits that these components lend to a out-of-order processor are primarily restricted to enhancing the execution of code regions that are no longer the responsibility of the core, and are now instead executed on the

accelerator. With these code regions now executed on the accelerator, the benefit that these structures lend to a core may no longer justify the cost of including them.

9.1 Platform Model

There are a large number of proposals for programmable accelerators [24, 50, 54]. While proposals differ slightly in the types of program regions they can cover, most accelerator proposals generally target loops surrounding small compute kernels that iterate over a volume of data.

The general work flow involved in using these programmable accelerators is also similar across a variety of designs. During compilation, a schedule of an identified compute kernel is generated and stored as a portion of the program. At runtime, a configuration describing the computation to be performed is loaded into the accelerator. After the accelerator is configured, it is invoked by a signal from a controlling core. After a period of time, the controlling core receives a signal back indicating that the requested work has been completed.

While different programmable accelerator proposals differ widely in architectural characteristics, advantages, and drawbacks, this similarity in program coverage makes it possible to discuss impact on program structure in an accelerator-agnostic fashion. For this reason, this work will consider a model accelerator that is meant to represent the constraints of more complex accelerator proposals, while simplifying the discussion about program coverage. This accelerator model is not meant to represent a optimal accelerator design, but is architecturally simple enough to drive a conversation about accelerator utility without becoming mired in a conversation about accelerator design. Accelerator design has already been covered extensively in previous chapters, and is not the focus of this work.

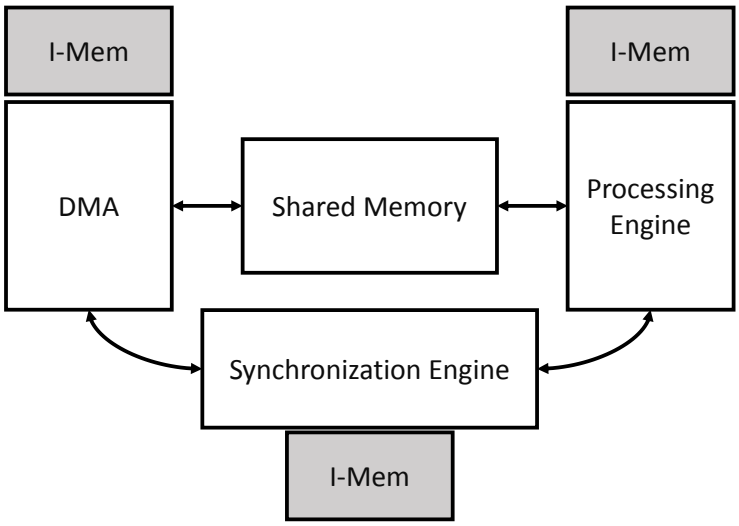


Figure 9.1: Architectural overview of the programmable accelerator model.

9.1.1 Accelerator Architecture

The accelerator model used was selected to capture the capabilities of many recent accelerator proposals without necessitating a detailed architectural discussion about the internals of the accelerator. As such, it is architecturally simple enough to be intuitive in its design. The selected accelerator is not intended to be the most efficient possible accelerator, nor is this work arguing that an accelerator of this type should be included in an actual architecture, but instead only intended to simplify the discussion of how the introduction of this accelerator influences core design.

The accelerator model consists of three compute components and a shared internal memory, each of which can be individually programmed by a controlling core, and communicates with the controlling core by a hardware managed message queue. The three components, illustrated in Figure 9.1, are 1) the DMA engine (DMA-E), 2) the synchronization engine (SE), and 3) the processing engine (PE). The SE sends control signals to both the DMA-E and PE instructing them on when to start working. The DMA-E is responsible for moving memory in and out of the accelerator, usually as bulk transfers. The PE is responsible for interacting with memory stored in the shared memory region, and does not interact directly with system memory. The SE interacts with the

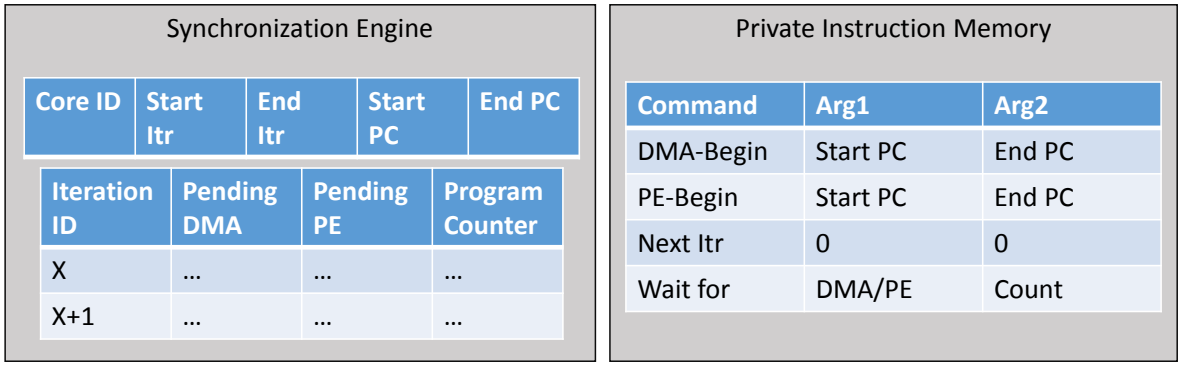


Figure 9.2: Internal design of the Synchronization Engine (SE).

DMA-E and PE with a hardware queue instructing the DMA-E or PE what tasks to perform, and a single signal leading back to the SE indicating that a task has been completed. Each component of an accelerator features a small instruction memory which holds pre-decoded commands for just that component.

The SE, shown in Figure 9.2 is responsible for tracking dependencies on a coarse grain between memory reads/write and compute. It does this by sending a message to either the DMA-E or PE, and waiting for a response indicating completion. Commands that are sent to the DMA-E or PE are bundled into coarse grain segments, between which dependencies are tracked. The instruction set for the SE allows for four types of commands: 1) command to DMA-E, 2) command to PE, 3) wait-on device, and 4) next-iteration signal. Commands to either the DMA-E or PE feature start and end indicies in the respective engines instruction memory, and result in a message being enqueued in the message queues leading to the respective engines. The wait-on instruction explicitly stalls for responses from the DMA-E or PE, to enable coarse grain dependency tracking. Because the majority of the work that the accelerator does is loop processing, the SE also features a "next iteration" command, which enables scheduling of commands for the next iteration in a loop where applicable. The state tracking components within the SE, such as program counters and iteration counters, are replicated twice, to allow for concurrent execution of two loop iterations simultaneously.

The DMA-C, is a structure designed for bulk memory transfers from memory. The

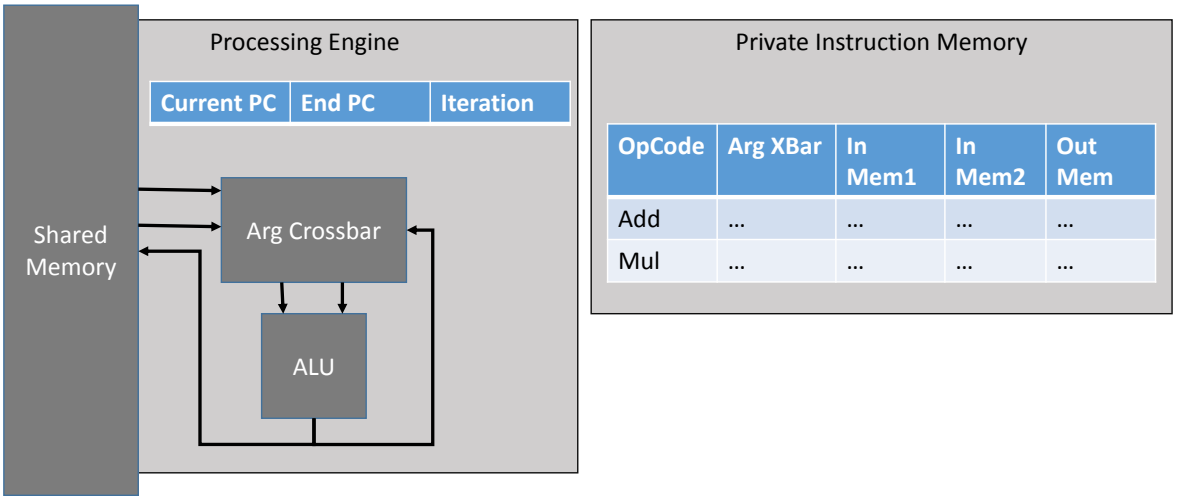


Figure 9.3: Internal design of the Processing Engine (SE).

DMA does not perform any kind of checking for dependencies between memory accesses. Instructions for the DMA-E each consist of a memory base address, two-dimensional element count and stride, and a number of elements to transfer with each command, along with a base address for the accelerator shared memory region that is assumed to be sequentially addressed. The SE sends as part of its command message a "loop index" that is used to calculate the effective address relative to the stored base address. Even though memory accesses may finish out of order, the DMA-E sends finish signals to the SE in-order. There is a small queue leading into the DMA-C that is used to buffer a number of commands from the SE.

The PE, shown in Figure 9.3, is the main compute engine of the accelerator. It performs no dependency tracking or scheduling in hardware, and simply executes one "instruction" each cycle. An instruction in the PE consists of a setting for a crossbar that routes arguments from either register reads or a forward channel from the ALU to the compute engine internal to the PE, an opcode to the ALU, and addresses for accesses to the shared memory space. The compute engine internal to the PE has a fixed latency for any particular operation, and the program running in the PE is authored with knowledge of these latencies. A dummy code is used for the memory address when a memory access is not necessary to save on memory read costs. Everything in this system is statically

scheduled as part of compilation, and everything from the perspective of the PE occurs with a latency known at compile time.

Software interacts with the accelerator by first loading appropriate instructions into each of the three engines, and then sending a message to the SE engine indicating the number of iterations to compute. The accelerator then runs until the computation is completed, and sends a completed signal to the requesting core indicating completion. Shared memory is used for communication of compute results between the accelerator and software. The command queue that the controlling core uses to communicate with the accelerator allows the controlling core to enqueue multiple commands, thus keeping the accelerator busy.

9.1.2 Compilation

Like other programmable accelerator platforms, this one relies heavily on a compiler to achieve performance and program coverage. The compiler has two jobs: 1) finding program regions that can be mapped to the accelerator, and 2) scheduling these regions into accelerator configurations. These two responsibilities will be discussed separately, with a focus on the first responsibility.

9.1.2.1 Region Selection

The primary target of regions for acceleration are loops. The reason for this is that there is an overhead associated with configuring the accelerator prior to being used. As such, the more work that can be performed by a single configuration, the greater the advantage that can potentially be gained with the use of an accelerator. In addition to this, the accelerator used in this work has facilities to make the execution of highly uniform loops very efficient by internalizing the loop iterator, thus allowing the core to be uninvolved for the entire loop execution. In addition to this, the instruction memory of the accelerator is small compared to the total program size, and as such the accelerator can only be used to target small hot program regions.

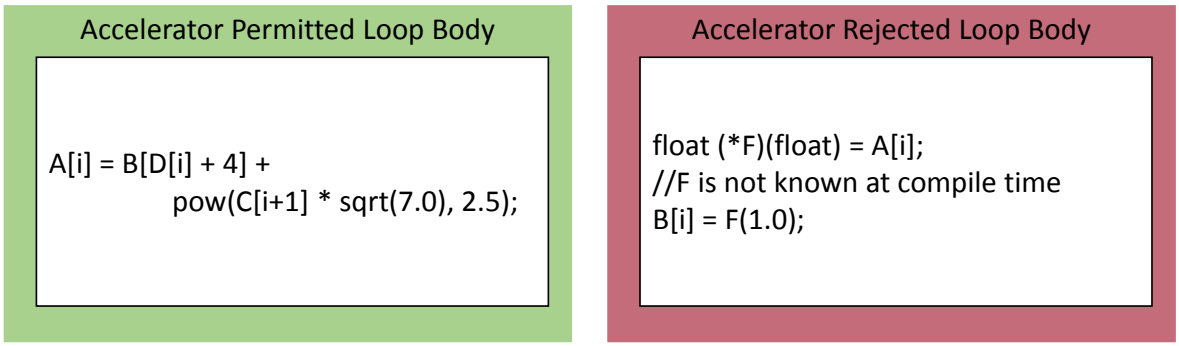


Figure 9.4: Examples for admissible and inadmissible loop bodies.

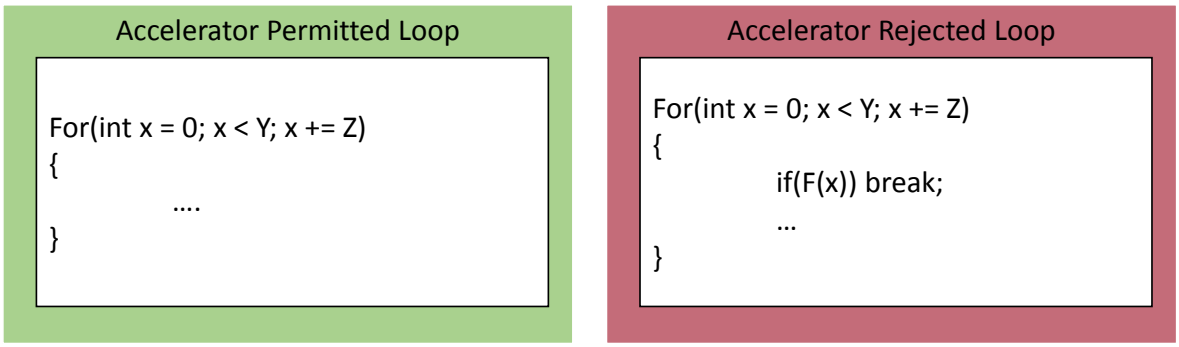


Figure 9.5: Examples for admissible and inadmissible loops, not including loop bodies.

The compilation strategy used in this work targets loops. Loops are broken up into three categories: 1) cannot be accelerated, 2) the body can be accelerated, but the loop itself cannot, and 3) the loop and loop body can be accelerated. A loop that can be accelerated is one for which there is a single entrance, a single exit, and a loop iteration count that can be computed prior to entering the loop. A loop body that can be accelerated is one which does not itself contain loops, or an identifiable bound on flow control, and consists entirely of operations that the either memory transfers or operations that the PE is capable of performing. In short, an acceleratable loop body is one which consists of straight-line code that can be completely inlined or unrolled. An example of acceleratable and unacceleratable structures are shown in Figure 9.4 and Figure 9.5 for the body and loop respectively.

If the loop body is acceleratable, it is passed to the scheduling step described in

Section 9.1.2.2. If the loop is acceleratable, it is replaced entirely with an accelerator invocation. If the body is acceleratable but the loop is not, the controlling core executes the loop flow control component, and replaces the body of the loop with a message to the command queue to the accelerator.

9.1.2.2 Program Scheduling

Once a dataflow graph has been selected from a program, it must be translated into an accelerator schedule. The dataflow graph is first broken up into maximal contiguous components of memory transfers and computation. This allows for more memory transfers to be scheduled simultaneously. Scheduling the memory transfer components is straight forward, with each memory transfer becoming part of a DMA-C instruction. Because there is no possibility for dependencies between co-scheduled memory accesses, nothing more needs to be done for this component.

The portions of the dataflow graph that constitute computation are passed to a solver to find an optimal schedule. The output of the scheduler is a setting for each multiplexer and arithmetic unit in the PE. Because of a consistent issue rate of one instruction per cycle, along with no hardware assistance for data forwarding, the scheduler is responsible for asserting that there are no conflicts on lines used for data forwarding or write back to the local memory. If the scheduler cannot find a schedule such that the produced program can fit in the available instruction memories for all components of the accelerator, the region cannot be accelerated, and is instead executed on the CPU, or broken into smaller parts and re-submitted to the program scheduling pass.

9.2 Methodology

To evaluate the impact on a core of introducing a programmable accelerator to the system, the system shown in Figure9.6 was used. This system featured a basic two-level cache, with stream prefetching in the L2 cache. The L1 cache was accessible only by the core, with the accelerator accessing the L2 cache directly. Table 9.1 describes the specific

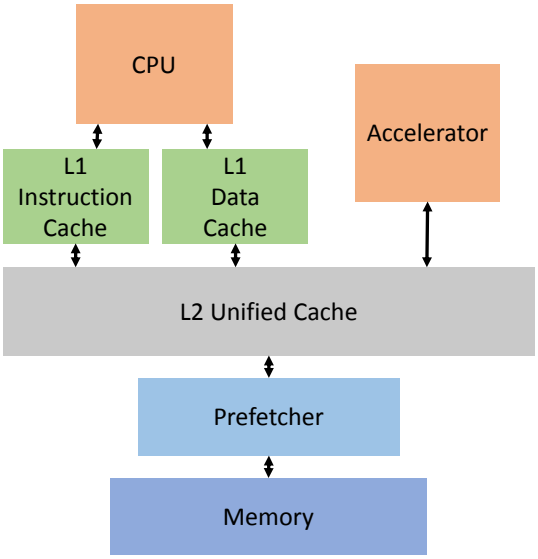


Figure 9.6: System architecture of evaluated system.

sizing and composition of various components of the evaluated system.

Measurements of resource utilization were gathered with the use of dynamic binary instrumentation with PIN [96]. This was used instead of a cycle accurate simulator because this study is focused on the utility of specific subcomponents of a system, rather than the contribution those components make to performance. Using a cycle accurate simulator is very time consuming compared to the used PIN-based implementation, which would limit the scope of the study being performed.

9.2.1 Benchmark

Chosen to evaluate this work were a set of workloads from the PARSEC [39] benchmark suite. The only restriction placed on the selection of workloads was that the programs not require linking to any libraries that constituted a substantial portion of the spent compute time. This is because this technique required the entire program source code to be visible to the compiler, and linking against precompiled libraries limited the capability of the compiler to make decisions about what could potentially run on the accelerator. Besides this, workloads were taken unmodified.

Table 9.1: Characteristics of the evaluated system

Component	Configuration
Processor	x86, 2-bit branch prediction, 64-entry I-fetch line cache, 128-instruction window, 4-wide issue
Accelerator- DMA	32-entry instruction memory size, 2D- parameterized address generation
Accelerator- PE	1 FP-ALU, 1 Int-ALU, 4-Read 2-Write memory access, 512-entry instruction memory
Accelerator- SE	64-entry instruction memory
Accelerator- Memory	512 64-bit memory locations.
L1 Cache	Private 32KB, split instruction/data, 4-way set associative, MESI coherence, 64-byte Block size
L2 Cache	Shared 8MB, 8-way set associative, 64-byte Block size
Prefetcher	32-stream strided prefetcher. Accessed upon L2 miss

Benchmarks were compiled with a modified version of LLVM [36] that was extended to identify accelerator candidate locations, and schedule accelerator programs. All other compilation passes are performed prior to searching for accelerator candidates, and there were no program transformations to morph the program into a form that may be more amenable for acceleration. While there were opportunities in the evaluated programs to expose additional opportunity for acceleration by modifying program structure, these opportunities were not examined within this work.

9.3 Results

Table 9.2 presents all experimental results which will be discussed in the following sections.

Table 9.2: Impact on processor resource utilization over all benchmarks. Rows indicating removed events show number of events that occur in the CPU only case as compared to the CPU and accelerator case. Acc- indicates systems that feature only conventional CPUs, while Acc+ indicates systems that feature CPUs and accelerators. All numbers are percentages. Values that were greater than 99.95% were rounded up to 100

Device	Black Scholes	Canneal	Ferret	Fluid Animate	Freqmine	Stream Cluster	Swaption	x264
Dynamic Instr removed	100	83.4	76.7	97.8	49.8	86.8	74.2	92.1
Acc- branch predict rate	97.4	80.2	85.5	93.9	90.8	94.9	72.1	77.9
Acc+ branch predict rate	98	84	84.6	92.8	92.8	97.2	86.8	86.2
L1-D Accesses removed	100	83.5	70.6	92.9	37.4	74.9	74.6	90.7
L1-D Acc- hit rate	98	98.4	99.7	99.8	99.7	97.0	99.8	99.8
L1-D Acc+ hit rate	99	98.6	99.6	99.3	99.8	100	100	100
Prefetch- Streams removed	100	81.8	40	72.7	42.7	45	44.1	89.9
Prefetch- Acc- hit rate	99.9	0.1	41.4	80.3	59	97.9	50.7	61.5
Prefetch- Acc+ hit rate	0.7	0.1	32.3	64.7	56.5	97.8	65.8	46.6

9.4 Accelerator Resource Utilization

Most loops that could be targeted by accelerators in this work naturally consisted of only a small number of operations. In order to achieve high accelerator utilization, the scheduling step used in this work attempts to unroll these loops so as to increase the number of operations in a single loop body. Unrolling was done until some resource was exhausted, either the shared memory internal to the accelerator, or the instruction memories of any of the accelerators' engines. This allows for a greater number of memory accesses to be concurrently outstanding, and also reduces the degree to which the accelerator as a whole is sensitive to memory latency. Because of this strategy, instead of discussing memory contention or instruction memory occupancy, a more interesting result to focus on is the proportion of time that the compute engine within the accelerator remains busy, as this indicates the proportion of peak compute throughput that is being achieved.

While not shown in the results of Table 9.2, it is worth noting that the program that targets an accelerator-featuring system was identical to the CPU only program for over 95% of the total number of static instructions. Only a select number of small loops were shifted to the accelerator, even in the cases where the the accelerator reduced the number of dynamic instructions greatly.

9.5 Core Utilization

In accelerator-featuring systems, for all evaluated workloads except Freqmine, the overwhelming majority of computational work ended up shifted to the accelerator. The role of the CPU was then limited to executing irregular code, and segments of control code between accelerated loops. Even though the types of loops that were no longer executed on the CPU tend to contain branches that are highly predictable, branch prediction was found to be similarly effective in both evaluated systems. This was found to be attributable primarily to two main sources: 1) accelerator-targetable code that resides

outside the scope of the optimized code, such as calls to the memcpy function, or string processing functions, and 2) error-checking code that is trivially predictable under normal operating conditions as being not-taken branches.

Dependencies among nearby Instructions were also found to be more common in code that could not be accelerated. Intuitively this makes sense, since code regions that are accelerator friendly are also regions that loop unrolling can be used to arbitrarily increase potential ILP in program code. Even if loop unrolling isn't performed, the branch instructions used to construct these types of loops are highly predictable. Measurements of potential ILP, which was performed by counting the number of independent instructions could be found in the 256 entry instruction window that could potentially be co-scheduled with the current front of the instruction window, revealed that for all evaluated workloads code regions that could not be accelerated contained on average 40%-60% less potential ILP as compared to acceleratable regions.

9.5.1 Memory System Utilization

Even though only a small portion of the total program code was moved to the accelerator, the accelerator emitted the majority of cache-missing memory accesses, primarily because the accelerator frequently operated over data volumes that are larger than the L1 cache size. Much of the data over which accelerators operated was highly structured, and is the type of access pattern for which prefetchers have been shown to be highly effective. While these access streams are predictable, a high prefetcher hit rate would not actually bring value to the system as a whole due to the accelerator being insensitive to the memory latency that the prefetcher helps eliminate. In several cases the prefetcher continued to show considerable value to the core, but these cases were found to consist primarily of cases in which the core would touch data the accelerator had worked on after the accelerator was finished in an way that could not be migrated to the accelerator. This would cause data to be re-fetched. This problem could be more effectively dealt with by employing a tiling strategy, and reusing data in the core while it is still resident in

the shared L2 cache after having been fetched by the accelerator, but this was not done in this work. Other than this, which could be more appropriately be dealt with during compilation, the prefetcher was not observed to provide much utility in the evaluated system.

In addition to impacting the utility of the prefetcher, L1 cache for the conventional processor is no longer exposed to the data that the accelerator touches. This allows a smaller L1 cache to achieve a similar performance as would be expected to be seen in a larger L1, since the L1 cache primarily serves the purpose of keeping program stack data resident instead of more general data. While not shown above, it was observed that the L1 cache can be shrunk from 32K to 4K in a system with accelerators with less than 0.2% impact on hit rates for most workloads.

CHAPTER 10

Related Work

10.1 On-Chip Accelerators

There is a large amount of work that implements an application-specific coprocessor or accelerator through either ASIC or FPGA [97, 98]. These works mostly consider a single accelerator dedicated to a single application. Convey [99] and Nallatech [100], 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 [101], UltraSPARC T2 [6], Intel's Larrabee [5] and IBM's WSP processor [4] are examples of this. Most of these platforms (except WSP) are tightly coupled with processor cores (or core-clusters). Our work 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 [102]. In contrast, we focus on hardware support for accelerator management.

There have also been a number of recent designs of heterogeneous architectures, like EXOCHI [103], SARC [27], and HiPPAI [104]. Similar to our work, EXOCHI's focus is on a heterogeneous non-uniform ISA. HiPPAI, like our work, aims to eliminate 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 [105] 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 [54] 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 [50] implements an FPGA like accelerator, with fixed functionality arithmetic units coupled with a configurable communication network. Our work distinguishes 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.

10.2 Off-Chip Near-Memory Accelerators

Studies investigating integrating small accelerators directly into DRAM memory has previously been investigated. Works such as CRAM [45], IRAM [46], DIVA [47] and FlexRam [48] fall into this category. These architectures allow for exploitation of the massive bandwidth that is available to structures internal to a DRAM chip, prior to selecting data for transmission across the DRAM packaging. While the bandwidth of these architectures is much larger than our proposal has access to, the design and implementation complexity of these types designs is very high, and the need to integrate accelerators directly into the internal structure of DRAM limits the complexity and expressiveness of the integrated compute engines. The AIM proposal does not disrupt memory design, thus can take advantage of off-the-shelf components, and does not impose any significant restrictions on the type or level of sophistication of compute engines.

Architectures have also been proposed that argue for stacking a high density SRAM directly onto the main processor, as either a very large last level cache or an embedded memory. [106]. While the capacity of these designs provide large bandwidth and reducing

communication latency to the integrated memory, they still do not provide the storage capacity of conventional memory systems. 3D integration also is still a maturing technology. While we view this technology as being an interesting enhancement to a system featuring accelerators, we view it as an orthogonal topic due to the disparity of capacity between integrated memory and a more conventional memory system.

Integrating accelerators or FPGAs into a DIMM form-factor has also been previously explored. Copacobana [107] and Pilchard [108] both identified the memory bus as a convenient solution to the need for a high bandwidth and low latency interconnect between FPGAs and the CPU, and created boards that conform to a DIMM form factor with FPGAs embedded. These designs have the advantage of limiting design overhead of implementing an accelerator platform. Neither project however features memory on board, and could not be communicated with using conventional memory protocols, and thus could not be used in a system intended to run conventional programs or boot a conventional operating system. For these reasons, these modules were mostly appropriate for embedded, or otherwise specialized systems. Our proposal strives to be transparent to the surrounding system system, so as to be usable in existing systems with minimal overhead.

10.3 Emerging Network Technology

To provide guaranteed quality of service in NoC in terms of throughput and latency, hybrid circuit switching and packet switching is introduced in recent work. The key is to dynamically construct virtual circuit. The first categories of work use Time-Division-Multiplexing (TDM). In a particular time-interval, the available network bandwidth is exclusively dedicated to a virtual circuit. One of the representative work is the Philips Ethereal [109] which uses two separate NoC: a Guaranteed Service (GS) circuit switching sub-network and a Best-Effort (BE) packet-switching sub-network, where the BE network is used to configure circuit switching in the GS network. TDM-based circuit-switching is particularly well adapted for long and frequent messages like multimedia streams.

However, it suffers long circuit set-up time and complex commutation scheduling. Recent work in [110] uses a hybrid router design which intermingles circuit-switched flits and packet-switched flits to reduce the circuit setup time. Spatial-Division-Multiplexing (SDM) is later introduced in [111] and enhanced by [112] and [113] to further reduce the circuit setup overhead by allocating a sub-set of the link wires to a given circuit for the whole connection lifetime. The proposed stream arbitration is similar to virtual circuit switching in the way of dynamically allocating communication channels between requested communication nodes. However, stream arbitration distinguishes the previous virtual circuit switching in that it circulates the information of available communication resources and the competing senders across all the arbitration participants, thus provide much higher resource utilization and global fairness. Such information is impractical to be circulated in the traditional RC-wire-based NoC, as it will incur significant latency and power overhead. Therefore, stream arbitration is not suitable for traditional RC-wire-based NoC. It is dedicated for the emerging high-bandwidth low-latency interconnects, such as RF-interconnect and optics.

To efficiently utilize the high bandwidth provided by emerging interconnects, researchers in [81] propose application-specific shortcuts which can be realized by dynamically tuning the on-chip RF transceiver frequencies. Shortcut is allocated based on the communication profiling of the application so that intensively communicated nodes will be allocated RF bandwidth. This approach is suitable for MPSoCs where the communication pattern is predictable or applications with long and frequent messages like multimedia streams. However, it can not well adapt to the dynamic runtime variation in general-purpose CMPs. Token arbitration is proposed in [75, 80] to perform runtime arbitration of the communication channels by token-passing among the arbitration participants. Therefore, it works well for general-purpose CMPs with unpredictable dynamic variation behaviors. However, the communication bandwidth allocated to each receiver is fixed and may result in bandwidth waste when that receiver is not frequently used. The proposed stream arbitration distinguishes from [81] by allocating communication resource at runtime, and distinguishes from [75, 80] by allowing for any communication pairs to

make use of communication channel, in order to maximize the bandwidth utilization.

The curl-based arbitration channel used in stream arbitration is similar to the Intel ring-based communication architecture [114], but with a much higher transmission latency and bandwidth because of the use of RF interconnect. Since it is difficult for the RF signal be transmitted in a closed-loop ring due to the impedance switching, therefore we use a curl style transmission line to allow the signal to take a round-trip of all the RF nodes. Moreover, the introducing of curl also allows us to pipeline the stream arbitration as the first trip is physically disjoint from the second trip.

CHAPTER 11

Conclusion

Increasing expectations for power efficiency and performance demands, combined with a glut of transistors provided by ever improving fabrication technology, has made including accelerators in compute systems an attractive option. When combined with advancements that have been made in the area of high-level synthesis, the cost associated with introducing accelerators is very low. The barrier that blocks accelerators from playing a more central roll in commodity systems are the challenges associated with using accelerators by normal software and cooperating with other components in a normal system. This work focused primarily on these issues. Compilation technology was detailed that allowed for common programming idioms to be automatically translated to programs that run on accelerators, architectural advancements were proposed to manage and interact with accelerators in a way that doesn't require specialized arbitrators in operating systems, and accelerators were introduced of sufficient simplicity to be both efficient and compiler friendly.

While this work does not argue that the accelerators discussed are ideal choices, it illustrates that even simple accelerators like those discussed in Chapter 3 are sufficient to get large performance and energy efficiency gains. Furthermore, future efforts in choosing better accelerator designs could make use of the architectural and compiler frameworks presented in this work to be rapidly introduced to a system while not requiring application software modification or operating system modification.

11.1 Future work

While accelerators satisfy the need of performing computations at both high performance and with low energy consumption, the rest of the system still remains as it was. As was hinted at in Chapter 4 and Chapter 6, the cost of computation constitutes an ever shrinking proportion of total system energy. In modern commodity processors with highly aggressive out-of-order processors, the compute core itself still constitutes a large portion of both the area used and the energy consumed, but in systems that feature highly efficient compute cores, the energy consumption shifts heavily to supplying a clock, memory, and the on-chip interconnect.

Accelerators like those discussed within this work, which operate asynchronously, and often autonomously, present many interesting potential directions to lowering energy consumption of these other components as well. The system is naturally asynchronous and highly fragmented, and in many instances has no need for a centralized clock or tight coupling over large areas. This allows for the potential to put compute engines in a variety of places, and reduce clock and interconnect energy consumption by moving the compute to the data, similar to what was preliminarily explored in Chapter 6. The system is also naturally highly redundant, which potentially opens up possibilities to experiment with near-threshold designs to further save power and address topics of fault tolerance.

Within the context of this work, accelerators were all hand selected. There has been a wealth of prior work in selecting accelerator candidates out of samples of program code, typically for the purposes of developing custom instructions, but no work that incorporates the potential data-flow style compute paradigm emphasized within this work. Custom instructions have the benefit of not requiring complete coverage, while systems like those discussed in this work do not.

Bibliography

[1] *ITRS. International Technology roadmap for semiconductors, 2012 edition.* http://www.itrs.net/Links/2012ITRS/Home2012.htm., .

[2] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 365–376. IEEE, 2011.

[3] Patrick Schaumont and Ingrid Verbauwhede. Domain-specific codesign for embedded security. *Computer*, 36(4):68–74, 2003.

[4] Hubertus Franke, Jimi Xenidis, Claude Basso, Brian M Bass, Sandra S Woodward, Jeffrey D Brown, and Charles L Johnson. Introduction to the wire-speed processor and architecture. *IBM Journal of Research and Development*, 54(1):3–1, 2010.

[5] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, et al. Larrabee: a many-core x86 architecture for visual computing. In *ACM Transactions on Graphics (TOG)*, volume 27, page 18. ACM, 2008.

[6] Umesh Nawathe, Mahmudul Hassan, Lynn Warriner, King Yen, Bharat Upputuri, David Greenhill, Ashok Kumar, and Heechoul Park. An 8-core, 64-thread, 64-bit, power efficient sparc soc (niagara 2). *ISSCC, http://www. opensparc. net/pubs/preszo/07/n2isscc. pdf*, 2007.

[7] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation cores: reducing the energy of mature computations. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 205–218. ACM, 2010.

[8] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, and Glenn

Reinman. Architecture support for accelerator-rich cmps. In *Proceedings of the 49th Annual Design Automation Conference*, pages 843–849. ACM, 2012.

[9] Markus Puschel, José MF Moura, Jeremy R Johnson, David Padua, Manuela M Veloso, Bryan W Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, et al. Spiral: Code generation for dsp transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.

[10] Luminita A Vese and Stanley J Osher. Image denoising and decomposition with total variation minimization and oscillatory functions. *Journal of Mathematical Imaging and Vision*, 20(1-2):7–18, 2004.

[11] Eitan Tadmor, Suzanne Nezzar, and Luminita Vese. Multiscale hierarchical decomposition of images with applications to deblurring, denoising and segmentation. Technical report, DTIC Document, 2007.

[12] Igor Yanovsky, Carole Le Guyader, Alex Leow, Paul Thompson, and Luminita Vese. Nonlinear elastic registration with unbiased regularization in three dimensions. In *Computational Biomechanics for Medicine III, MICCAI 2008 Workshop*, 2008.

[13] Tony F Chan and Luminita A Vese. Active contours without edges. *Image processing, IEEE transactions on*, 10(2):266–277, 2001.

[14] S. Lazebnik, C. Schmid, and J. Ponce. A sparse texture representation using local affine regions. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 27:1265–1278, 2005.

[15] Frédéric Jurie. A new log-polar mapping for space variant imaging : Application to face detection and tracking. *Pattern Recognition*, 32(5):865–875, May 1999.

[16] H. Bay, A. Ess, T. Tuytelaars, and L. Van Gool. Surf: Speeded up robust features. *CVIU*, 110(3):346–359, 2008.

[17] Jose-Luis Blanco. Derivation and implementation of a full 6d ekf-based solution to bearing-range slam. Technical report, University of Malaga, Spain, http://babel.isa.uma.es/~jlblanco/papers/RangeBearingSLAM6D.pdf, Mar 2008.

[18] Matteo Frigo and Steven G Johnson. The design and implementation of fftw3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[19] Peter S Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[20] Milo MK Martin, Daniel J Sorin, Bradford M Beckmann, Michael R Marty, Min Xu, Alaa R Alameldeen, Kevin E Moore, Mark D Hill, and David A Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *ACM SIGARCH Computer Architecture News*, 33(4):92–99, 2005.

[21] Zhiru Zhang, Yiping Fan, Wei Jiang, Guoling Han, Changqi Yang, and Jason Cong. Autopilot: A platform-based esl synthesis system. In *High-Level Synthesis*, pages 99–112. Springer, 2008.

[22] Synopsys. Synopsys Design Compiler. `http://www.synopsys.com/ Tools/ Pages/ default.aspx`, 2013.

[23] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 469–480. IEEE, 2009.

[24] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, and Glenn Reinman. Charm: A composable heterogeneous accelerator-rich microprocessor. In *Proceedings of the 2012 ACM/IEEE international symposium on Low power electronics and design*, pages 379–384. ACM, 2012.

[25] Jason Cong, Hui Huang, and Wei Jiang. A generalized control-flow-aware pattern recognition algorithm for behavioral synthesis. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 1255–1260, 2010.

[26] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, 2006.

[27] A. Ramirez, F. Cabarcas, B. Juurlink, M. Alvarez Mesa, F. Sanchez, A. Azevedo, C. Meenderinck, C. Ciobanu, S. Isaza, and G. Gaydadjiev. The sarc architecture. *Micro, IEEE*, 30(5):16 –29, sept.-oct. 2010.

[28] Premkishore Shivakumar and Norman P Jouppi. Cacti 3.0: An integrated cache timing, power, and area model. Technical report, Technical Report 2001/2, Compaq Computer Corporation, 2001.

[29] Hang-Sheng Wang, Xinping Zhu, Li-Shiuan Peh, and Sharad Malik. Orion: a power-performance simulator for interconnection networks. In *Microarchitecture, 2002.(MICRO-35). Proceedings. 35th Annual IEEE/ACM International Symposium on*, pages 294–305. IEEE, 2002.

[30] Jason Cong, Beayna Grigorian, Glenn Reinman, and Marco Vitanza. Accelerating vision and navigation applications on a customizable platform. In *Application-Specific Systems, Architectures and Processors (ASAP), 2011 IEEE International Conference on*, pages 25–32. IEEE, 2011.

[31] Jason Cong, Vivek Sarkar, Glenn Reinman, and Alex Bui. Customizable domain-specific computing. *IEEE Design and Test of Computers*, 28(2):6–15, 2011.

[32] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, Hui Huang, and Glenn Reinman. Composable accelerator-rich microprocessor enhanced for adaptivity and longevity. In *Low Power Electronics and Design (ISLPED), 2013 IEEE International Symposium on*, pages 305–310. IEEE, 2013.

[33] Ian Kuon and Jonathan Rose. Measuring the gap between fpgas and asics. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(2):203–215, 2007.

[34] Jason Cong and Bingjun Xiao. Optimization of interconnects between accelerators and shared memories in dark silicon. In *Proceedings of the International Conference on Computer-Aided Design*, pages 630–637. IEEE Press, 2013.

[35] Amir Hormati, Nathan Clark, and Scott Mahlke. Exploiting narrow accelerators with data-centric subgraph mapping. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 341–353. IEEE Computer Society, 2007.

[36] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[37] Xilinx vivado design suite, . URL `http://www.xilinx.com/ products/ design-tools/ vivado/ index.htm`.

[38] Xilinx ise design suite, . URL `http://www.xilinx.com/ products/ design-tools/ ise-design-suite/ index.htm`.

[39] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81. ACM, 2008.

[40] OpenCV. *Open Source Computer Vision*. http://opencv.willowgarage.com/.

[41] MRPT. *The Mobile Robot Programming Toolkit*. http://mrpt.org/.

[42] Sravanthi Kota Venkata, Ikkjin Ahn, Donghwan Jeon, Anshuman Gupta, Christopher Louie, Saturnino Garcia, Serge Belongie, and Michael Bedford Taylor. Sd-vbs:

The san diego vision benchmark suite. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 55–64, Washington, DC, USA, 2009. IEEE Computer Society.

[43] Michael Lustig, David Donoho, and John M Pauly. Sparse mri: The application of compressed sensing for rapid mr imaging. *Magnetic resonance in medicine*, 58(6): 1182–1195, 2007.

[44] Wm A Wulf and Sally A McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995.

[45] Duncan G Elliott, Michael Stumm, W Martin Snelgrove, Christian Cojocaru, and Robert McKenzie. Computational ram: Implementing processors in memory. *Design & Test of Computers, IEEE*, 16(1):32–41, 1999.

[46] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A case for intelligent ram. *Micro, IEEE*, 17(2):34–44, 1997.

[47] Mary Hall, Peter Kogge, Jeff Koller, Pedro Diniz, Jacqueline Chame, Jeff Draper, Jeff LaCoss, John Granacki, Jay Brockman, Apoorv Srivastava, et al. Mapping irregular applications to diva, a pim-based data-intensive architecture. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 57. ACM, 1999.

[48] Yi Kang, Wei Huang, Seung-Moon Yoo, Diana Keen, Zhenzhou Ge, Vinh Lam, Pratap Pattnaik, and Josep Torrellas. Flexram: Toward an advanced intelligent memory system. In *Computer Design (ICCD), 2012 IEEE 30th International Conference on*, pages 5–14. IEEE, 2012.

[49] Roger Espasa, Federico Ardanaz, Joel Emer, Stephen Felix, Julio Gago, Roger Gramunt, Isaac Hernandez, Toni Juan, Geoff Lowney, Matthew Mattina, et al. Tarantula: a vector extension to the alpha architecture. In *Computer Architecture,*

2002. Proceedings. 29th Annual International Symposium on, pages 281–292. IEEE, 2002.

[50] Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. Dyser: Unifying functionality and parallelism specialization for energy-efficient computing. *IEEE Micro*, 32(5):0038–51, 2012.

[51] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: using data parallelism to program gpus for general-purpose uses. In *ACM SIGARCH Computer Architecture News*, volume 34, pages 325–335. ACM, 2006.

[52] Souradip Sarkar, Gaurav Ramesh Kulkarni, Partha Pratim Pande, and Ananth Kalyanaraman. Network-on-chip hardware accelerators for biological sequence alignment. *Computers, IEEE Transactions on*, 59(1):29–41, 2010.

[53] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th annual symposium on Computer Architecture*, pages 81–87. IEEE Computer Society Press, 1981.

[54] Hyunchul Park, Yongjun Park, and Scott Mahlke. Polymorphic pipeline array: a flexible multicore accelerator with virtualized execution for mobile multimedia applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 370–380. ACM, 2009.

[55] Louis-Noël Pouchet, Peng Zhang, P. Sadayappan, and Jason Cong. Polyhedral-based data reuse optimization for configurable computing. In *21st ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'13)*, Monterey, California, February 2013. ACM Press.

[56] Justin Holewinski, Louis-Noël Pouchet, and P. Sadayappan. High-performance code generation for stencil computations on gpu architectures. In *ACM International Conference on Supercomputing (ICS'12)*, Venice, Italy, June 2012. ACM Press.

[57] Joe Jeddeloh and Brent Keeth. Hybrid memory cube new dram architecture increases density and performance. In *VLSI Technology (VLSIT), 2012 Symposium on*, pages 87–88. IEEE, 2012.

[58] Kanit Therdsteerasukdi, Gyung-Su Byun, Jeremy Ir, Glenn Reinman, Jason Cong, and MF Chang. The dimm tree architecture: A high bandwidth and scalable memory system. In *Computer Design (ICCD), 2011 IEEE 29th International Conference on*, pages 388–395. IEEE, 2011.

[59] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick. The potential of the cell processor for scientific computing. In *Proceedings of the 3rd conference on Computing frontiers*, pages 9–20. ACM, 2006.

[60] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[61] Michael Lyons, Mark Hempstead, Gu-Yeon Wei, and David Brooks. The accelerator store framework for high-performance, low-power accelerator-based systems. *Computer Architecture Letters*, 9(2):53–56, 2010.

[62] Carlos Flores Fajardo, Zhen Fang, Ravi Iyer, German Fabila Garcia, Seung Eun Lee, and Li Zhao. Buffer-integrated-cache: A cost-effective sram architecture for handheld and embedded platforms. In *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pages 966–971. IEEE, 2011.

[63] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Chunyue Liu, and Glenn Reinman. Bin: a buffer-in-nuca scheme for accelerator-rich cmps. In *Proceedings of the 2012 ACM/IEEE international symposium on Low power electronics and design*, pages 225–230. ACM, 2012.

[64] Jason Cong, Peng Zhang, and Yi Zou. Combined loop transformation and hierarchy allocation for data reuse optimization. In *Proceedings of the International Conference on Computer-Aided Design*, pages 185–192. IEEE Press, 2011.

[65] Moinuddin K Qureshi and Yale N Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 423–432. IEEE Computer Society, 2006.

[66] Sangyeun Cho and Lei Jin. Managing distributed, shared l2 caches through os-level page allocation. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 455–468. IEEE Computer Society, 2006.

[67] Bradford M Beckmann, Michael R Marty, and David A Wood. Asr: Adaptive selective replication for cmp caches. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pages 443–454. IEEE, 2006.

[68] Dan Quinlan. Rose: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(02n03):215–226, 2000.

[69] Ilya Issenin, Erik Brockmeyer, Miguel Miranda, and Nikil Dutt. Drdu: A data reuse analysis technique for efficient scratch-pad memory management. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 12(2):15, 2007.

[70] .

[71] Jason Cong, Hui Huang, Chunyue Liu, and Yi Zou. A reuse-aware prefetching scheme for scratchpad memory. In *Proceedings of the 48th Design Automation Conference*, pages 960–965. ACM, 2011.

[72] Jason Cong, Peng Zhang, and Yi Zou. Combined loop transformation and hierarchy allocation for data reuse optimization. In *Proceedings of the International Conference on Computer-Aided Design*, pages 185–192. IEEE Press, 2011.

[73] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Chunyue Liu, Glenn Reinman, and Yi Zou. Axr-cmp: Architecture support in accelerator-rich cmps. In *2nd Workshop on SoC Architecture, Accelerators and Workloads*, 2011.

[74] Alex Bui, Kwang-Ting Cheng, Jason Cong, Luminita Vese, Yi-Chu Wang, Bo Yuan, and Yi Zou. Platform characterization for domain-specific computing. In *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, pages 94–99. IEEE, 2012.

[75] Dana Vantrease, Robert Schreiber, Matteo Monchiero, Moray McLaren, Norman P Jouppi, Marco Fiorentino, Al Davis, Nathan Binkert, Raymond G Beausoleil, and Jung Ho Ahn. Corona: System implications of emerging nanophotonic technology. In *ACM SIGARCH Computer Architecture News*, volume 36, pages 153–164. IEEE Computer Society, 2008.

[76] Yan Pan, Prabhat Kumar, John Kim, Gokhan Memik, Yu Zhang, and Alok Choudhary. Firefly: illuminating future network-on-chip with nanophotonics. *ACM SIGARCH Computer Architecture News*, 37(3):429–440, 2009.

[77] M-C Frank Chang, Eran Socher, Sai-Wang Tam, Jason Cong, and Glenn Reinman. Rf interconnects for communications on-chip. In *Proceedings of the 2008 international symposium on Physical design*, pages 78–83. ACM, 2008.

[78] Chunhua Xiao, Frank Chang, Jason Cong, Michael Gill, Zhangqin Huang, Chunyue Liu, Glenn Reinman, and Hao Wu. Stream arbitration: Towards efficient bandwidth utilization for emerging on-chip interconnects. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):60, 2013.

[79] Rakesh Kumar, Victor Zyuban, and Dean M Tullsen. Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling. In *Computer Architecture, 2005. ISCA'05. Proceedings. 32nd International Symposium on*, pages 408–419. IEEE, 2005.

[80] Dana Vantrease, Nathan Binkert, Robert Schreiber, and Mikko H Lipasti. Light speed arbitration and flow control for nanophotonic interconnects. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 304–315. IEEE, 2009.

[81] M Frank Chang, Jason Cong, Adam Kaplan, Mishali Naik, Glenn Reinman, Eran Socher, and S-W Tam. Cmp network-on-chip overlaid with multi-band rf-interconnect. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pages 191–202. IEEE, 2008.

[82] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Reactive nuca: near-optimal block placement and replication in distributed caches. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 184–195. ACM, 2009.

[83] Theofanis Constantinou, Yiannakis Sazeides, Pierre Michaud, Damien Fetis, and Andre Seznec. Performance implications of single thread migration on a chip multi-core. *ACM SIGARCH Computer Architecture News*, 33(4):80–91, 2005.

[84] Bradford M Beckmann and David A Wood. Managing wire delay in large chip-multiprocessor caches. In *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*, pages 319–330. IEEE, 2004.

[85] M-C Frank Chang, Eran Socher, Sai-Wang Tam, Jason Cong, and Glenn Reinman. Rf interconnects for communications on-chip. In *Proceedings of the 2008 international symposium on Physical design*, pages 78–83. ACM, 2008.

[86] M Frank Chang, Jason Cong, Adam Kaplan, Mishali Naik, Glenn Reinman, Eran Socher, and Sai-Wang Tam. Cmp network-on-chip overlaid with multi-band rf-interconnect. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pages 191–202. Ieee, 2008.

[87] Yanghyo Kim, Gyung-Su Byun, Adrian Tang, Chewn-Pu Jou, Hsieh-Hung Hsieh, Glenn Reinman, Jason Cong, and Mau-Chung Frank Chang. An 8gb/s/pin 4pj/b/pin single-t-line dual (base+rf) band simultaneous bidirectional mobile memory i/o interface with inter-channel interference suppression. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International*, pages 50–52. IEEE, 2012.

[88] Hao Wu, Lan Nan, Sai-Wang Tam, Hsieh-Hung Hsieh, Chewpu Jou, Glenn Reinman, Jason Cong, and Mau-Chung Frank Chang. A 60ghz on-chip rf-interconnect with $\lambda/4$ coupler for 5gbps bi-directional communication and multi-drop arbitration. In *Custom Integrated Circuits Conference (CICC), 2012 IEEE*, pages 1–4. IEEE, 2012.

[89] Sai-Wang Tam, Eran Socher, Alden Wong, and Mau-Chung Frank Chang. A simultaneous tri-band on-chip rf-interconnect for future network-on-chip. In *VLSI Circuits, 2009 Symposium on*, pages 90–91. IEEE, 2009.

[90] Sangyeun Cho and Lei Jin. Managing distributed, shared l2 caches through os-level page allocation. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 455–468. IEEE Computer Society, 2006.

[91] Hyunjin Lee, Sangyeun Cho, and Bruce R Childers. Stimuluscache: Boosting performance of chip multiprocessors with excess cache. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12. IEEE, 2010.

[92] Hyunjin Lee, Sangyeun Cho, and Bruce R Childers. Cloudcache: Expanding and shrinking private caches. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 219–230. IEEE, 2011.

[93] Jason Cong, Chunyue Liu, and Glenn Reinman. Aces: application-specific cycle elimination and splitting for deadlock-free routing on irregular network-on-chip. In *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, pages 443–448. IEEE, 2010.

[94] Jose Duato and Timothy Mark Pinkston. A general theory for deadlock-free adaptive routing using a mixed set of resources. *Parallel and Distributed Systems, IEEE Transactions on*, 12(12):1219–1235, 2001.

[95] Niket Agarwal, Tushar Krishna, Li-Shiuan Peh, and Niraj K Jha. Garnet: A detailed on-chip network model inside a full-system simulator. In *Performance Anal-*

ysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on, pages 33–42. IEEE, 2009.

[96] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM Sigplan Notices*, volume 40, pages 190–200. ACM, 2005.

[97] Dimitris Bouris, Antonis Nikitakis, and Ioannis Papaefstathiou. Fast and efficient fpga-based feature detection employing the surf algorithm. In *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, pages 3–10. IEEE, 2010.

[98] Jason Cong and Yi Zou. Fpga-based hardware acceleration of lithographic aerial image simulation. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2(3):17, 2009.

[99] Convey computer. `http://conveycomputer.com/`, .

[100] Nallatech FSB - development systems. `http://www.nallatech.com/Intel-Xeon-FSB-Socket-Fillers/ fsb-development-systems.html`, .

[101] John R Hauser and John Wawrzynek. Garp: A mips processor with a reconfigurable coprocessor. In *Field-Programmable Custom Computing Machines, 1997. Proceedings., The 5th Annual IEEE Symposium on*, pages 12–21. IEEE, 1997.

[102] Philip Garcia and Katherine Compton. Kernel sharing on reconfigurable multiprocessor systems. In *ICECE Technology, 2008. FPT 2008. International Conference on*, pages 225–232. IEEE, 2008.

[103] Perry H Wang, Jamison D Collins, Gautham N Chinya, Hong Jiang, Xinmin Tian, Milind Girkar, Nick Y Yang, Guei-Yuan Lueh, and Hong Wang. Exochi: architecture and programming environment for a heterogeneous multi-core multithreaded system. In *ACM SIGPLAN Notices*, volume 42, pages 156–166. ACM, 2007.

[104] Paul M Stillwell, Vineet Chadha, Omesh Tickoo, Steven Zhang, Ramesh Illikkal, Ravishankar Iyer, and Don Newell. Hippai: High performance portable accelerator interface for socs. In *High Performance Computing (HiPC), 2009 International Conference on*, pages 109–118. IEEE, 2009.

[105] Nathan Clark, Amir Hormati, and Scott Mahlke. Veal: Virtualized execution accelerator for loops. In *Computer Architecture, 2008. ISCA'08. 35th International Symposium on*, pages 389–400. IEEE, 2008.

[106] SQ Gu, Pol Marchal, Marco Facchini, F Wang, M Suh, D Lisk, and M Nowak. Stackable memory of 3d chip integration for mobile applications. In *Electron Devices Meeting, 2008. IEDM 2008. IEEE International*, pages 1–4. IEEE, 2008.

[107] Tim Guneysu, Timo Kasper, Martin Novotny, Christof Paar, and Andy Rupp. Cryptanalysis with copacobana. *Computers, IEEE Transactions on*, 57(11):1498–1513, 2008.

[108] Philip Heng Wai Leong, Monk-Ping Leong, Ocean YH Cheung, T Tung, CM Kwok, Ming Yiu Wong, and Kin-Hong Lee. Pilcharda reconfigurable computing platform with memory slot interface. In *Field-Programmable Custom Computing Machines, 2001. FCCM'01. The 9th Annual IEEE Symposium on*, pages 170–179. IEEE, 2001.

[109] Kees Goossens, John Dielissen, and Andrei Radulescu. Æthereal network on chip: concepts, architectures, and implementations. *Design & Test of Computers, IEEE*, 22(5):414–421, 2005.

[110] Natalie D Enright Jerger, Li-Shiuan Peh, and Mikko H Lipasti. Circuit-switched coherence. In *Proceedings of the Second ACM/IEEE International Symposium on Networks-on-Chip*, pages 193–202. IEEE Computer Society, 2008.

[111] Anthony Leroy, Paul Marchal, Adelina Shickova, Francky Catthoor, Frédéric Robert, and Diederik Verkest. Spatial division multiplexing: a novel approach for

guaranteed throughput on nocs. In *Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 81–86. ACM, 2005.

[112] Mehdi Modarressi, Hamid Sarbazi-Azad, and Mohammad Arjomand. A hybrid packet-circuit switched on-chip network based on sdm. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 566–569. European Design and Automation Association, 2009.

[113] A Kuti Lusala and J Legat. A hybrid noc combining sdm-based circuit switching with packet switching for real-time applications. In *NORCHIP, 2010*, pages 1–4. IEEE, 2010.

[114] Reid Riedlinger, Ron Arnold, Larry Biro, Bill Bowhill, Jason Crop, Kevin Duda, Eric S Fetzer, Olivier Franza, Tom Grutkowski, Casey Little, et al. A 32 nm, 3.1 billion transistor, 12 wide issue itanium® processor for mission-critical servers. *Solid-State Circuits, IEEE Journal of*, 47(1):177–193, 2012.