CHARM: A Composable Heterogeneous Accelerator-Rich Microprocessor

Jason Cong Mohammad Ali Ghodrat Michael Gill

Beayna Grigorian Glenn Reinman

UCLA CS Department and Center for Domain-Specific Computing
{cong, ghodrat, mgill, bgrigori, reinman}@cs.ucla.edu

ABSTRACT

This work discusses CHARM, a Composable Heterogeneous Accelerator-Rich Microprocessor design that provides scalability, flexibility, and design reuse in the space of accelerator-rich CMPs. CHARM features a hardware structure called the accelerator block composer (ABC), which can dynamically compose a set of accelerator building blocks (ABBs) into a loosely coupled accelerator (LCA) to provide orders of magnitude improvement in performance and power efficiency. Our software infrastructure provides a data flow graph to describe the composition, and our hardware components dynamically map available resources to the data flow graph to compose the accelerator from components that may be physically distributed across the CMP. Our ABC is also capable of providing load balancing among available compute resources to increase accelerator utilization. Running medical imaging benchmarks, our experimental results show an average speedup of 2.1X (best case 3.7X) compared to approaches that use LCAs together with a hardware resource manager. We also gain in terms of energy consumption (average 2.4X; best case 4.7X).

Categories and Subject Descriptors

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

General Terms

Design

Keywords

Chip Multiprocessor, Hardware Accelerators, Accelerator Composition

1. INTRODUCTION

Domain-specific architectures [4] have emerged to satisfy demands for power-efficient and high-performance chip multiprocessors. Such architectures feature heterogeneous resources that are designed for a particular application domain. One important component in application-specific or domain-specific hardware is the *loosely coupled accelerator (LCA)*, which acts independently of individual cores and is more amenable to sharing. An example of this could be a dedicated encryption unit that runs a particular encryption algorithm on data it receives from different cores that share access to the accelerator [15].

ISLPED'12, July 30–August 1, 2012, Redondo Beach, CA, USA.

LCAs have their own strengths and weaknesses. LCAs can help provide many orders of magnitude improvement in power-efficiency and performance by offloading computation from the less power-efficient cores entirely [13] [9]. However, it is difficult to integrate sufficient numbers of LCAs to satisfy all execution scenarios, particularly in application domains with aggressive performance requirements and a wide range of applications and implementations. Moreover, the degree of heterogeneity inherent in these *accelerator-rich* architectures becomes cumbersome to manage, especially considering the lack of block reusability and the lack of flexibility for future generations of software within the same domain.

We observe three key trends that motivate our proposed design. First, the tasks performed by LCAs tend to have a great deal of data parallelism, which can be exploited by the use of multiple LCAs of the same type. Second, there is often a variety of LCAs (with possible overlap), required by different applications, or regions of a single application, within a domain. This results in the utilization of any particular LCA being somewhat sporadic. These two trends illustrate that it would be expensive to have both sufficient diversity of LCAs to handle the various applications that comprise a domain and sufficient quantity of a particular LCA to handle the parallelism that exists within a particular accelerator task. Third, we observe that LCAs for a single domain have a large degree of similarity in terms of type of computation primitives being used, indicating that these LCAs can be built using a limited number of smaller, more general LCAs that we call *accelerator building blocks (ABBs)*.

There has been some preliminary work exploring hardware management of LCAs [13] [9], but they do not have a hardware mechanism to effectively leverage multiple accelerators for a single task (i.e. effectively load balance among cooperating accelerators), and therefore cannot leverage the parallelism we observe. Moreover, they do not exploit the sporadic nature of accelerator use, and would need to implement an LCA for each function they want to accelerate.

To address these concerns, we propose a multi-core design where loosely coupled ABBs are distributed in islands around the chip and may be dynamically composed together into an LCA. Thus, the ABBs are effectively the building blocks of our application-specific accelerators. Instead of having discrete LCAs for every desired accelerator function, the use of ABBs gives us tremendous flexibility in composing different accelerators at different times, thereby better matching application demand. In addition, we may better share resources at this fine granularity and improve ABB utilization. From a design perspective, the use of ABBs creates more regularity and static homogeneity in a design that still provides virtual heterogeneity through dynamic composition. Our paper makes the following contributions:

- We first analyze the LCA demands for the application domain of medical imaging. We then validate the system flexibility by targeting two independent application domains, namely navigation and computer vision. For all our benchmarks, we ex-

amine the potential for composability and accelerator sharing.
- We propose a hardware mechanism to control and compose ABBs that may be distributed anywhere in our system. This hardware provides virtualization of LCAs and allows software to interact with shared accelerators without the need to consider arbitration or contention.
- We further demonstrate how our approach provides load balancing to better distribute work among ABBs, thereby improving utilization for either a single application with multiple accelerator tasks or across multiple applications.

The rest of this paper is organized as follows: Section 2 provides analysis of our target domain and additional benchmarks. Section 3 details our approach to providing the flexible composition of ABBs into LCAs. Sections 4 and 5 outline our evaluation methodology and results. We compare our approach to prior work in Section 6 and conclude with Section 7.

2. THE CASE FOR COMPOSITION

In this section, we examine the characteristics of accelerator-rich architectures that will help motivate our design. Specifically, we show how LCAs from the medical imaging domain can be decomposed using ABBs. In Section 5.4 we will show how these ABBs are reusable in completely different domains.

2.1 Medical Imaging Application Domain

Medical imaging is an important tool for diagnosis and treatment, but is prohibitively time consuming for use in real-time clinical diagnostics. This need, combined with the fact that these applications feature a large amount of data parallelism and highly regular computation, make medical imaging an ideal candidate for accelerators. A more complete description of the applications in this domain, along with existing acceleration strategies, can be found in [5].

An interesting observation about all of these applications is that they can be decomposed into only 4 types of ABBs. These are listed in Table 1, along with the applications that utilize them.

Table 1: Accelerator Building Blocks (ABBs) used in medical imaging

ABBs	Denoise	Deblur	Registration	Segmentation
Float Reciprocal (FInv)	✓	✓		✓
Float Square-Root (FSqrt)	✓	✓	✓	✓
Float Polynomial-16 (Poly16)	✓	✓	✓	✓
Float Divide (FDiv)	✓	✓	✓	✓

2.2 Accelerator Demand

Rather than trying to provide sufficient LCAs for all possible execution scenarios, we propose providing a set of ABBs to use as a pool of building blocks to dynamically compose any desired LCAs. This idea is based on the following observations:

Overlapping Functionality - Table 1 emphasizes the considerable overlap in the functionality of ABBs and LCAs. We will leverage this commonality by using ABBs to construct LCAs on the fly.

LCA Utilization - Our experiments demonstrate that while LCAs have dramatic potential to improve performance and reduce power dissipation, their overall utilization is rather low. LCAs in the medical imaging domain, for instance, are utilized less than 10% of the time due to variations in accelerator resource demand. These relatively high idle times indicate an opportunity for accelerator sharing.

3. CHARM ARCHITECTURE

In this section, we address the design goals from Section 2 and propose a novel architecture that provides flexibility, scalability, and design reuse. Our Composable Heterogeneous Accelerator-Rich Microprocessor (CHARM) design includes both software and hardware components. This work is primarily focused on discussing the architectural contributions of CHARM, so discussion of the software component will be limited.

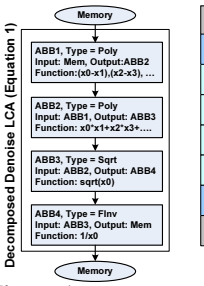

Figure 1: Composition

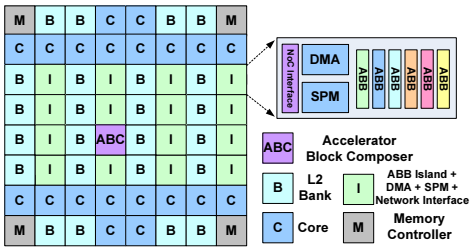

Figure 2: Architecture of CHARM

3.1 CHARM Software Infrastructure

CHARM's software component is responsible for: 1) LCA candidate selection - identifying program hotspots that would benefit from LCA implementation [13]; 2) ABB selection - generating a set of ABBs to cover a set of LCAs under physical design constraints (area, timing, power) [11] and 3) ABB flow graph creation - used to compose LCAs from ABBs. We have currently automated the second and third parts of the software component, and perform manual LCA candidate selection for now.

The structure of the ABB flow graph is the same as that of a task flow graph [14] (see Figure 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 SPM 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 1 illustrates this for one of the LCAs used in Denoise, whose functionality is formulated by Equation 1:

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

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

3.2 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 2 shows an example of the CHARM architecture. It consists of cores, L2 cache banks, memory controllers, ABB islands and an *accelerator block composer* (ABC), which is the means of con-

trol 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 3.2.1 for details on ABC TLB handling).

3.2.1 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.

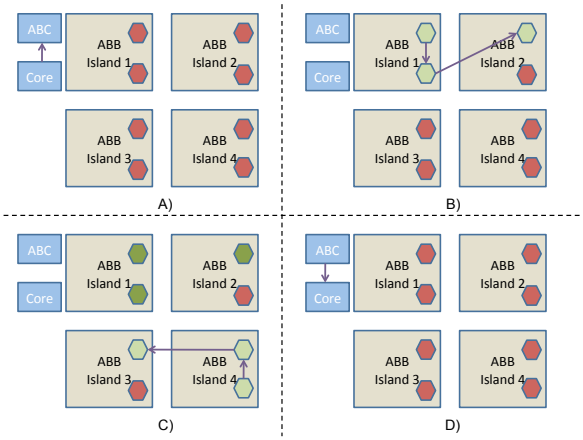

Figure 3: LCA composition example: A) A core sends a request for an LCA to the ABC; B) An LCA instance is allocated; C) An LCA instance is allocated with consideration for balancing DMA utilization; D) The ABC signals completion to the core.

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 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.

3.2.2 Example of Composition

Figure 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 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 3.2.1. After finding a match, consisting of two ABBs in Island 1 and a single ABB in Island 2 (Figure 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 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 3D).

4. EVALUATION METHODOLOGY

To evaluate the CHARM architecture, we have modified Simics [24] and GEMS [25] to model accelerator-rich many-core architectures. Table 2 shows the parameters used in our simulations.

Table 2: 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

We have also implemented a series of supporting tools to automatically generate accelerators as well as application code that makes use of these accelerators. The complete sequence of steps involved in adding accelerators and generating programs that use accelerators in CHARM is shown in [12]. 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 and Table 4 show the area and power overhead (using the Synopsys 32nm SAED library and CACTI 5.3 [1]) 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. To study the overhead of ABBs, we have synthesized the Poly16 ABB, the results of which are shown in Table 3. The internal structure of a Poly ABB is shown in Figure 4 (this Figure is actually showing a Poly8). It consists of Adder/Subtractor/Multiplier (ASM) modules, an SPM bank,

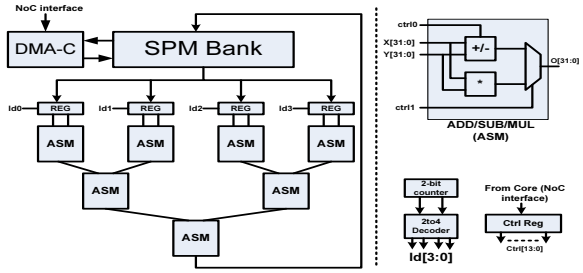

Figure 4: Poly ABB Details

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-C. For the 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. In [12] we have investigated sharing SPM between ABBs. Table 5 shows the area for the main components of the chip.

Table 3: 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

Table 4: 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
Registeration	3853098	183.9	18
EKF-SLAM	1188252	42.0	24
LPCIP	239159	6.11	6
SPM-2KB 2R,1R/W	37043	17.5	-

Table 5: Total area(mm^2)- CHARM-HW is the total of ABB islands and ABC

Core	NoC	Cache & Dir	CHARM-HW	CHARM-Total	LCA HW	LCA Total
10.8 Ref[2](scaled to 32nm)	0.3 Ref[21]	39.8 Ref[1]	8.3(Table3) (14%)	59.2	8.5(Table4) (14.3%)	59.4

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. 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 and the LCA area numbers in Table 4). 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.

5. RESULTS

We compare the following architectures to evaluate CHARM:

Physical LCA sharing with Global Accelerator Manager (LCA+GAM): In 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) [13] 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 [13].

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

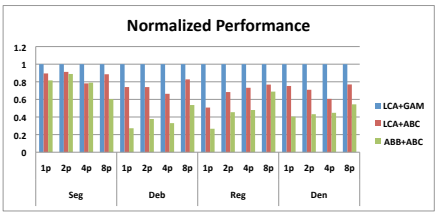

Figure 5: Performance improvement

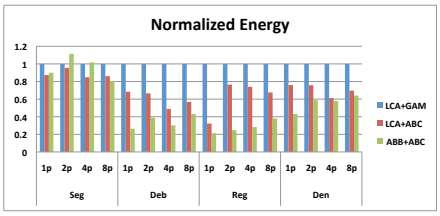

Figure 6: Energy improvement

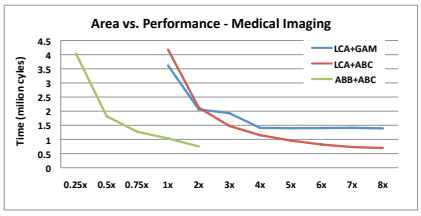

Figure 7: Effect of increasing accelerators

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 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.

5.1 Improvement over LCA-based systems

Figure 5 and Figure 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 (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.

5.2 Effect of adding accelerators

Figure 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 cases (LCA+GAM and LCA+ABC), the number of LCAs range from 1 to 8. For the ABC+ABB scheme, the quantity of ABBs range from 1/4 of the ABB number that area-wise matches 1 set of the LCAs in the domain, to 2 times that number.

There are several observations here. 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 memory 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.

5.3 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 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 8 and Figure 9, 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 8). When the task-grain is 128, only one set of ABBs is used by each thread. The values shown in Figure 9 describe the ABBs allocated for a single LCA instance per thread. Also shown in Figure 8 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.

5.4 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) [20] from computer vision and Extended Kalman Filter-based Simultaneous Localization and Mapping (EKF-SLAM) [22, 3] from navigation. A more detailed description of these two applications and existing acceleration strategies can be found in [8]. Figure 10 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.

6. RELATED WORK

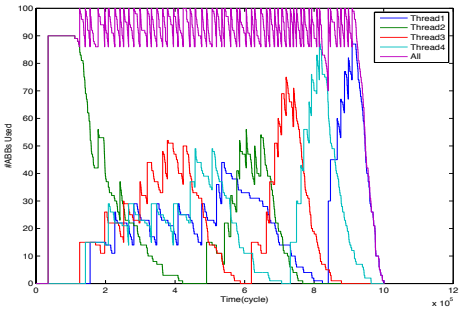

Figure 8: ABB utilization (Task-grain=8)

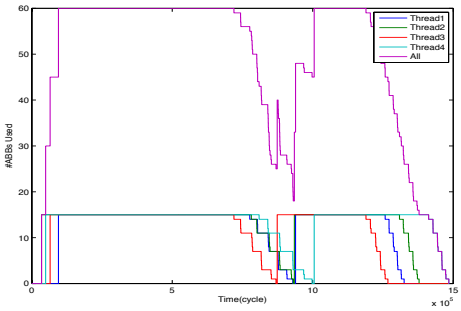

Figure 9: ABB utilization (Task-grain=128)

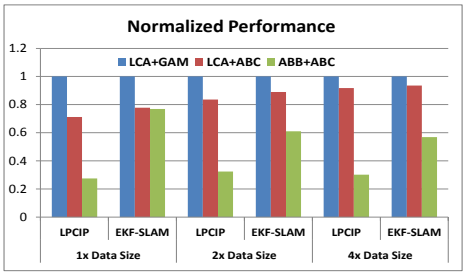

Figure 10: Performance improvement for computer vision and navigation

Some previous work has considered the on-chip integration of accelerators. Garp [17], UltraSPARC T2 [19], Intel's Larrabee [28] and IBM's WSP processor [15] are examples of this. Most of these platforms (except WSP) are tightly coupled with processor cores (or core-clusters). Our paper, on the other hand, focuses on loosely coupled accelerators that can be shared between multiple cores.

There have also been a number of recent designs of heterogeneous architectures, like EXOCHI [31], HiPPAI [29], SARC [27], and QsCores [30]. EXOCHI presents an architecture (exoskeleton sequencer) and a programming environment (C for heterogeneous integration) for a multi-core system. HiPPAI, like our work, eliminates system overhead involved in accessing accelerators through the use of a software layer. SARC also has a core and accelerator architecture similar to our work that relies on software for resource management. QsCores, although lacking dynamic accelerator management and load balancing, also relates to our work as it uses specialized cores to provide energy efficiency by exploiting similar code patterns within and across applications. We contrast with these software-based methodologies by advocating the use of hardware-based accelerator management.

Prior work has considered the composition of simple cores to form more complex general-purpose cores, as in core fusion [18], core spilling [7], and TRIPS [16]. However in those works, the composition is of coarser grain than ABBs, which allows for less flexibility in exploiting pipeline parallelism existing between ABBs. Also, in our approach there is no restriction on ABBs, which cannot be said for any of the above work. There is also some related work in accelerator virtualization, namely VEAL [6] and PPA [26]. VEAL [6] uses an architecture template for a loop accelerator and proposes a hybrid static-dynamic approach to map a given loop on that architecture. PPA [26] uses an array of PEs which can be reconfigured and programmed. It uses a technique called virtualized modulo scheduling which expands a given static schedule on available hardware resources. While for both of these works the target input is a schedule on a small nested loop, our approach is not limited in terms of either control flow or acceleratable problem size.

7. CONCLUSION

This work introduces CHARM - a design that provides scalability, flexibility, and design reuse in the space of accelerator-rich CMPs. CHARM features a hardware structure called the accelerator block composer (ABC), which dynamically composes accelerator building blocks (ABBs) into loosely coupled accelerators (LCAs), load balances tasks among the available ABBs, and includes a central TLB. Our experimental results show that on average we get a 2.1X speedup (best case 3.7X) running medical imaging benchmarks compared to approaches that use LCAs together with a hardware resource manager. We also gain in terms of energy consumption (on average 2.4X and best case 4.7X). These results are achieved with a platform that is flexible and easy to program, and which may be retargeted to a variety of domains. In [10], we have studied sharing SPMs in the NUCA for accelerator-rich CMPs.

8. ACKNOWLEDGEMENTS

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

9. REFERENCES

[1] Cacti 5.3: http://quid.hpl.hp.com:9081/cacti/.
[2] http://en.wikipedia.org/wiki/ultrasparc_iii.
[3] J.L. Blanco. Derivation and implementation of a full 6d ekf-based solution to bearing-range slam. Technical report, University of Malaga, Spain, Mar 08.
[4] Alex Bui et al. Customizable domain-specific computing. *Design and Test of Computers, IEEE*, Mar/Apr 2011.
[5] Alex Bui et al. Platform characterization for domain-specific computing. In *ASPDAC*, 2012.
[6] Nathan Clark, , et al. Veal: Virtualized execution accelerator for loops. ISCA '08.
[7] J. Cong et al. Accelerating sequential applications on cmps using core spilling. *Parallel and Distributed Systems, IEEE Trans. on*, pages 1094 –1107, 2007.
[8] J. Cong et al. Accelerating vision and navigation applications on a customizable platform. In *ASAP*, 2011.
[9] Jason Cong et al. Architecture support for accelerator-rich cmps. In *Proceedings of the 49th Annual Design Automation Conference (DAC 2012)*.
[10] Jason Cong et al. Bin: A buffer-in-nuca scheme for accelerator-rich cmps. In *ISLPED 2012)*.
[11] Jason Cong et al. A generalized control-flow-aware pattern recognition algorithm for behavioral synthesis. DATE '10.
[12] Jason Cong et al. UCLA computer science department technical report #120008.
[13] Jason Cong et al. AXR-CMP: Architecture support in accelerator-rich cmps. In *2nd Workshop on SoC Architecture, Accelerators and Workloads*, Feb 2011.
[14] Kayvon Fatahalian et al. Sequoia: programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006.
[15] H. Franke et al. Introduction to the wire-speed processor and architecture. *IBM Journal of Research and Development*, pages 3:1 –3:11, 2010.
[16] Mark Gebhart et al. An evaluation of the trips computer system. ASPLOS '09.
[17] J.R. Hauser and J. Wawrzynek. Garp: a MIPS processor with a reconfigurable coprocessor. FCCM'97, pages 12 –21, 1997.
[18] Engin Ipek et al. Core fusion: accommodating software diversity in chip multiprocessors. ISCA '07, pages 186–197.
[19] Tim Johnson and Umesh Nawathe. An 8-core, 64-thread, 64-bit power efficient sparc soc (niagara2). ISPD '07, pages 2–2, 2007.
[20] F. Jurie. A new log-polar mapping for space variant imaging : Application to face detection and tracking. *Pattern Recognition*, pages 865–875, 1999.
[21] A. B. Kahng et al. Orion 2.0: a fast and accurate noc power and area model for early-stage design space exploration. DATE '09, pages 423–428.
[22] R. E. Kalman. A new approach to linear filtering and prediction problems. *Journal of Basic Engineering*, pages 35–45, 1960.
[23] S. Li et al. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. MICRO '09.
[24] Peter S. Magnusson et al. Simics: A full system simulation platform. *Computer*, 35:50–58, 2002.
[25] M. Martin et al. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. In *Computer Architecture New, Sep 2005*.
[26] Hyunchul Park et al. Polymorphic pipeline array:a flexible multicore accelerator with virtualized execution for mobile multimedia application. MICRO 42.
[27] A. Ramirez et al. The sarc architecture. *Micro, IEEE*, pages 16 –29, 2010.
[28] Larry Seiler et al. Larrabee: A many-core x86 architecture for visual computing. *IEEE Micro*, 29:10–21, 2009.
[29] P.M. Stillwell et al. HiPPAI: High performance portable accelerator interface for SoCs. HiPC 2009, pages 109–118, 2009.
[30] G. Venkatesh et al. QsCores: trading dark silicon for scalable energy efficiency with quasi-specific cores. MICRO-44 '11, pages 163–174.
[31] Perry H. Wang et al. EXOCHI: architecture and programming environment for a heterogeneous multi-core multithreaded system. PLDI '07, pages 156–166.