

SChISM: Scalable Cache Incoherent Shared Memory

John H. Kelm, Daniel R. Johnson, Aqeel Mahesri, Steven S. Lumetta, Matthew Frank, and Sanjay Patel
University of Illinois at Urbana-Champaign
Urbana, IL 61801
{jkelm2, djohns53, mahesri, lumetta, mfrank, sjp}@illinois.edu

Abstract

This paper motivates and describes a class of accelerator architectures that manage cache coherence in software to exploit data sharing and communication characteristics present in emerging highly parallel workloads. Based on previous findings and our own studies, we show that our target applications have structure to their communication patterns that can be leveraged to move most cache coherence management into software. Replacing hardware cache coherence with software mechanisms allows die area to be reclaimed for more compute resources while also reducing hardware design complexity. Moreover, there is a prevalent programming style for large-scale parallel computation which can be mapped into a low-level task-based programming model that manages coherence in software without sacrificing usability and performance.

We also observe that the main benefit of hardware cache coherent systems is not for supporting data-parallel applications, but rather for implementing preemptive multitasking operating systems where migratory data and global locking constructs must be supported efficiently. To support the system software features that are required for applications running on an accelerator platform, we demonstrate an implementation of the Rigel Task Model. The Rigel Task Model is a low-level programming model that performs work distribution, task scheduling, software-enforced cache coherence, and synchronization in software, with limited specialized hardware, for an accelerator architecture without hardware cache coherence.

1 Introduction

Contemporary general-purpose chip multiprocessor (CMP) development is driven by the need to support multitasking operating systems, legacy code, and a broad spectrum of applications. The design goals of CMPs are distinctly different from those of accelerator platforms which have fewer requirements of system software, are less beholden to legacy, and are optimized for narrower classes of workloads and programming styles. One key architectural element that leads to a divergence between CMPs and accelerators is hardware cache coherence. In this paper, we show that hardware cache coherence provides limited benefit for emerging highly parallel applications that can be accelerated with accelerators. The limited benefit of hardware cache coherence for our target workloads leads us to investigate a class of architectures that eliminate coherence to reduce hardware design complexity and reclaim die area that can be used for providing more compute resources.

We make a distinction between the *application software* developed by the user and the *system software* that supports resource management. The distinction is made due to the different requirements each places on cache coherence and how they differ across conventional and accelerator platforms. The first contribution of this paper is a perspective on cache coherence: we observe that a coherent view of memory is used extensively to support system services on conventional CMPs while cache coherence is not required to support highly parallel applications on accelerator platforms. The second contribution of this paper is the Rigel Task Model (RTM), a mechanism for work distribution and synchronization where cache coherence and synchronization are controlled by software. The RTM demonstrates that the system software support required of accelerator architectures can be implemented without the use of hardware cache coherence.

Contemporary accelerator architectures have evolved into programmable multicore processors, with an order of magnitude more processing elements compared to CMPs, each with their own local memories, and with computational resources and caching systems that are tailored to particular classes of workloads [1, 2]. Motivating this work is the desire to provide a conventional view of memory for accelerators while also recognizing an opportunity to exploit the dissimilar constraints placed on caches for accelerators versus conventional CMPs. The programming trend motivating this work is the result of the wider acceptance of data-parallel accelerators as a computing platform. Graphics processing units (GPUs) and other accelerator architectures [3, 1] with tens and even hundreds of cores are becoming available. Emerging applications that are being demonstrated today can provide guidance to architects exploring future massively-parallel accelerators. These applications are scaling up to hundred-way and thousand-way parallel implementations on accelerator platforms such as those ported to GPUs [4]. We demonstrate that not only the structure of the applications should be studied and exploited, but also that the programming models in use for their development can be leveraged in defining the next generation of accelerator architectures.

There are three observations that we make about accelerator platforms that compel us to explore the replacement of hardware cache coherence management as found in CMPs with software-enforced cache coherence in accelerator architectures. Our first observation, presented in Section 2.1, is that the way in which application developers express this structure shares a common underlying form that can be leveraged to place coherence management in software without greatly altering the way in which accelerator applications are developed. Our second observation, presented in Section 2.2, is that emerging large-scale data-parallel applications have inherent structure to their synchronization and sharing that does not require hardware cache coherence. Lastly, we observe in Section 3 that the primary use of cache coherence is to support system services and is of little utility for the accelerator applications we evaluate.

Our observations lead us to study *Software-enforced Cache-coherent Accelerators* (SCA). SCAs provide mechanisms to allow software to flexibly enforce cache coherence and to provide efficient task management for accelerator workloads. In contrast to SCAs, CMPs rely upon cache coherence and global synchronization mechanisms to provide resource management. SCA architectures on the other hand rely on a weakly-consistent memory model, explicit local and global memory operations, and a task-based programming model to execute the coherence actions needed to enforce the memory model at barriers, thus providing structure without sacrificing performance. We describe one implementation of an SCA architecture that we are developing called Rigel in Section 4 and its task-based programming model in Section 5. Using detailed simulation, in Section 6 we evaluate the Rigel Task Model (RTM), a low-level task-based programming model for SCA architectures. The RTM allows cache

coherence hardware found in conventional systems to be replaced by a low-level programming model that supports an already prevalent style of parallel programming. We demonstrate that the RTM can enable highly parallel applications to be supported by flexible software constructs on an accelerator platform without cache coherence. Furthermore, we demonstrate that doing so does not require hardware cache coherence nor specialized task management mechanisms to achieve programmability and performance.

2 Motivating Trends

Our investigation of SCAs is motivated by trends we observe in the structure of emerging applications for accelerator architectures and the programming models in use today for developing these applications. The workloads we cite from other researchers and those that we evaluate independently are chosen for their relevance to the modeling and visualization of virtual and physical worlds which can provide an immersive, visually-rich computing experience. These applications are greatly enhanced by highly parallel accelerators and are representative of applications driving both existing and emerging consumer and high-performance computing markets.

2.1 Observation 1: Programming Trends

Our first observation is that programming styles adopted for developing highly parallel applications today share a common structure. The structure is similar to bulk synchronous processing and is independent of which high-level sharing model is used. One way to view the composition of large-scale parallel applications is as a collection of mostly-data-parallel units of work, which we refer to as *tasks*, executing concurrently while logically exchanging little or no data until a global synchronization barrier is reached. When the barrier is reached, modified shared data is made globally visible, and the next phase of computation begins. Tasks have no defined ordering with respect to other tasks between two barriers. Any data modified by a task can only be assumed by the programmer to be made globally visible after the next barrier. Any modified data that is shared between barriers must be annotated by the programmer explicitly. Tasks can be of fixed length, have uniform control flow, and access data with a very regular pattern or have highly variable lengths, contain divergent control flow, and possess arbitrary data access patterns. In a barrier-synchronized, mostly-data-parallel task-based programming model, features found in conventional CMP architectures such as hardware cache coherence are of marginal utility.

The organization of data sharing in parallel systems can be classified by the type of *high-level* programming model being employed. Examples include: message passing, distributed shared memory, and coherent shared memory. The choice of which data sharing model to use has historically been driven by system scalability issues, portability for legacy software and to future hardware generations, and perceived programmer effort. The common structure is a modified form of bulk-synchronous computation whereby periods of mostly parallel, possibly irregular, computation are interleaved with intervals where modified state is exchanged between execution streams separated by barriers. Whether the streams of computation use message passing, explicit globally-shared memory regions, or rely on hardware cache coherence as the *mechanism* to exchange values, the *sharing pattern* is common across sharing models.

A theoretical basis for reasoning about parallel systems using the bulk-synchronous parallel (BSP) model was described by Valiant [5]. BSP continues to be reflected in languages prevalent today including CUDA [6] from NVIDIA. CUDA is currently used to map data-parallel kernels to GPUs comprising hundreds of processing el-

ements. Conventional multi-core processors make use of programming annotations such as those provided by OpenMP [7] for conveying parallel regions to the compiler. BSP can be expressed using library and template-based runtime-supported models such as Intel’s Threaded Building Blocks [8] and STAPL [9] where high-level constructs are mapped to low-level system APIs running on cache coherent machines. Compiler and language extensions such as Split-C [10], Titanium [11], and Unified Parallel C [12] provide more examples of tools that allow for a bulk synchronous model to be expressed on distributed memory and cache coherent machines. Data-driven message passing applications are an alternative to purely data-parallel workloads. However, even these applications can have an underlying structure that is amenable to barrier-based synchronization as described in [13]. In Zheng et al. [13] the authors show that the inherent structure of large-scale applications can be leveraged for simulation of massively-parallel multiprocessors much in the way we find that it can be used for designing the actual systems and programming constructs.

We observe that many high-level programming models in use today for developing large-scale data-parallel applications do not necessarily depend on the flexibility provided by conventional systems. Furthermore, the common structure found in all of these parallel applications is rooted in the fact that the programmer attempted to create scalable code in a manner that is conceptually simple and thus, there is little sharing. The key insight is to realize that the common underlying structure to the data sharing can be exploited in the design of accelerator platforms regardless of which high-level programming models become widely-adopted in the future.

2.2 Observation 2: Application Trends

Our second observation is that emerging applications for systems incorporating accelerators have common data sharing and synchronization characteristics that can guide the design of future accelerator architectures. Previous work has investigated microarchitectural mechanisms for exploiting data-parallel applications [14, 15], but with the emergence of highly parallel accelerators, architectural and application-level tradeoffs that can leverage greater degrees of task-level parallelism for data-parallel workloads are of renewed interest. Examples of such workloads include the recognition, mining, and synthesis (RMS) [16] and physical simulation benchmarks [17] for providing more realistic virtual worlds that are being investigated within Intel. A variety of highly data-parallel workloads have been evaluated for conventional multi-core processors such as PARSEC [18] and ALPBench [19]. Accelerator workloads targeting current generation GPU computing have been studied [4] while studies motivating future accelerator architectures have focused on characterizing visual computing workloads [20].

2.2.1 Application Characteristics

The sharing patterns for a set of visual computing workloads are depicted in Figure 1 and Figure 2. Figure 1 shows the number of unique memory references that are shared across and between barriers per thousand instructions in a set of visual computing applications. Figure 2 breaks down the number of sharers for each benchmark into groups based on the number of sharers of writes (shared writes) and the number of sharers of reads that come from writes made prior to the last barrier (input reads).

Our independent study of visual computing applications shows similar data sharing and synchronization patterns, but we further investigate the sharing patterns of our workloads *across synchronization boundaries*. We

use a suite of data-parallel workloads that are representative of accelerator applications chosen from the visual computing space. Figure 1 shows the frequency of non-private loads and stores which are those that are produced by one task and consumed by one or more other tasks. Non-private accesses are broken down into whether the values are shared between synchronization boundaries, which we call write-shared data, or across barriers, which we call output data. The figure shows that the majority of non-private loads are to data produced before the last synchronization boundary. At the same time, both reads and writes to data shared between synchronization boundaries are rare. Finally, note that output writes which are stores that are loaded in other threads across barriers are more common than true shared writes which require inter-barrier synchronization; Moreover, they both constitute a very small fraction of overall execution.

The benchmarks presented in Hughes et al. [17] demonstrate that future architectures must accommodate variable task sizes to mitigate the impact of load imbalance. We further make the point that, although these workloads require a means to mitigate load imbalance due to variable task granularity, *they do not require migration of running tasks*; we will revisit this point in our evaluation of hardware cache coherence in the next section. Furthermore, there exist small parallel regions which are sensitive to parallelization overheads incurred for task creation, scheduling, communication of shared state, and synchronization upon task completion. Therefore developing a means to efficiently distribute work across the chip is of paramount importance.

There are five common characteristics we see in accelerator workloads. The first characteristic is the presence of large amounts of unmodified, read-shared data within a barrier interval as shown in Figure 1 as input reads. Examples of read-shared data from our workloads include scene and model descriptions or blocks of streaming media data. The second is that synchronization is coarse-grained and barrier-based which in turn motivates our investigation of bulk coherence management at task boundaries as opposed to the fine-grained synchronization provided by hardware cache coherence. Indicative of this pattern are output writes in Figure 1 which demonstrate that modified data is often read by another core *after* the next barrier. Characteristic three is that there exists only small amounts of write-shared data between barriers, shown in Figure 1 as shared reads and writes, indicating that tasks are highly data-parallel with few inter-barrier data dependences. The fourth characteristic is that fine-grained synchronization is present but is rare. An example where this is used is to atomically update shared data structures. Lastly we observe that when inter-barrier write-shared data does exist, it is usually between few sharers as is shown by the number of shared writes in Figure 2. These characteristics demonstrate that little coherence management is required between barriers and that there is the potential to push coherence management into software to be logically performed at barriers. At the same time, mechanisms must be present to allow small amounts of fine-grained synchronization and data sharing between barriers.

2.3 Insights

Our analysis provides three insights that we use to motivate SCA architectures: Our first is that very few writes are shared across parallel tasks between barriers. The common case is that writes are private to a task or, when shared, are output writes and thus need not be made globally visible until a barrier is reached. The pattern is common due to the programmer attempting to write scalable parallel code where a high degree of sharing would limit scalability due to the high cost of global communication. This insight motivates our investigation of software

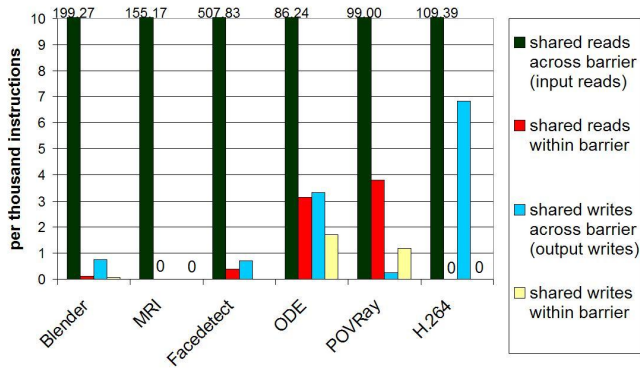


Figure 1. Read and write sharing between independent tasks for the our benchmark suite.

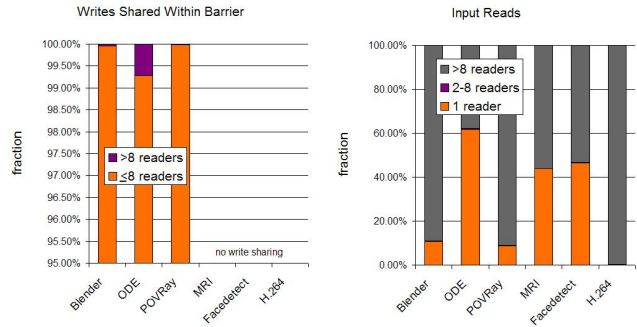


Figure 2. Number of sharers for each benchmark.

coherence management that takes place at barriers. Our second insight is that there are few shared writes that require some form of inter-core communication between barriers. This insight is the motivation for providing memory operations that can be made globally visible to allow for inter-cluster communication when necessary, but point-to-point messaging is not a prime design consideration. Our third insight is that the number of sharers is often limited with the vast majority being fewer than eight. The third insight leads us to investigate grouping tasks programatically and exploiting locality in SCA architectures by building clusters of cores that are able to exchange data between barriers with reduced latency.

3 Evaluating Cache Coherence

Cache coherence provides mechanisms to build efficient multitasking operating systems, but can have marginal utility for applications that have little or highly structured sharing. Hardware cache coherence can provide benefits to application software by shifting the burden of enforcing a coherent view of memory into hardware, but at the cost of additional hardware complexity and potential loss of optimization opportunities. In this section we explore the costs of cache coherence, past and current approaches to building scalable hardware-enforced cache coherence, and previous work related to software management of cache coherence.

3.1 Benefits of Hardware Cache Coherence

SCA architectures are motivated by the observation that the primary value of maintaining cache coherence in hardware is to support multiprogrammed operating systems that require migration of running tasks and efficient implementation of highly contended locks. Hardware cache coherence provides a mechanism to accelerate both of these use cases at the cost of added design complexity and area overhead. Cache coherence also provides a mechanism for distributing communication in a shared memory system via cache-to-cache communication, better balancing network load for applications with arbitrary sharing patterns. An additional benefit of cache coherence is its ability to aid in debugging by ensuring a coherent view of memory in hardware *at all times*. We now discuss these points in greater detail.

Cache coherence allows long-running, time-multiplexed tasks to be preempted and moved between processors by accelerating a pull-based model for migratory data. When a running task is relocated to another core, the working set may still be present in the local cache of the core where the task was previously resident. That data

is migrated to the new local cache as needed by hardware cache coherence. To support a similar model without hardware cache coherence would require potentially unnecessary flushing of data when a task is preempted. When tasks are not preempted or migratory, or when the latency of moving that data is not performance critical, hardware cache coherence provides little performance benefit since there is no unnecessary flushing to be performed.

Cache coherence provides mechanisms for broadcast notification and atomic modification which are used to build synchronization primitives for use by operating systems, run time systems, and low-level parallel libraries. Remote invalidation of data also has the benefit of enabling speculative prefetching of data into local caches prior to synchronization points. Coherence is often used for implementing efficient atomic operations, such as load-linked/store-conditional and compare-and-swap, but hardware cache coherence is not required. Hardware cache coherence allows communication of shared variables to be point-to-point between local caches instead of serialized through memory or higher-level cache resources, thus reducing latency and minimizing contention for global cache ports. While there are locks in the applications we study, they are few in number and are not highly contended. Furthermore, if there were highly contended locks used by accelerator applications, the applications would likely be unscalable as a result.

Shared memory systems with cache coherence can provide the appearance of a single global view of memory thus aiding programmability while still allowing for the performance benefits provided by distributed local caches. Coherence allows data to be transferred from one node to another transparently while leveraging a distributed communication network. The alternative is to provide a centralized location for data exchange, such as at a second-level cache, thus involving a third party when only two parties, the two local caches in this case, want to communicate. Another approach is to provide ISA-level point-to-point communication mechanisms which would then require data marshalling and explicit message passing that would not be required in the cache coherent case.

The debuggability of large parallel applications is a concern for future multiprocessors. Reducing the complexity of the abstract machine model used by the programmer can reduce the difficulty of tracking bugs by reducing the number of potential state variables that the programmer must reason about. Cache coherence, in combination with stronger memory consistency models, reduces the complexity of the abstract machine model by providing guarantees about what the value of a particular location is in time. Software can be used to provide similar guarantees, but at a potential performance cost, and may obscure the root cause of a software defect.

3.2 Hardware Cost of Cache Coherence

There are two costs to consider when evaluating the overhead of maintaining cache coherence in hardware: added die area for tracking the state of cached lines and additional interconnect traffic for coherence messages. The added area is the result of coherence state tracking hardware in snoop-based coherent designs or directories in directory-based designs. Both of these structures scale up with cache sizes. Moreover, directories can be difficult to provision appropriately and can become highly contended resources; Heinrich et al. [21] compares four directory protocols, demonstrating the difficulty in meeting the demands of a wide variety of applications and the trade-offs inherent in the design of scalable coherent machines. The network traffic generated in coherent machines is due to invalidation messages or updates when writes to shared data occur as well as request and acknowledgement messages between cores and the directory when write permission is obtained. If hardware coherence is removed,

die area from directories and the additional networking resources for coherence support could be reclaimed for more compute.

For highly parallel workloads, reclaiming area dedicated to non-compute resources such as cache, networking, or coherence management and reallocating that area to compute resources such as ALUs and floating-point units is a potential performance advantage. On the one hand, the common access pattern in data-parallel workloads consists of read and write data requests between the global caches/off-chip pins and the cores. On the other hand, invalidation and acknowledgements needed to implement hardware coherence rely upon broadcast operations and direct messages originating at one source, e.g. the directory, which are distinct communication patterns not otherwise present in data-parallel workloads. Any way to remove non-compute area and non-data traffic that reduces performance less than the area or bandwidth reclaimed is a performance and design complexity win for accelerator architectures.

3.3 Conventional Schemes

Snoop-based protocols [22] provide a simple mechanism for maintaining coherence between caches for a small number of processing elements. However, such schemes do not scale well beyond a small number of cores. Directory-based schemes [23] add complexity [24], but can scale to many more nodes by reducing the need for global broadcasts. However, even for scalable directory schemes, there is a cost in terms of area that could be repurposed for more compute. Hennessy et al. [25] provide a thorough discussion of the challenges faced in building scalable coherent architectures.

Limited directory schemes which track subsets of sharers have been employed to reduce the area overhead of full directories where each sharer is tracked by the directory. Chaiken et al. [26] survey a variety of directory techniques. In these schemes, the area overhead is greatly reduced relative to a full directory; however, a trade-off is made between directory die area and invalidation message bandwidth requirements. Furthermore there is still a need for control logic for directory controllers at each cache bank. Note that per-processor private caches and memory banks present in conventional directory schemes are replaced by local caches and on-chip global cache banks in the context of the multicore accelerators we discuss here.

As a working example, we will assume 256 local caches comprising 16 MiB of data that needs to be kept coherent. Each line is 32 bytes and there are 64 directories. For a scheme such as Dir1B [26], there would be at least eight bits tracked ($\log_2(N_{caches})$) per shared line resulting in a minimum distributed directory size of 512 KiB or 8 KiB per directory bank. However, this number assumes a uniform distribution of cached data across global cache banks. To avoid performance degradation due to load imbalance, a larger amount of directory space per bank must be allocated and the proper allocation per-bank is heavily workload dependent making provisioning difficult. The directories are further constrained in that they need to be organized like an inclusive L2 cache. Thus either the associativity of the directory needs to be unreasonably large (at least as large as the number of L1 caches) or one risks invalidation of useful cache data (and potentially thrashing) because of conflict replacements in the directory. Furthermore, such a scheme could result in a large number of broadcast invalidations that could potentially degrade network performance.

Benchmark	Task Length (K instructions)			Benchmark	Task Length (K instructions)		
	mean	20 th	1 st		mean	20 th	1 st
Blender	525	512	511	Open Dynamics Engine (ODE)	44.7	11.9	11.9
MRI	237	237	237	POVRay	88.3	25.3	4.34
Facedetect	405	320	286	H.264	77.2	75.8	75.8

Table 1. Mean, 1st, and 20th percentile task lengths for our benchmarks.

3.4 Current Proposals

In-network [27] and virtual circuit switching [28] coherence management are alternatives to directory-based schemes that exploit typical sharing patterns to reduce the cost of cache management in hardware. These mechanisms leverage networking resources to reduce the marginal cost of maintaining coherence by building groups of sharers or clusters of coherence on-the-fly. Doing so reduces the need for global broadcasts and distributes the cache state information to reduce serialization points and probe latency. Note that these schemes assume a network on a chip capable of point-to-point communication between processing elements and are geared toward general-purpose CMP workloads.

Region-based schemes [29] attempt to decrease the amount of state tracked per cache line by grouping together multiple contiguous cache lines and tracking them as a single region. Region-based designs can reduce traffic by generating a single cache probe that covers multiple lines instead of the single or even multiple probes needed for a coherence action in a conventional snoop or directory-based scheme. Even with the reduction provided by a region-based scheme, there is still a need to track coherence state at some granularity. Furthermore, region-based tracking may be less effective when smaller, more scattered data sets are shared across large arrays of cores. In such a case, the ability to form regions is reduced, minimizing the utility of a region-based coherence scheme.

3.5 Alternative Designs

Software managed coherence represents an alternative design point that places the burden of maintaining coherence on compilers, libraries, and runtime systems. Methods for enforcing coherence in software have been studied at both the fine and coarse granularities for networks of workstations and distributed shared memory machines. Coarse-granularity approaches [30] track coherence state at the page-level, on the order of 4-8 KiB, while fine-granularity approaches [31, 32] track coherence state at roughly the cache line level. A study comparing fine-grained to coarse-grained approaches [33] provides a good analysis of the relative trade-offs of the two approaches. Hill et al [34] present a hybrid approach where programmer annotations are used in conjunction with lightweight hardware. The hybrid approach in [34] offers a simple conceptual model for the programmer that uses flexible software when possible and lightweight hardware when necessary to achieve performance.

The lesson from previous software-coherence techniques is that the performance overhead is relative to the amount of sharing and the performance of an application is related to the correspondence between its granularity of sharing and that of the underlying software coherence scheme used. Another observation we make is that the programmability of these systems was not harmed by using a software coherence scheme. Shasta [32], for instance, instrumented the binary with coherence tracking instructions without programmer intervention. However,

it is possible to reduce the overhead of coherence, both hardware-managed and software-enforced, by restructuring applications to make them aware of the underlying coherence mechanism. One example of an optimization is demonstrated in [33] where the authors were able to leverage the cache-coherent SMPs within the larger system they built providing a hybrid software-hardware coherent multiprocessor system. The use of cluster-based architectures will be revisited in the context of the Rigel architecture and task model in future sections.

3.6 Future Scalable Coherence

Based on the characteristics of our workloads, we conjecture that the value of hardware-managed global cache coherence for our target applications is minimal. The real value of cache coherence is its ability to efficiently support multiprogrammed operating systems. While OS support is critical for general-purpose platforms, as accelerators, SCA architectures do not have the same system software requirements as conventional CMP platforms. Applications such as transactional databases with large amounts of unpredictable fine-grained sharing and microprocessors that must support general-purpose multiprogrammed workloads may be greatly aided by cache coherence, but are not the focus of SCA architectures.

While current efforts attempt to extend hardware coherence management for future CMPs with large numbers of cores, we see an opportunity to pursue another avenue using software to enforce coherence for accelerator architectures. Prior work using software-enforced coherence demonstrates that it is possible to achieve correct program behavior without placing undue burden on the programmer. Previous examples of software-enforced coherence did not gain widespread use, however, due to a shift toward small-scale cache-coherent SMPs and the adoption of message passing for programming systems of a larger scale. The situation today has shifted again as single-chip, single-address-space multiprocessors comprising hundreds and even thousands of cores with local caches become viable. By leveraging application characteristics and efficiently supporting the mechanisms historically reliant upon hardware cache coherence, future accelerator architectures have the potential to demonstrate high compute density without undue hardware complexity for the highly parallel workloads they support.

4 SCA Architectures

In this section we define a class of machines that we call *Software-enforced Cache-coherent Accelerator* (SCA) architectures. To more clearly articulate our observations and to demonstrate concrete examples of software-enforced coherence and task management, we introduce a particular implementation of an SCA architecture named Rigel which we are currently developing.

4.1 SCA Overview

An SCA architecture is a massively-parallel single-chip multiprocessor design targeted at applications that benefit from high compute density. The characteristics of an SCA design are that it avoids explicit hardware management of cache coherence and work allocation. Instead SCAs provide instruction set primitives for maintaining cache coherence in software, flexible memory operations, and atomic primitives for implementing software task management schemes. Using software mechanisms allows for a greater degree of flexibility and tunability over

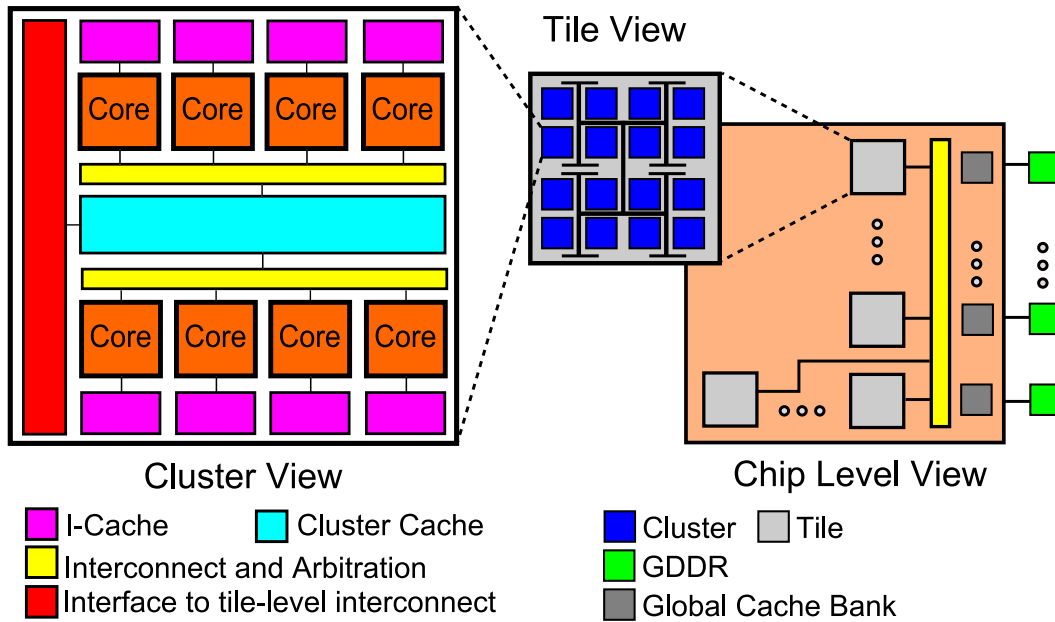


Figure 3. Diagram of the Rigel processor.

specialized hardware mechanisms while avoiding the cost of those hardware mechanisms being paid by applications where the use of specialized hardware is inappropriate. In such instances, hardware mechanisms are avoided or must be engineered around to achieve acceptable performances.

4.2 Rigel Architecture Overview

Rigel is an implementation of an SCA architecture for accelerating data-parallel workloads in the areas of computer vision, imaging, and physical simulation that scale up to thousands of concurrent tasks. The design goal of Rigel is to provide the highest compute density possible by minimizing per-core area. We improve density by removing features found in conventional designs that are of minimal benefit to the workloads targeted by Rigel. A block diagram of Rigel is shown in Figure 3.

The fundamental processing element of Rigel is an area-optimized dual-issue in-order processing core with a fully-pipelined single-precision floating-point unit and an independent fetch unit which executes a RISC-like instruction set. Eight cores are attached to a unified cache named the *cluster cache*. The cores, core-to-cluster-cache interconnect and the cluster-to-global interconnect logic comprise a single Rigel *cluster*. Clusters are connected and grouped logically into a tile. Tiles are distributed across the chip and are attached to global cache banks via a bi-directional tree-structured interconnect. The global caches provide buffering for multiple high-bandwidth memory controllers and can serve as a coherent view of memory. We believe that at 45nm a 400mm² chip attached to 16 GDDRs could contain up to 64 global cache banks and 16 tiles where each tile contains 16 clusters each with a cache and 8 cores.

4.3 Caching and Memory Model

All cores running on the Rigel processor share a single address space. The cores within a cluster have the same view of memory while global coherence is not explicitly maintained by the hardware across clusters. There are two forms of memory operations: *global* and *local*. Local memory operations constitute the majority of memory operations and support low-latency and high throughput for data accesses by applications code. Global operations allow for efficient system resource management and synchronization for a system without hardware cache coherence. Global memory operations on Rigel are not cached by the cluster cache and complete at the global caches which is the point of coherence. Memory locations operated on solely by global memory operations are kept consistent across the chip. Global memory operations are used to construct synchronization mechanisms. Global memory operations also enable fine-grained inter-cluster communication by way of the global caches. The cost of global memory operations is high relative to local operations due to the greater relative latency of accessing the global caches versus the local cluster caches. Furthermore, the achievable global memory operation throughput is limited by the number of global cache ports and on-chip global cache interconnect bandwidth.

Low-latency and high-bandwidth memory accesses are achieved using local memory operations. Local memory operations are cacheable at the cluster cache, but are not kept coherent between clusters by hardware. Local memory operations are used for accessing read-only data, private data, and data that is shared intra-cluster. Software must enforce cache consistency when inter-cluster read-write sharing exists.

Memory ordering on Rigel is defined separately for local and global memory operations. Local memory operations are processor consistent, as defined in [35], within a cluster. No ordering guarantees are provided inter-cluster for local operations. Global operations are processor consistent across the entire Rigel chip. Global memory operations are kept coherent across the chip with respect to other global memory operations by forcing all global memory operations to be complete at the global caches. The ordering between local and global operations from a single core can be enforced by using explicit memory barrier operations.

4.4 Coherence and Synchronization

Without hardware mechanisms to enforce coherence between cluster caches, alternative approaches must be used for conflicting shared data access. Write-shared data on Rigel could be kept coherent between sharers by forcing all modification to be made using global stores and all reads using global loads, however the cost of using only global memory operations would be high and strain global network and cache resources. One of the key motivations for SCA architectures is that low frequency of inter-core write-shared data *between* two consecutive barriers. Instead, most read-write sharing occurs *across* barriers and SCA architectures exploit this fact to hide or avoid long-latency global memory operations.

The sharing pattern present in our target workloads allows SCA architectures to leverage local caches for storing shared data between barriers and then lazily make modifications globally-visible. Lazy updates can be performed as long as coherence actions performed to write output data are complete before a barrier is reached. Rigel enables software management of cache coherence in two ways. One is by providing instructions for explicit cluster cache management that include cache flushes and invalidate operations at the granularity of both the line and the entire cache. Explicit cluster cache flushes update the value at the global cache, but do not modify nor invalidate copies

that may be cached by other clusters. The second is broadcast invalidation and broadcast update operations are provided to allow software to implement data synchronization and wakeup operations that rely on invalidation or update-based coherence in conventional cache coherent CMP designs. Broadcast operations will be revisited in the context of parallel reductions and synchronization barriers in the next section.

5 Programming the Rigel Architecture

In this section we describe the use and implementation of resource management, task support, and synchronization primitives that fall outside of the coarse-grained coherence mechanism used for application software. The Rigel Task Model (RTM) is a software implementation of a queue-based low-level programming model that enforces coherence in software and performs synchronization using barriers. While task management in conventional systems often relies upon cache coherence, the RTM demonstrates that by using software-enforced cache coherence, low-overhead task management is possible for SCA architectures. Higher-level constructs for producer-consumer parallelism and bulk-synchronous execution can be implemented based on the primitive operations exposed to the programmer through the Rigel Task Model API. Our baseline architecture provides instructions for local and global atomic operations, but does not explicitly provide support for task management in hardware.

In this section we present the API for RTM that is exposed to programmers. For most of this section a single queue is assumed; however, our model supports an arbitrary number of queues. Blocking dequeue operations, the ability to enqueue tasks between barriers, and implicit barriers at points when all cores are blocked at dequeue are assumed. We also describe the prototype implementation of the task model which we evaluate in the next section.

5.1 Software API

The software API for the Rigel Task Model is composed of basic operations for (1) managing the resources of queues located in memory and (2) inserting and removing units of work from those queues. Applications are written for the Rigel Task Model using a single-program multiple-data (SPMD) execution model where all cores share a single address space and application binary. The programmer defines parallel work units, which we refer to as *tasks*, that are inserted and removed from queues between barrier operations. The barriers thus provide a partial ordering of tasks. In the Rigel Task Model, barriers are used to synchronize the execution of all cores using a queue. Barriers also define a point at which all locally-cached non-private data modified during that interval must be made globally visible. Tasks that are inserted between two barriers should not be assumed to be ordered by the programmer and any inter-barrier write-shared data between barriers must be specified by the programmer.

5.1.1 Tasks

Each task is tracked by a *task descriptor*. We define a *task group* as a set of tasks that are guaranteed by the Rigel Task Model to execute on a single Rigel cluster. Tasks within a task group have the ability to communicate at a fine granularity due to the fact that they execute on the same cluster. Tasks in the same task group can use local memory operations and have those accesses be visible to other tasks within the task group. The number of tasks in a task group can be set by the programmer, but is nominally eight for the purposes of this work. Each task group is tracked using a *task group descriptor* consisting of pointers to task descriptors. Task groups execute with the

same scheduling guarantees as would be provided to eight ungrouped tasks. Enforcing concurrent execution of tasks within a task group is possible using cluster-level barrier operations inserted by the programmer.

5.1.2 Queue Management

The Rigel Task Model provides the following set of API calls to the programmer:

1. `TQ_Create`
2. `TQ_EnqueueGroup`
3. `TQ_Dequeue`
4. `TQ_Destroy`

Each queue created in the system has a logical queue ID associated with it. `TQ_Create` is called once for each queue generated in the system. The call to `TQ_Create` calls a software routine to allocate resources for the queue and make it available to all other cores in the system. Once a queue has been created, any core in the system can enqueue tasks on the queue or can attempt to dequeue tasks from the queue. Each basic enqueue and dequeue action operates on a single task descriptor. The `TQ_EnqueueGroup` operation provides a single operation to enqueue a DO-ALL-style parallel loop similar to the loop operation available in [36]. `TQ_Destroy` is called at the end of a computation to free the resources allocated to the task queue.

An initialized queue can be in one of four states: full, tasks-available, empty, and completed. A newly-initialized task queue, or more generally any initialized task queue without available tasks but not all cores blocking on dequeue, will be in the *empty* state waiting for tasks to be enqueued by one of the cores in the system. Any core that attempts a dequeue operation with an empty queue will block¹. When tasks are enqueued, the state of the queue becomes *tasks-available*. When tasks are available, dequeue operations return tasks without blocking. If cores are blocking on the task queue and the queue transitions to the tasks-available state, blocking cores are allocated newly available tasks and become unblocked. Tasks are removed in-order from the front of the queue, but may complete in any order between barriers. Should the queue exceed its defined size limit, the queue becomes *full* and any enqueue operation returns notifying the core attempting the enqueue of the queue's full status. It is the responsibility of the programmer to ensure that a consumer core is always available to dequeue tasks thus ensuring forward progress and avoiding deadlock.

The *completed* state is used to provide an implicit barrier in the Rigel Task Model and requires special consideration. When all cores participating in a barrier interval have completed all tasks in the system, they will all begin blocking on the task queue and the task queue will transition into the completed state. When the completed state is reached, a barrier is executed and all cores are returned a notification that a barrier has been reached. The semantics of the completed state are such that work can be generated between barriers and is not constrained to only occur at the start of an interval. An example of where this may be useful is in the traversal of a tree structure where sibling subtrees can be processed in parallel, but the number of tasks generated is not known a priori.

		Atomic (L)	Atomic (G)	Memory (G)	Notes
Group Enqueue		-	1 AI, 2 XC	2 GLD	9 LF
Dequeue (L)		1 LL, 1 SC	-	-	-
Dequeue (G)		3 LL, 3 SC	1 AI	2 GLD	-
Barrier (L)	W/U	-	-	-	
	ENT	1 LL, 1 SC	-	-	-
Barrier (G)	W/U	-	-	-	LO
	ENT	-	1 AI	2 GLD	-

Table 2. Special Memory Instruction Counts for Queue Operations. Notes: For barriers, W/U refers to the operations to unblock a blocked core. Global wake-up also wakes up the other cores of a cluster blocking on the queue. ENT refers to the time a core spends entering the barrier if a task is not available. AI: Global Atomic Increment, XC: Global Atomic Exchange, GLD: Global Load, GST: Global Store, LF: Cluster Cache Line Flush, LL: Local Load-linked, SC: Local Store Conditional, LO: Local Memory Operations Only

5.2 Implementation

In this section we describe the Rigel Queuing System (RQS), a two-level hierarchical task queuing system running on the Rigel Architecture that implements the Rigel Task Model. Note that the implementation details here allow for a task queuing system to be built *without the use of global cache coherence but rather through the use of memory operations that bypass possibly incoherent local caches*. We refer to the memory operations that bypass the cluster caches as *global operations*. We discuss the operations provided by the architecture that enable the task model and the rationale for the design choices made in our implementation. For the following sections, we will reference Table 2 which provides a listing of the number of global operations that are used in constructing the various pieces that make up each primitive operation.

The RQS is highly tunable with many parameters that can be adjusted statically based on known application characteristics or dynamically based on runtime conditions. Furthermore, different implementations could be tailored for different classes of workloads. Here our goal is to demonstrate a proof of principle implementation of a task-based programming model built on top of an SCA architecture; we leave the study of alternate implementations, tuning, and optimizations to future work.

5.2.1 Global Task Queue

The baseline design of the RQS provides a single global task queue (GTQ) for every logical task queue instantiated by the programmer using a `TQ_Create` API call. The GTQ is operated upon using global operations and atomic operations that complete at the global cache only. No part of the GTQ is cached at the cluster cache. The GTQ is organized as an indexed array of task group descriptors with a secondary structure holding the task descriptors pointed to by the task group descriptors. A GTQ tail index and head index are used to insert and remove entries from the queue. A lock protects enqueue operations. No locking is necessary for GTQ dequeue operations.

An enqueue is performed by acquiring the enqueue lock for the queue. The lock is implemented using atomic exchanges. The next step is to check for an overflow condition and if an overflow will not occur, task group

¹Non-blocking calls are possible, but are not considered here to avoid unnecessary complexity in describing the Rigel Task Model.

descriptor(s) and related task descriptor(s) are copied to the GTQ data structures. Once the copies are complete and made globally visible, i.e., flushed out of the enqueueing core's cluster cache, the tail index is incremented. Incrementing the tail index makes the new tasks globally visible.

A GTQ dequeue operation is performed by incrementing the head index, using an atomic increment, and comparing it against the tail index read using a global load. Note that atomic increments take place at the global caches and thus can be pipelined through the network, allowing up to one dequeue at a global cache bank per cycle. If the head index is less than the tail index modulus the size of the queue, the returned head index represents a valid task group descriptor. In this situation, the core performing the dequeue now owns the task group and associated tasks which it can then add to its local task queue (LTQ).

If the head index is not less than the tail index, there are no available tasks in the global task queue and the dequeuing core begins blocking on what may be a barrier. The incremented head index that the dequeuing core has is a reservation for a task group; In the meantime, another core may enqueue tasks thus incrementing the tail index and making the head index held by a waiting core valid. A barrier is reached when a core finds that its incremented head index is equal to the tail index plus the number of cores that should be waiting on the barrier modulo the number of tasks in the queue. The barrier implementation is discussed further below.

5.2.2 Local Task Queues

A key issue for SCA architectures is finding ways to exploit locality to minimize global communication and synchronization. The cluster abstraction provided in Rigel offers a natural partitioning for inserting a level of hierarchy into the task queuing system that reduces the bandwidth requirements of the global cache ports and interconnect. Furthermore, by bringing groups of tasks into the cluster cache using a single transaction with the GTQ, the system is better able to amortize the latency incurred by global operations over more useful work being completed by the cores.

Local dequeues are performed by acquiring a cluster-level lock, checking for local tasks, returning a task to the application if one is present or attempting to get a new task group from the GTQ when none are present locally. The LTQ is implemented as a linked list of task descriptors. To ensure forward progress and to reduce the traffic on the cluster cache bus, we implement a ticket lock at the cluster level which allows a core to get a reservation and poll locally on the current ticket value. Once the core has the LTQ lock, it attempts to dequeue a task locally. If there is an available task, the local lock is dropped and the new task is returned to the core. If no task is found, the core attempts to execute a global dequeue if no other global dequeue from the cluster is already pending. While attempting a global dequeue, the local lock is not held to allow for overlapping local dequeues while global dequeues are filled into the LTQ. During the dequeue process, the core also checks to see if a barrier has been reached. If a barrier has been reached, the core returns to the application a notification that the barrier has been reached.

A clustered implementation of an SCA architecture such as Rigel can amortize the cost of GTQ dequeues by fetching multiple tasks and bringing them into the local task queue with one transaction. However, it is difficult to reduce the cost of local dequeues beyond a certain threshold without added hardware support. Having support for automatic prefetching for as little as a single task, as was done in Carbon [36], provides a large benefit for

fine-grained tasks as it allows the fetching of the next task to be overlapped with the execution of the current task. If the target length for tasks and the relative overhead which that length allows exceeds the time for software to dequeue tasks from the local task queue, we may consider providing local task queue acceleration in hardware. For the task lengths we observe in our target workloads, which are on the order of many thousands of cycles as shown in Table 1, the benefit of special-purpose hardware is less clear.

5.2.3 Barrier and Parallel Reduction Implementation

Barriers, and more generally reductions, are frequent global operations on SCA architectures. As tasks become shorter and parallel slackness is reduced, the cost of barrier operations can become a limiting factor for parallel scalability. For example, in our conjugate gradient linear solver benchmark, the cost of doing the two reductions per time step increases from a negligible amount at eight cores to almost 20% for a 128-core configuration. Parallel reductions consist of every node participating in the reduction contributing to a single value. A barrier is a special case of the reduction operation where the contribution is simply incrementing a counter with the added step of all participating nodes being notified when the last node enters the barrier.

An efficient reduction can be built using a multi-level tree. A tree structure is necessary on a SCA such as Rigel because it can reduce port contention at the global cache bank hosting the reduction value, minimize global communication, and reduce the latency experienced by contributing nodes. One issue that arises from the lack of hardware cache coherence is the inability to easily notify all participating nodes when a barrier is reached. Commonly, some variant of test-and-set can be employed by all nodes and thus only the last node to join the barrier generates invalidation traffic notifying all polling cores that the barrier has been reached. Even this is an imperfect solution as it forces all waiting cores to fetch the now invalidated cache entry from memory. In an SCA architecture, the problem is made worse by the fact that all test operations must propagate to global caches thus flooding the network and creating contention for global cache ports which may starve remaining tasks attempting to do useful work. In an effort to mitigate this issue we have added a broadcast update and a broadcast invalidate instruction to the Rigel ISA which allows for words to be globally invalidated or updated, albeit at some cost, by any core in the system. A more in-depth discussion is beyond the scope of this paper and is left to future work.

5.3 Debugging and Optimization

Finding and removing software defects in large-scale parallel programs can be a daunting task. Placing the onus of coherence management on software and into the hands of the programmer can greatly exacerbate the already difficult task of detecting, reproducing, and eliminating bugs in parallel programs. To mitigate the debugging issues that might occur when using the RTM, one can apply an iterative refinement approach. The approach entails software evolving from a known-working design mapped to the RTM API where tasks are forced to execute in enqueue order to a correct, parallelized version using a standard implementation of the RTM such as the one described in this work. Global operations would be used for all data accesses thus bypassing local caches completely. By avoiding local memory operations, the system is kept trivially coherent at the cost of performance while an application is under development.

Parameter	Value	Units	Parameter	Value	Units	Parameter	Value	Units
Clusters	16	–	Cluster Cache			Global Cache		
Cores per Cluster	8	–	Size	64	KiB	Size	512	KiB
Total Cores	128	cores	Associativity	8	ways	Associativity	8	ways
Pipeline Configuration			Banks	1	–	Banks	4	–
Width	2-wide In-order		MSHRs per Bank	16	MSHRs	MSHRs per Bank	16	MSHRs
Stages	6	–	Ports per Bank	1	ports	Ports per Bank	1	ports
FP/MEM/INT Unit	1 each	–	Line Size	32	bytes	Line Size	32	bytes
Core Caches			Access Latency	4	cycles	Access Latency	2	cycles
Size	4096(I) 32(D)	bytes	Cluster Cache Interconnect			Global Cache Interconnect		
Associativity	2(I) 1(D)	ways	Topology	Shared Bus		Topology	Tree	
Banks	1	–	Latency	2	cycles	Latency	2, 2, 4	cycles
Latency	1	cycle	Bus Width	256	bits	Data Width	256	bits

Table 3. RigelSim configuration for one tile.

Once a working parallel design is achieved in a mode where coherence is not an issue, the developer or potentially the compiler can begin to improve performance by replacing global operations with local memory operations. Known-private data can be accessed using local memory operations without any further action. Accesses shared across barriers can be made local by inserting the proper cache management API calls to enforce coherence in software at task boundaries. The application development road map for RTM-based codes can be extended to optimization steps. Programmers concerned with achieving maximum performance can strategically break through the RTM API to gain greater performance when required, which is a feature that may not be possible in systems where the task management and all coherence actions are controlled by hardware mechanisms. One example where this technique may be useful is to allow software-controlled prefetching of data before the completion of a barrier. Without programmer intervention prefetching across barriers may not be possible in the baseline RTM as updates from one barrier interval are not guaranteed to be valid for a consumer to prefetch until after the barrier has completed; however, if the data is known to be globally-visible or the data is read-only the optimization can be correctly applied. Further investigation of debugging and optimization of the RTM and for SCAs in general is left to future work.

6 Evaluation

In this section we evaluate an implementation of the Rigel Task Model running on an execution-driven simulator modeling the Rigel processor. We use both synthetic benchmarks and a set of optimized parallel kernels for the evaluation. We show the costs associated with task enqueue and dequeue, synchronization operations, along with costs borne by explicit cache management instructions inserted to maintain coherence when needed.

6.1 Methodology

To evaluate programming an SCA we use RigelSim, an execution-driven simulator that models the Rigel memory model, cache hierarchy, tile-level interconnect between cluster caches and global caches, and uses a cycle-accurate model of GDDR4 memory. Rigel is intended to have hundreds of clusters and thus thousands of cores. Due to the intractability of simulating a full Rigel chip, we use a fractional configuration in our studies of up to

16 clusters which results in a 128-core system being simulated. Table 3 lists the relevant parameters used in our simulations.

We generate code using the LLVM 2.1 compiler with a backend target that we are developing for the Rigel ISA. All library and system calls used by the benchmarks are simulated and their costs are incorporated into the measured results. The benchmarks presented were written using the RTM API described in the previous section. The API is implemented by a library written in a mix of C and Rigel assembly that makes use of the coherence and memory management instructions provided by the Rigel ISA. We evaluate five optimized parallel benchmarks natively on Rigel. The conjugate gradient solver (CG), Sobel edge detect (edge), and dense matrix multiply (DMM) benchmarks were written by hand and optimized for the Rigel architecture. Heat is adopted from the Cilk benchmark with optimizations applied for Rigel. MRI is a port of the VISBench [20] medical imaging benchmark to Rigel.

6.2 Anatomy of a Task

We use synthetic task benchmarks to evaluate the relative costs of each component of a task in the RTM shown in Figure 4. The figure shows tasks with lengths of 1K, 4K, and 16K cycles running on 1, 4, and 16 cluster Rigel systems corresponding to 8, 32, and 128 core configurations, respectively. The costs include the time spent acquiring global task groups and inserting tasks into the LTQ, the local dequeue operation for each task, the time spent waiting at the local barrier while one core attempts to fetch new tasks from the GTQ, the time spent waiting on the global barrier when the GTQ is empty, and the cost of enqueueing tasks. The total amount of work is kept constant at one billion core cycles for all runs.

Figure 4 shows that overheads for task sizes of 4K and 16K cycles are 6.5% or less for all configurations. Task prefetch via task groups and LTQs provides fast local dequeue and amortizes more expensive accesses to the GTQ where there are potentially more accessors to contend with and longer latency. In contrast, the relative overheads for short 1K cycle tasks are large, from 24.8% to 310.9%. For short tasks, the system becomes enqueue-limited: cores complete tasks faster than new tasks can be supplied to the queue, resulting in excess idle time as cores attempt to dequeue from empty queues. This effect is exacerbated as core count and thus rate of task dequeue increases. The impact of shorter 1K cycle tasks can also be seen in the amount of time spent in local dequeue on 16 clusters. A significant portion of the overhead corresponds to local dequeue. Local task prefetch via task groups is not successfully overlapped with task execution due to increased GTQ latency caused by contention; this leaves many cores waiting at their LTQ for work to arrive. If the system were not enqueue-limited and large numbers of small tasks were available in the GTQ, overheads could be reduced by increasing task group size and thus prefetching more tasks into the LTQ, potentially at the expense of load balancing. The high overheads for 1K cycle tasks are not fundamental to all software task queues and could be reduced or hidden with appropriate tuning and optimization.

Table 1 shows that even at the 1st percentile, task lengths are still thousands of cycles. While regions of fine-grained tasks do exist, the table shows that support for extremely fine-grained tasks on the order of 1K instructions is not necessary for current parallel applications that have been demonstrated to scale beyond one thousand cores. With the Rigel Task Model, we have demonstrated efficient task management for tasks as small as 4K cycles,

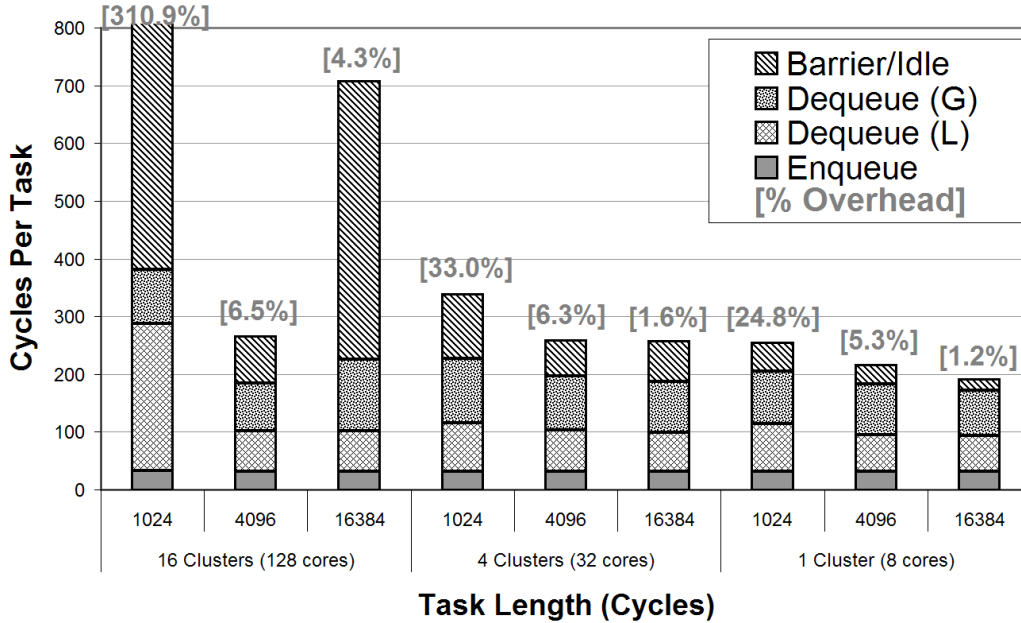


Figure 4. Overhead of software task queues for a fixed amount of work.

orders of magnitude smaller than those found in our benchmark suite.

For full-tile configurations of sixteen clusters, the absolute overhead cost of a task in cycles increases on 16K cycle tasks versus 4K cycle tasks, however, the cost as a percentage of the task length decreases. The increase in absolute cost is partly due to load imbalance. With the amount of work constant, longer tasks result in fewer tasks available and greater opportunity for imbalance. Local task prefetching can increase load balance toward the end of execution when tasks begin to run out in the GTQ. In this case, both increased idle time and waiting at the barrier for tasks are both a result of load imbalance.

6.3 Task Model Evaluation

To understand the performance and scalability of the software task model and underlying software coherence mechanisms we have written and optimized a set of kernels to run on Rigel. In Figure 5 we show the performance results for our five kernels running on 1, 4, and 16 clusters. We have implemented a hardware task queuing system for Rigel that is very similar to optimal Carbon [36] to serve as our baseline (HW-OPT). The HW-OPT configuration should be viewed as an idealized limit for a hardware task queuing system. There is no charge paid for coherence actions in the HW-OPT configuration, enqueues and dequeues require a single instruction and execute in 20 processor cycles, and the wakeup from a barrier after all cores have begun blocking on a completed task queue requires 100 cycles. Kumar et al. [36] provides a more in-depth analysis of the costs and performance sensitivities of these parameters. The HW-OPT configuration does not generate task groups. In our second configuration (SWTQ-HWCC) we evaluate the performance of our benchmarks using software task queues without inserting explicit cache flushes, invalidations, global loads, and global stores. Instead, idealized hardware cache coherence is assumed. The third configuration (RTM) simulates an implementation of the Rigel Task Model with cache coherence and task management fully handled by software.

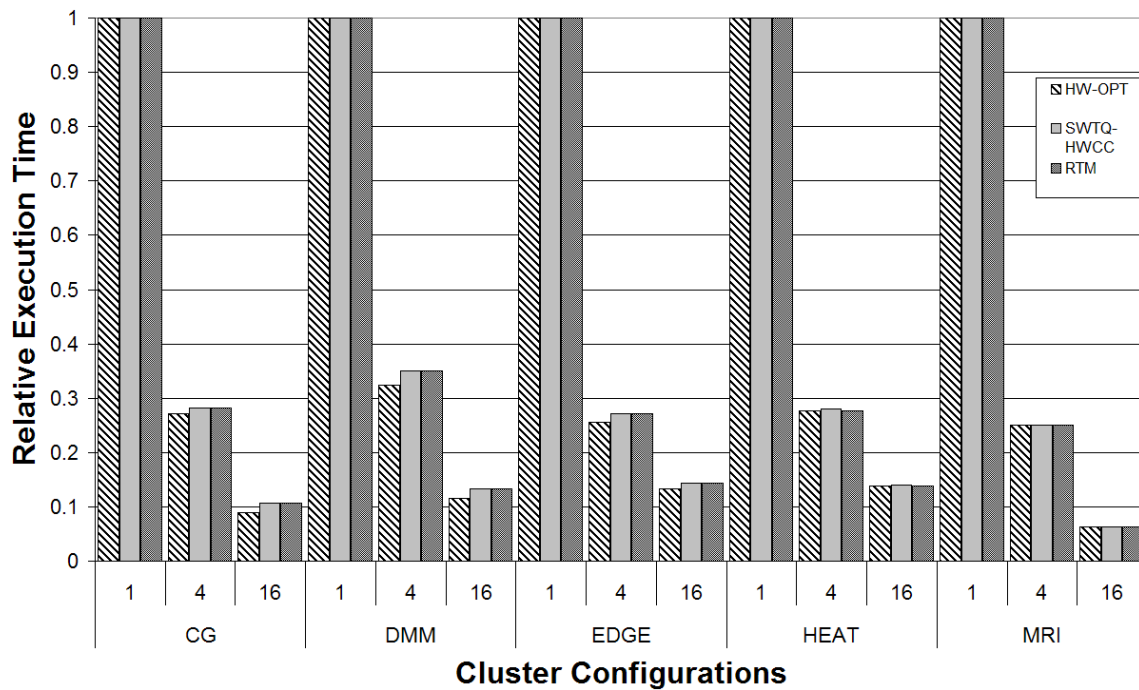


Figure 5. Performance results for 1, 4, and 16 cluster configurations. HW-OPT: Hardware task queues with idealized cache coherence SWTQ-HWCC: Software task queues with hardware cache coherence RTM: Rigel Task Model with software task queues and software coherence

Comparing the HW-OPT and SWTQ-HWCC configurations demonstrates the relative cost of software and hardware task queues. Comparing the SWTQ-HWCC and RTM configurations allows us to understand the cost of implementing coherence actions such as flushes and invalidates in software. Figure 5 shows that both SWTQ-HWCC and the RTM achieve performance approaching HW-OPT. Hardware task queues provide minor performance benefits for each benchmark by reducing the marginal cost of task dequeue and, due to their idealistic implementation, fix the cost of barriers and enqueue operations which impacts scalability for fine-grained tasks. For our workloads, such benefits are small and likely outweighed by the relative flexibility of the software model and potential complexity and area reductions of using software mechanisms versus dedicated hardware. A possible exception, as shown earlier in Figure 4, is when task granularity is extremely small, on the order of 1000 cycles.

For applications such as CG, which has shorter task lengths and more barriers, performance scales less effectively with software task queues. As the number of cores increases, the amount of work to be performed between barriers on a per-core basis decreases. Barriers are more expensive in the software model than in HW-OPT, reducing scaling potential for applications with frequent barriers or small amounts of work between barriers.

For the kernels studied, the cost of required software coherence actions are small. Performance for the SWTQ-HWCC and RTM configurations are nearly identical for all five kernels. The low cost can in part be attributed to the relatively few required coherence actions for our class of applications as well as our ability to perform coherence updates lazily between barriers. Explicit flushes and invalidates in software executed as part of the RTM can even provide modest performance improvements due to better caching behavior, mitigating the cost of performing the operations themselves. Such performance improvements can be observed in both DMM and heat for large numbers of cores. None of our benchmarks exhibit degraded performance when coherence actions are performed in software as opposed to hardware.

Although our selection of kernels performs well with the software task queues and the Rigel Task Model, this may not be the case for all types of applications. In particular, applications with frequent barriers, fine-grained synchronization or data sharing, or extremely short tasks may perform poorly with the RTM implementation as described. Applications with these characteristics may require different optimizations and configurations than those discussed here and are left to future work.

7 Conclusion

Based on the observations and data presented in this paper, we advocate for a class of architectures that eschew cache coherence. We observe that: emerging applications for massively-parallel processors have data sharing patterns that motivate the investigation of alternatives to conventional hardware-managed cache coherence; cache coherence greatly aids the development of general-purpose multiprogrammed operating systems but highly parallel accelerator architectures require a subset of services that can be implemented without cache coherence; and the prevalent programming models used to develop these applications today share a common bulk-synchronous structure that can be exploited by software coherence management. These facts are used to justify Software-enforced Cache-coherent Accelerators (SCA) as an alternative to more conventional CMPs with coherence between private caches enforced by hardware.

SCA architectures are tailored to relevant workloads in lieu of complete generality to allow for more area efficient and scalable designs compared to current general-purpose CMPs. We describe the Rigel Task Model

to illustrate how the prevailing programming style for massively-parallel systems can be exploited by an SCA architecture. We evaluate the Rigel Task Model using execution-driven simulation of an SCA system we are developing called Rigel. We show that even without cache coherence we can support a flexible task management API for developers to target with little impact on performance.

8 Acknowledgments

The authors acknowledge the support of the Focus Center for Circuit & System Solutions (C2S2) and the GigaScale Research Center (GSRC), two of five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation Program. Mr. John Kelm is funded by an ATI Graduate Research Fellowship.

References

- [1] Larry Seiler et al., “Larrabee: a many-core x86 architecture for visual computing,” *ACM Trans. Graph.*, vol. 27, no. 3, 2008.
- [2] NVIDIA, “NVIDIA GeForce 8800 GPU architecture overview,” November 2006.
- [3] Michael Butts et al., “A structural object programming model, architecture, chip and tools for reconfigurable computing,” in *FCCM’07*.
- [4] Shane Ryou et al., “Optimization principles and application performance evaluation of a multithreaded GPU using CUDA,” in *PPoPP’08*, 2008.
- [5] L. G. Valiant, “A bridging model for parallel computation,” *Communications of the ACM*, vol. 33, no. 8, 1990.
- [6] J. Nickolls et al., “Scalable parallel programming with CUDA,” *Queue*, vol. 6, no. 2, 2008.
- [7] OpenMP Architecture Review Board, “OpenMP application program interface,” May 2008.
- [8] J. Reinders, *Intel Threading Building Blocks*. 2007.
- [9] N. Thomas et al., “A framework for adaptive algorithm selection in STAPL,” in *PPoPP’05*, 2005.
- [10] A. Krishnamurthy et al., “Parallel programming in split-C,” in *SC’93*, 1993.
- [11] A. Kamil et al., “Making sequential consistency practical in Titanium,” in *SC’05*, 2005.
- [12] W. Carlson et al., “Introduction to UPC and language specification,” Tech. Rep. CCS-TR-99-157, IDA Center for Comp. Sci., 1999.
- [13] G. Zheng et al., “BigSim: A parallel simulator for performance prediction of extremely large parallel machines,” *IPDPS*, 2004.

- [14] Karthikeyan Sankaralingam et al., “Universal mechanisms for data-parallel architectures,” in *MICRO’03*, 2003.
- [15] Owens, J.D. et al., “Media processing applications on the imagine stream processor,” *ICCD’02*, 2002.
- [16] P. Dubey, “Recognition, mining and synthesis moves computers to the era of tera,” *Technology Intel Magazine*, February 2005.
- [17] Christopher J. Hughes et al., “Physical simulation for animation and visual effects: parallelization and characterization for chip multiprocessors,” in *ISCA’07*, 2007.
- [18] Christian Bienia et al., “The PARSEC benchmark suite: Characterization and architectural implications,” Tech. Rep. TR-81108, Princeton University, January 2008.
- [19] Man-Lap Li et al., “The ALPBench benchmark suite for complex multimedia applications,” *IWCS’05*, Oct. 2005.
- [20] Aqeel Mahesri et al., “Tradeoffs in designing accelerator architectures for visual computing,” Tech. Rep. UILU-ENG-08-2208, University of Illinois, May 2008.
- [21] Mark Heinrich et al., “A quantitative analysis of the performance and scalability of distributed shared memory cache coherence protocols,” *IEEE Transactions on Computers*, vol. 48, no. 2, 1999.
- [22] J. R. Goodman, “Using cache memory to reduce processor-memory traffic,” in *ISCA’83*, (Los Alamitos, CA, USA), 1983.
- [23] D. Lenoski et al., “The directory-based cache coherence protocol for the dash multiprocessor,” in *ISCA’90*, 1990.
- [24] J. D. Kubiatowicz, “Closing the window of vulnerability in multiphase memory transactions,” in *ASPLOS’92*, 1992.
- [25] John Hennessy et al., “Cache-coherent distributed shared memory: perspectives on its development and future challenges,” *Proceedings of the IEEE*, vol. 87, pp. 418–429, Mar 1999.
- [26] D. Chaiken et al., “Directory-based cache coherence in large-scale multiprocessors,” *Computer*, vol. 23, no. 6, 1990.
- [27] N. Easley et al., “In-network cache coherence,” in *MICRO’06*, 2006.
- [28] N. Enright Jerger et al., “Virtual circuit tree multicasting: A case for hardware multicast support,” in *ISCA’08*, June 2008.
- [29] J. F. Cantin, J. E. Smith, M. H. Lipasti, A. Moshovos, and B. Falsafi, “Coarse-grain coherence tracking: Region scout and region coherence arrays,” *IEEE Micro*, vol. 26, no. 1, 2006.

- [30] Cristiana Amza et al., “Treadmarks: Shared memory computing on networks of workstations,” *Computer*, vol. 29, no. 2, 1996.
- [31] Robert Stets et al., “Cashmere-2I: Software coherent shared memory on a clustered remote-write network,” in *SOSP’97*, 1997.
- [32] Daniel Scales et al., “Shasta: a low overhead, software-only approach for fine-grain shared memory,” in *ASPLOS’96*, 1996.
- [33] Sandhya Dwarkadas et al., “Comparative evaluation of fine- and coarse-grain approaches for software distributed shared memory,” in *HPCA’99*, 1999.
- [34] Mark D. Hill et al., “Cooperative shared memory: software and hardware for scalable multiprocessors,” *ACM Trans. Comput. Syst.*, vol. 11, no. 4, 1993.
- [35] Kourosh Gharachorloo et al., “Memory consistency and event ordering in scalable shared-memory multiprocessors,” *SIGARCH Comput. Archit. News*, vol. 18, no. 3a, 1990.
- [36] Sanjeev Kumar et al., “Carbon: architectural support for fine-grained parallelism on chip multiprocessors,” in *ISCA’07*, 2007.