2009 18th International Conference on Parallel Architectures and Compilation Techniques

A Task-centric Memory Model for Scalable Accelerator Architectures

John H. Kelm, Daniel R. Johnson, Steven S. Lumetta, Matthew I. Frank*, and Sanjay J. Patel
University of Illinois at Urbana-Champaign
Urbana, IL 61801

* The author is now with Intel.

Abstract

This paper presents a task-centric memory model for 1000-core compute accelerators. Visual computing applications are emerging as an important class of workloads that can exploit 1000-core processors. In these workloads, we observe data sharing and communication patterns that can be leveraged in the design of memory systems for future 1000-core processors. Based on these insights, we propose a memory model that uses a software protocol, working in collaboration with hardware caches, to maintain a coherent, single-address space view of memory without the need for hardware coherence support.

We evaluate the task-centric memory model in simulation on a 1024-core MIMD accelerator we are developing that, with the help of a runtime system, implements the proposed memory model. We evaluate coherence management policies related to the task-centric memory model and show that the overhead of maintaining a coherent view of memory in software can be minimal. We further show that, while software management may constrain speculative hardware prefetching into local caches, a common optimization, it does not constrain the more relevant use case of off-chip prefetching from DRAM into shared caches.

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. In contrast, *compute accelerators* are hardware entities designed to improve performance and reduce power for a specific class of applications by exploiting the characteristics of the target domain. Current examples of compute accelerators include graphics processing units [1] (GPUs) and many-core variants of conventional microarchitectures such as Intel's Larrabee [2].

The design goals of contemporary CMPs are distinctly different from those of compute accelerators, which have less stringent requirements of system software, are constrained by the need for low-overhead work dispatch, and are less beholden to legacy code. Therefore, an accelerator can be optimized for a narrower class of workloads and programming styles. Furthermore, the trajectory of CMPs towards higher core counts and the growing importance of workloads that have historically targeted accelerators, such as gaming and visual computing applications, make compute accelerators a vehicle for evaluating novel system architecture for future CMPs.

In this work, we define the *task-centric memory model*, a hardware/software protocol for maintaining a coherent view of shared memory for accelerators. The model exploits sharing patterns we observe in visual computing workloads to reduce the hardware cost of coherence management. The visual computing applications we study are developed using a form of bulk synchronous processing [3] in which parallel work units, which we call *tasks*, execute independently between barriers, a period that we denote an *interval*. Logically, coherence updates and synchronization occur at the end of an interval. Our analysis of these applications shows that they have well-structured sharing patterns. The data access properties of these workloads include: a high degree of read-sharing among tasks within an interval, a private working set with updates that need only be made globally visible at the end of an interval if at all, and a small amount of data that is shared amongst tasks and must be kept coherent within an interval.

The task-centric memory model employs software-managed coherence. Our approach is similar to previous work, such as distributed shared memory (DSM) systems [4, 5] that provide the illusion of a single global address space in software on top of networked processors with distributed local memories. Our approach differs in that we target a single-chip multiprocessor with private caches where the cost of communication, through a shared global cache, is orders of magnitude less costly since it can be done on-chip. By providing explicit operations for accessing the coherent memory space, we can also provide software with a stricter consistency model on-demand. Our model in-

1089-795X/09 $25.00 © 2009 IEEE
DOI 10.1109/PACT.2009.16

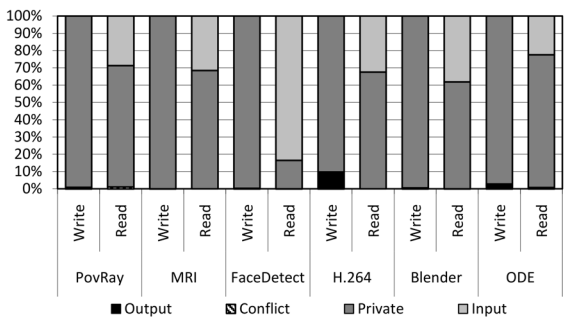

Figure 1. Read and write sharing between independent tasks in the VISBench suite

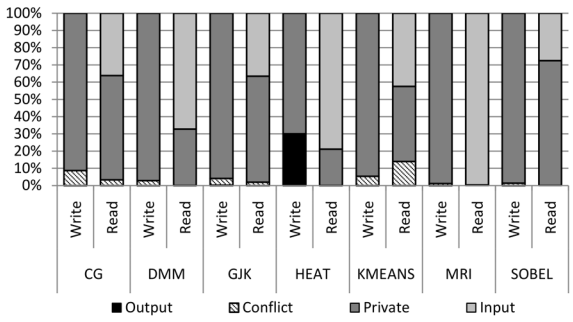

Figure 2. Read and write sharing between independent tasks in the Rigel suite.

teracts with hardware managed caches with per-word dirty bits, allowing us to take advantage of fine-grained sharing and reducing the programmability burden of false sharing while retaining the hardware support for exploiting spatial and temporal locality provided by caches.

We evaluate our task-centric memory model in the context of a clustered compute accelerator we are developing called Rigel [6]. Rigel is a 1024-core compute accelerator that has a single cacheable address space, but without hardware-enforced cache coherence across all cores on the chip. The implementation of our task management system is the Rigel Task Model (RTM). RTM utilizes the task-centric memory model to provide the illusion of a coherent address space for application developers, while supporting fine-grained parallel applications efficiently.

The contributions of this paper include:

1. the observation that data sharing patterns for a class of emerging workloads can be exploited in the design of accelerator architectures,

2. the definition of a scalable task-centric memory model for 1000-core single-chip multiprocessors,

3. the evaluation of an important optimization showing that, although arbitrary hardware prefetching is constrained by our model, prefetching from DRAM is unimpeded and most beneficial to performance, and

4. the evaluation of scheduling disciplines for coherence management operations showing that the performance overhead of the task-centric model can be minimal.

The remainder of the paper is organized as follows: Section 2 describes our motivation and presents related work. Section 3 describes RTM and the baseline architecture we use to evaluate it. Section 4 presents the task-centric memory model of the Rigel architecture. Section 5 evaluates the memory model using a 1024-core simulator. Section 6 concludes the paper.

2 Motivation and Background

Accelerators place different constraints on caches and coherence management relative to contemporary general-purpose CMPs. An opportunity exists, chiefly driven by characteristics of accelerator workloads, to exploit these differing constraints. To enlarge the space of applications accelerators target, they must not only support the data-parallel execution model prevalent today, but irregular task-parallel computation not well-suited to contemporary accelerators such as single instruction, multiple data (SIMD) GPUs. Support for flexible and evolving task management models is not easily implemented with area-efficient hardware mechanisms. Therefore, we are motivated to investigate software mechanisms when possible, and general hardware mechanisms as required, for supporting a memory model tuned for accelerators.

2.1 Application Characterization

Parallelism Structure We observe that the programming styles adopted by developers for accelerator applications share a common structure, similar to bulk synchronous processing [3]. These large-scale parallel applications are composed of a collection of concurrently executing tasks comprising mostly-data-parallel units of work. The tasks exchange little or no data within an interval. At the barrier, modified shared data is made globally visible and the next phase of computation begins.

Updates by a task during an interval can only be assumed by the programmer to be visible *after* the current interval has ended. Sharing modified data within an interval requires explicit programmer annotation. In a barrier-synchronized, mostly-data-parallel, task-based shared-memory programming model, coherence management is required to enable sharing; however, the *mechanisms* found in conventional CMP architectures to support arbitrary sharing through cache coherence are of marginal utility.

We observe that popular programming models used in developing large-scale data-parallel applications do not depend on the hardware support provided by conventional systems for arbitrary sharing. However, a mechanism for enabling some data, such as work queues or data structures, to be shared is required. Second, the common structure present in these parallel applications is rooted in the programmer's attempt to create scalable code in a manner that is conceptually simple; thus there is minimal sharing.

Sharing Patterns Our next observation is that emerging applications targeting accelerator systems have common data sharing and synchronization characteristics that can guide the design of future accelerator architectures. We provide analysis of a set of parallel visual computing workloads from VISBench [7] and from the Rigel kernel benchmark suite. VISBench consists of a set of full applications that we run on the x86 platform. The Rigel benchmarks conjugate gradient solver (CG), Sobel edge detection, k-means clustering, and dense matrix multiply (DMM) were written by hand and optimized for the Rigel architecture. The GJK collision detection benchmark was ported from a freely-available sequential version. Heat is adopted from the Cilk [8] benchmark with optimizations applied for Rigel. The MRI benchmark is a port of the VISBench [7] medical imaging code.

Analysis of these workloads shows similar data sharing and synchronization patterns. Specifically, we investigate the sharing patterns of our workloads *across synchronization boundaries*. Figure 1 shows the number of unique memory references that are shared across intervals, marked as input and output, and within an interval, marked as conflict, for VISBench applications and Figure 2 shows the same for the Rigel benchmark suite. Note that the analysis results for MRI versions differ due to the larger degree of register spilling on x86, resulting in more private reads on x86 compared to the Rigel variant. We exclude work distribution related sharing from results to highlight application-level characteristics.

Figures 1 and 2 show the frequency of non-private loads and stores, which are data produced by one task and consumed by one or more other tasks. Non-private accesses further are broken down into whether the values are shared between tasks within an interval, which we call *conflict* reads and writes, or across intervals, which we call *input reads* and *output writes*. The figures show that the majority of non-private loads are reads to data produced before the current interval began, i.e., input reads. At the same time, both conflict reads and writes to data shared within an interval are rare. Output writes, which are writes from one task in the current interval consumed by another task in the next interval, are more common in real applications than true shared writes which require intra-interval synchronization; moreover, they constitute a small fraction of overall execution. Also note that the number of unique output writes is much smaller than the number of input reads in the figure due to one-to-many sharing across intervals.

Accelerator Workload Characteristics We observe five common characteristics in accelerator workloads:

1. Large amounts of immutable, read-shared data is present within an interval. Examples of read-shared data from our workloads include scene and model descriptions or blocks of streaming media data.

2. Synchronization is coarse-grained. This in turn motivates our investigation of bulk coherence management at task boundaries. Indicative of this pattern are output writes and corresponding input reads in Figure 1 and Figure 2, which demonstrate that modified data is often read by a task *after* the interval in which the data was written has ended.

3. There exists only small amounts of write-shared data within an interval. This indicates that tasks are highly data-parallel with few data dependences between tasks within an interval. This is illustrated in Figure 1 and Figure 2 as a lack of conflict reads and writes. The conflicts that do exist are structured, such as the histogramming operation on k-means and reduction operations in CG.

4. Fine-grained synchronization is present but rare. An example of such synchronization is atomic updates to shared data structures. We observe that much of the fine-grained synchronization that we do find is used for task management and not for application code.

5. When write sharing within an interval does exist, it is usually between few sharers.

Collectively, these characteristics demonstrate that little coherence management is required within an interval, indicating the potential for pushing coherence management into software to be logically performed at the end of an interval. At the same time, mechanisms must be present to allow small amounts of fine-grained synchronization and data sharing within an interval for supporting task management efficiently. Our findings further motivate the use of shared caches that can amortize the costs associated with data access to read-shared data, a prevalent access pattern in our target workloads.

Cache Coherence Management A mechanism for maintaining coherence cannot simply be omitted from the design of future accelerators, but the constraints placed on accelerators with respect to coherence differ from those of CMPs.

CMPs rely on cache coherence and global synchronization mechanisms to provide shared resource management. Alternatively, an accelerator architecture can employ weakly-consistent memory models, 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. As a substitute for hardware cache coherence, we investigate the use of software enforcement of our task-centric memory model.

2.2 Related Work

The bulk-synchronous parallel (BSP) model was described by Valiant [3]. BSP continues to be reflected in languages prevalent today including CUDA [9] from NVIDIA and OpenCL [10]. CUDA is used to map data-parallel kernels to GPUs comprising hundreds of processing elements in a bulk-synchronous fashion, but requires SIMD-friendly code to achieve high execution efficiency. Conventional multi-core processors make use of programmer annotations, such as those provided by OpenMP [11], to identify parallel regions for the compiler. CMPs also use explicit task generation in models such as Intel's Threaded Building Blocks (TBB) [12].

Accelerator Workloads Examples of data- and task-parallel workloads that motivate our investigation of a task-parallel model include recognition, mining, and synthesis (RMS) [13] and physical simulation applications [14] for providing more realistic virtual worlds that are being investigated within Intel. A variety of highly parallel workloads, such as PARSEC [15] and ALPBench [16], have been evaluated for conventional multi-core processors. Accelerator workloads targeting current-generation GPUs have been studied [17], while studies motivating future accelerator architectures have focused on characterizing visual computing workloads [7]. While these studies investigate the scalability of visual computing workloads, we go further to point out the sharing patterns relevant to coherence management and show how these characteristics can be exploited in the design of future compute accelerators.

Memory Models Leverich et al. [18] investigate the implications of choosing between two different memory system configurations, coherent cache-based and streaming, for future CMPs. A third choice they name *Incoherent Software-based* is most similar to the model defined here, but is not investigated in that work. Rigel's memory model and coherence mechanisms are akin to software coherence mechanisms for distributed shared memory (DSM) systems. Unlike DSM systems, Rigel's model is intended for a single chip multiprocessor with a cache hierarchy that uses a shared address space for communication. Much like the task-centric model we study, Munin [5] uses multiple consistency models, based on data types specified by the programmer, allowing for communication-based per-type optimizations to be exploited by the runtime.

The consistency guarantees we investigate for write-output data at RTM task boundaries are similar to Scope Consistency [19] in that dirty data is implicitly made coherent at the end of the task's scope and updates can be deferred until the scope is reopened. TreadMarks [4] shares a property of our model not found in conventional coherence protocols: write sharing can occur without first obtaining ownership of a block, thus reducing write miss latency. Unlike TreadMarks, we do not need to reconcile dirty copies in software, but rather rely on proper use of cache management operations by software and global operations that complete at the point of coherence to enable a consistent view of write-shared data.

Eventual Consistency [20] enforces consistency using the object data abstraction, similar to the control abstraction of a task and its associated interval used here. The Cooperative Shared Memory model [21] provides a similar model to Rigel that relies on software to properly label shared accesses for performance, but relies on hardware mechanisms to guarantee correctness. Blumofe et al. [22] present Dag consistency and its associated BACKER coherence algorithm for implementing the memory model of the Cilk runtime system. Both our approach and the approach of Dag-consistency make use of synchronization structure in target applications to enforce memory ordering between executing tasks.

3 Rigel Architecture and Task Model

Rigel [6] is a MIMD compute accelerator targeting task- and data-parallel visual computing 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 high compute density by minimizing per-core area while still enabling a conventional programming model. Density is improved 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.

3.1 Overview

The fundamental processing element of Rigel is an area-optimized dual-issue in-order core with one single-precision floating-point unit and an independent fetch unit which executes a RISC instruction set. Eight cores are attached to a unified cache named the *cluster cache*. The

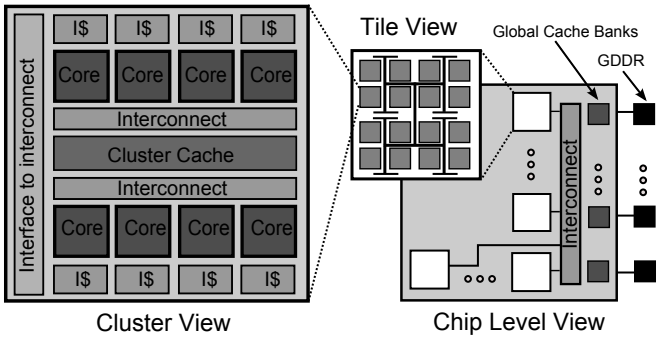

Figure 3. Diagram of the Rigel processor.

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* using a bi-directional tree-structured interconnect. Eight tiles are distributed across the chip and are attached to global cache banks via a multistage crossbar interconnect. The global caches provide buffering for multiple high-bandwidth memory controllers. Global cache banks provide a serialization point for inter-cluster shared data for maintaining a coherent view of memory. Our initial design incorporates 8 GDDR memory controllers and 32 global cache banks totaling 4 MB. The chip contains eight tiles, each tile contains 16 clusters, and each cluster consists of eight cores and the shared cluster cache.

3.2 Cache Management

All cores share a single global address space. Cores within a cluster have the same view of memory due to the shared cluster cache, while global coherence is not explicitly maintained by the hardware between clusters. When serialization of accesses is necessary between clusters, the global cache is the point of coherence. To access each level of cache directly, Rigel implements two classes of memory operations: *local* and *global*.

Local memory operations are intended to constitute the majority of memory operations. Low-latency and high-bandwidth memory accesses are achieved using local operations. Local read operations are cacheable at the cluster cache, but are not kept coherent between clusters by hardware. Local memory writes follow a writeback policy at the cluster cache: on eviction from the cluster cache, modified data is written back to the global cache. From the perspective of the programming model, local operations are used for accessing read-only data, private data, and data that is shared intra-cluster.

Global loads, stores, and atomic read-modify-write operations on Rigel are not cached by the cluster cache and complete at the global cache, which serves as the point of global coherence. Memory locations operated on solely by global memory operations are kept coherent across the chip. Global operations are key to providing system resource management and synchronization for a chip that supports global cache coherence in software. Global memory operations also enable fine-grained inter-cluster communication by way of the global caches without the need to obtain ownership as is found in invalidate-based coherence protocols. The cost of global memory operations is high relative to local operations due to the greater 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, the latency of performing a global operation, and cluster-to-global cache interconnect bandwidth.

3.3 Rigel Task Model

The Rigel Task Model is a queue-based low-level programming model, described in [6], that enforces coherence in software and performs synchronization using barriers. The Rigel ISA provides instruction primitives useful for implementing task management, such as local and global atomic operations, but does not provide explicit support for task management. In this section, we describe the relevant pieces of the API of the Rigel Task Model to provide background and the implementation details relevant to supporting the task-centric memory model.

Software API The software API for the Rigel Task Model is composed of basic operations for managing the resources of queues located in memory and 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 tasks that are inserted and removed from queues between barrier operations. The barriers thus provide a partial ordering of tasks. Barriers are used to synchronize the execution of all cores using the queue and define a point at which all locally-cached non-private data modified during that interval must be made coherent. Coherence is enforced by writing back modified, write-output data to the global cache and invalidating non-private, read-input data in the cluster cache. Write-shared data within an interval must be specified by the programmer. Intrinsics are provided by the API for global memory operations and atomic operations that are kept coherent across tasks within an interval.

Queue Management The Rigel Task Model provides the following set of API calls to the programmer: `TQ_Create`, `TQ_EnqueueLoop`, and `TQ_Dequeue`. `TQ_Create` allocates resources for the queue and makes it available to

the system. Each `TQ_Dequeue` action operates on a single *task descriptor*. A unique task descriptor is generated for each task enqueued and contains two user-defined word-sized data fields and two parameters set by the runtime to a range that can represent values such as loop iterations. The `TQ_EnqueueLoop` operation provides a single operation to enqueue a DO-ALL-style parallel loop similar to the loop operation available for Carbon [23]. The runtime uses parameters to the enqueue call to select the proper range to deliver to dequeuing cores.

An initialized queue can be in one of the following states: *tasks-available*, *empty*, or *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. Any core that attempts a dequeue operation with an empty queue will block. 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.

The *completed* state is used to provide an implicit barrier in the Rigel Task Model. When all cores participating in a barrier interval have completed all tasks, all cores will be 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. Although our fork operation does not have the synchronization semantics found in other models, such as TBB [12] and Cilk [8] where a fork operation is a synchronization between parent and child, the semantics of the completed state are such that tasks can be enqueued at any point within an interval—tasks are not constrained to only be enqueued 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 is not known a priori.

Implementation RTM is a highly tunable multi-level hierarchical task queuing system running on Rigel. Note that the implementation detailed here allows 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. Coherence management is intertwined with the RTM implementation. Multiple policies for managing coherence are described and evaluated in Section 5. We leave the study of alternate implementations, tuning, and optimizations to future work.

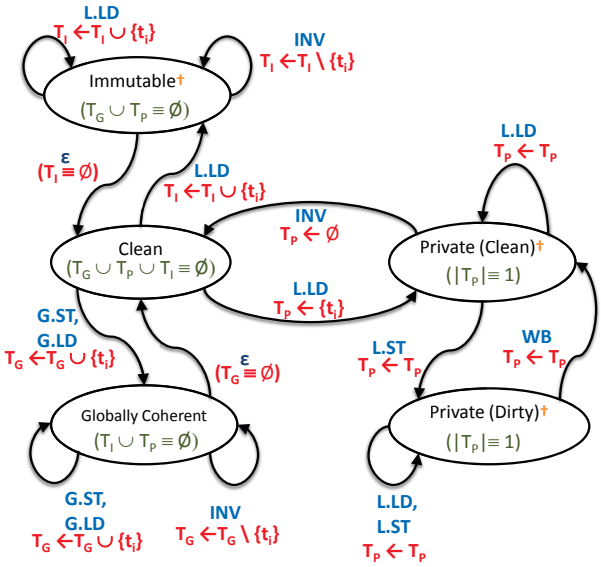

Figure 4. State transitions for memory blocks in the task-centric memory model. Actions include: Local loads (L.LD), local stores (L.ST), global loads (G.LD), global stores (G.ST), write backs to the global cache (WB), and cluster cache invalidates (INV). The † notes states that may cache a block at the cluster cache. The set of tasks sharing a block in state X is denoted by T_X. Any unlisted transition is disallowed by the model.

4 Memory Model

The memory model we define allows software to achieve the behavior of a coherent design without hardware cache coherence by having the programmer reason about memory blocks that are read-only, shared, or private during each interval. The task-centric memory model defines a *coherence domain* as a logical grouping of memory blocks for which coherence guarantees are provided collectively by the model. Software performs the necessary actions to transition blocks between the different domains during program execution. Once a block is moved into a state other than the initial state, i.e., the `clean` state to be described shortly, during an interval, it cannot transition to another coherence domain until after a global synchronization point is reached.

4.1 Coherence Algorithm

A block of memory follows the state machine depicted in Figure 4. Tasks t_i operate on blocks using the following memory instructions: Local loads (`L.LD`), local stores (`L.ST`), global loads (`G.LD`), global stores (`G.ST`), write-back operations that write a line back to the global cache if present in the cluster cache and mark the line unmodified at the cluster cache (`WB`), and invalidate operations that make the line invalid in the cluster cache if present (`INV`). Each

set represents the collection of tasks sharing a block in a particular state: `clean` (T_C), `globally coherent` (T_G), `immutable` (T_I), and `private` (T_P). Note that the private domain is broken into two states for clarity.

There are six properties defined for the memory model:

1. All blocks start in the `clean` state. ($\emptyset \equiv T_I \cup T_G \cup T_P | time = 0$)
2. Blocks may only transition between accessible states by first passing through the `clean` state and after a barrier is reached.
3. A barrier is a global point of synchronization. All memory operations performed before a barrier must be complete after that barrier.
4. A block may be in exactly one accessible state from the perspective of all cores in the system at any time. $((\emptyset \equiv T_I \cap T_G) \wedge (\emptyset \equiv T_I \cap T_P) \wedge (\emptyset \equiv T_G \cap T_P))$
5. A block in the `private` state must have $|T_P| \equiv 1$.
6. A block in the `globally coherent` state returns the last write to that location and all cores in the system see the same ordering of updates to that location, i.e., the block is kept coherent.

The software coherence protocol must interact properly with the underlying hardware to ensure correct execution. For instance, the `private (clean)` state corresponds to a data value in the cluster cache that does not have its dirty bit set. The cluster cache controller may invalidate the line on an eviction, implicitly moving the line into the `clean` state. Should the core previously holding the block in the `private` state reissue a load to that location, the cluster cache controller must fetch the value from the global cache. The value is guaranteed to return the same value as if the eviction had not occurred since, by properties 4 and 5 above, ownership of the block is held solely by the core issuing the load. Note that global atomics are performed only to `globally coherent` blocks and have the same semantics from the perspective of the memory model as global loads and stores. Having two distinct cluster caches hold the same block in the dirty state represents a race condition that is possible in hardware, but is disallowed by the coherence protocol defined above.

4.2 Memory Ordering

Ordering of memory operations is defined separately for operations performed within distinct coherence domains. Ordering must be defined when *conflicting* accesses exist. A conflict is defined as at least two cores accessing the same block with at least one access being a write. Blocks in the `clean` and `immutable` states can never have conflicting accesses by definition. Blocks in the `clean` and `immutable` states have a single value that is visible by all cores in the system.

Property 5 of the memory model ensures that updates to `private` blocks are only ever visible to a single core and therefore by definition no conflicting accesses may occur. Loads by a core return the last store to the block performed by the core while in the `private` state or, if the block has not been written by the core since becoming `private`, the value of the block when it was in the `clean` state is returned. Blocks in the `private` state therefore need only respect dependences constrained by program order. Accesses to `private` blocks between cores are unordered.

Conflicts may occur for blocks in the `globally coherent` state. We define the ordering of all blocks in the `globally coherent` state to conform to processor consistency [24]. The choice of processor consistency is fueled by the desire to maintain as strict of a consistency model as possible without precluding the hardware optimizations of benefit to a hierarchical clustered accelerator. The strongest model, sequential consistency [25], would preclude write buffering, message reordering in the network, and multi-banked global caches and was therefore rejected. A weaker model than sequential consistency, total store order [26], was rejected as it would have disallowed a core to read global stores performed by other cores within its cluster early. Processor consistency allows for the aforementioned optimizations without requiring explicit synchronization to achieve correct ordering, as would be needed to support weak consistency [27], for relevant use cases on Rigel.

A global ordering of accesses is defined at barriers by property 3. For implementation and optimization reasons, the memory model defines ordering between dependent operations that *cross coherence domains* from a single core similarly to weakly consistent models. The memory model defines that reads to `private` blocks followed by writes to `globally coherent` blocks from a single core respect program order. Reads to `immutable` or `globally coherent` blocks followed by writes to `private` blocks from a single core respect program order. Other orderings across cores and coherence domains are undefined by the model. A memory fence operation is provided by the ISA to ensure all memory operations, including writebacks and invalidates, issued by a core executing the fence complete before the fence may retire. No new memory operation may be initiated until after the fence has retired. A global memory fence can be constructed by all cores issuing a memory fence prior to entering a global synchronization barrier.

4.3 Optimizations

The task-centric memory model is able to provide the appearance of a coherent single address space on a chip multiprocessor without hardware cache coherence. However, strict adherence to the model would: limit software and hardware prefetching capabilities, force shared data to conservatively access high-latency global caches, and unnecessarily require aggressive invalidation and cache flushing. By extending the baseline model presented, many of these issues can be addressed. We briefly cover some of the issues here, but leave further analysis to future work.

We evaluate different policies for deciding when to perform coherence actions *before* the end of an interval. Further optimization can be performed by taking a *thread-centric* view of coherence management, i.e., a view that considers the sequence of all tasks run on a single core within an interval as one unit for which to schedule coherence actions instead of at the completion of tasks. As an example, we can weaken property 2 by adding: Not all blocks need to be clean at barriers, only those that undergo state transition across the barrier. While the general problem of determining what data may be made coherent lazily is difficult, there are opportunities to exploit always-private data, such as stack allocations, and programmer assertions for immutable data, such as the `const` keyword in C.

When available, locality could be exploited by augmenting an underlying assumption of the model that a task maps to a single core. Optimization can be performed using cluster-level sharing by extending the model to map tasks to clusters in groups instead of a single task to a core. By reconsidering the level at which work is mapped to execution resources, a low-level cache, such as the cluster cache on Rigel, can now be used as the point of coherence for data. Doing so allows for data that would otherwise be required to exist in the `globally coherent` state, thus suffering high latency to access the highest level of the hierarchy, to be effectively privatized to more local caches when all tasks accessing the data can be co-located as part of a group.

The task-centric memory model supports staged porting of applications initially developed assuming full hardware cache coherence and porting efforts starting from a sequential implementation. To do so, initially the `globally coherent` state is used for all data in the application to provide the appearance that all data is kept coherent at all times. While a performance penalty is paid for using the `globally coherent` state for all data due to the restrictions on local caching, the assumption of coherence holds, and thus enables correctness for ported software. With a correct implementation on the new platform to serve as a baseline, software can be modified to make use of other states in the memory model to improve performance by relaxing coherence guaranties as needed.

5 Evaluation

In this section we evaluate the task-centric memory model using an implementation of the Rigel Task Model running on an execution-driven 1024-core simulator of the Rigel accelerator. Previous work [6] has shown the scalability of the Rigel Task Model and the cost of intrinsics it provides. In this work, we demonstrate two key results. The first is that the overhead of software-enforced coherence, compared to an optimistic hardware-coherent baseline, is less than 10% in most cases and that eager coherence actions can even improve performance in other cases by reducing instantaneous bandwidth demands placed on the system at barriers. The second result is that a common hardware optimization, hardware prefetching, is highly beneficial to performance when performed by the global cache, a case allowed trivially by our model, and is of questionable benefit when performed at the cluster cache, the case that is handled with difficulty by our model.

5.1 Methodology

The benchmarks presented are written in C using the Rigel Task Model API. System call and task management overhead is simulated. 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. The choice of data to write back and invalidate is driven by programmer annotation to maintain the model presented in Section 4. Note that data that is immutable for the duration of the benchmark is not invalidated in our benchmarks. We evaluate six of the optimized parallel benchmarks described in Section 2.1.

5.2 Discussion

Impact on Hardware Optimizations The task-centric memory model provides the benefits of reduced complexity and increased density, but may constrain the design of a processor that implements the model. One limitation is the system's restricted ability to do speculative non-binding prefetches since speculative hardware cache actions are invisible to the software-enforced coherence algorithm defined earlier. However, a benefit of our model is that while prefetching into the cluster cache cannot be done arbitrarily, the model is defined such that hardware prefetching from DRAM into the global cache is feasible. Instruction prefetching is not impeded by our proposed memory model since instructions are assumed immutable. Enabling self-modifying code written and executed within the same interval is outside the scope of this work.

The lack of hardware-managed coherence inhibits non-binding hardware prefetching at the cluster cache. The rea-

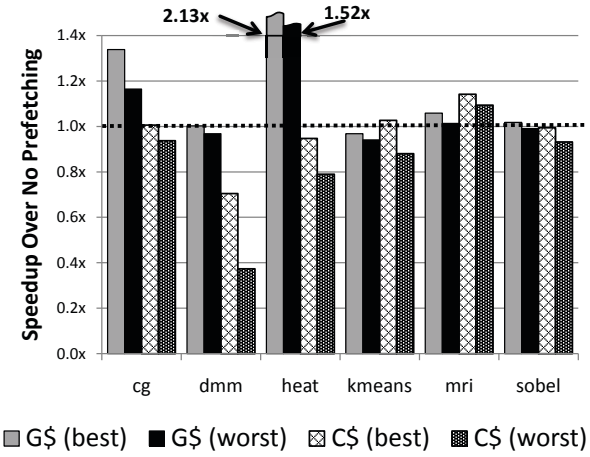

Figure 5. Best- and worst-case performance for global (G$) and cluster cache (C$) next-N-line ($N = (1, 2, 4, 8)$) prefetching normalized to no prefetching ($N = 0$).

son is that the cluster cache prefetcher brings lines into the cache speculatively, thus taking them out of the coherence protocol because software will have incomplete knowledge of the cluster cache contents. The prefetched lines may not be invalidated by the software coherence protocol and thus the earlier prefetch action could then result in a stale value being read later. We note that software prefetches that are tracked as part of the software algorithm presented are not limited in our model and can be used where necessary.

Hardware prefetching at the global cache is not impeded. Blocks in the global cache are kept coherent with memory, resulting in prefetching actions from the global cache being transparent to the task-centric memory model. Global cache accesses will return the same value to a core in the three possible states a block may be in: **Case 1:** the block is already present in the global cache due to an earlier read or write to the block (a normal memory operation generated the original request, not a prefetch); **Case 2:** the block is not present in the global cache and the block is retrieved from memory (no prefetch occurred and DRAM has the current value); or **Case 3:** if a prefetch operation was performed previously, the block is in the cache and is returned (the value returned is the same from the prefetch as would be returned by having the miss generate a DRAM request). Note that per-word dirty bits are tracked at both the cluster and global caches allowing for us to avoid lost updates due to false sharing. Figure 5 shows the value of next-N-line prefetching at the global cache (G$ in the figure). We evaluate the potential benefits of prefetching at the cluster cache (C$ in the figure) using an omniscient memory model that eliminates spurious values due to prefetching, but still models cache timing. To elucidate, a cluster cache miss will be charged the cost of going to the global cache in the omniscient model, but a load performed to a line cached at the cluster cache from a prefetch occurring in the previous interval returns the latest value from an update by another cluster during the current interval and is only charged the cost of a cluster cache hit. To demonstrate the range of benefit provided by hardware prefetching, we show the best and worst performing prefetcher at both the global and cluster level.

Cores sharing a small local cache, the cluster cache on Rigel, create tension between common cache-oriented optimizations, such as blocking the data set, and useless hardware prefetches that pollute the cache by evicting useful data. Due to this tension, some benchmarks benefit slightly from cluster cache prefetching, while some benchmarks, such as `dmm` and `heat`, suffer degraded performance. A pathological example is `dmm`, which has performance that scales inversely with the number of lines prefetched on a miss. The effect is due to useless prefetches from areas outside of the matrix blocks in the cache, evicting read-shared/reused data already present.

Figure 5 shows that hardware prefetching at the global cache provides a large benefit in the best case and rarely hurts performance measurably in the worst case. The rationale is that a large amount of on-chip bandwidth is available for servicing cluster cache misses at low latency while cluster cache size is limited. At the same time, the amount of off-chip bandwidth for servicing global cache misses is an order of magnitude less. The internal structure of DRAM makes random access expensive and, coupled with the interleaving of MIMD memory access streams, frustrates memory scheduling, increasing the cost of DRAM accesses due to queuing delays from the global cache through the network. Intuitively, more emphasis should be placed on reducing the cost and frequency of random DRAM accesses than on reducing the frequency of global cache accesses. Our results show that a large performance benefit is possible using global cache prefetching, which is inherently supported by our memory model, while the benefits of cluster cache prefetching are not clear and not trivially supported.

Coherence Management Optimizations A naïve implementation of the memory model, which strictly adheres to task-centric actions, would require a large number of write-backs and invalidates to occur at task boundaries. The added memory traffic at the start and end of each task may lead to poor bandwidth utilization. Due to queuing delays in the network and at the memory controller, the latency for memory operations during these periods also grows precipitously. Lastly, the read sharing benefit of immutable data would be decreased if shared data were aggressively invalidated from the shared cluster caches. The combination of these effects leads us to explore alternative policies for scheduling coherence actions.

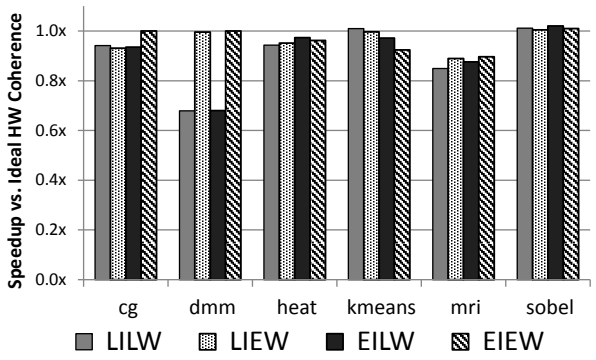

Figure 6. Speedup of the four combinations of eager/lazy and invalidation/writeback policies relative to zero-cost coherence.

Coherence actions need not occur at task boundaries. Coherence actions can be deferred by the runtime as long as state changes that occur across a barrier are completed by the end of the interval. To that end, we evaluate combinations of two policies, *lazy* and *eager*, for the writeback and invalidate components of coherence management. We use an optimistic baseline that mimics the effects of write-update hardware coherence with zero-cost updates between cluster caches; no software coherence actions are taken in the baseline. Lazy actions occur en masse at barriers and eager actions occur at task boundaries. The results in Figure 6 show eager invalidate/eager writeback (EIEW), lazy invalidate/eager writeback (LIEW), eager invalidate/lazy writeback (EILW), and lazy invalidate/lazy writeback (LILW) relative to the optimistic baseline.

The results show that different policies provide the best performance for each benchmark and that only one benchmark (`mri`) suffers greater than 10% overhead relative to an optimistic zero-cost hardware coherence baseline at eight tiles (1024 cores) due to software coherence actions. Since the model is under software control, a mix of policies across applications can be deployed. In general we find two trends. First, eager writebacks overlap write traffic with useful execution and should be used as much as possible to increase memory system concurrency. The coherence actions result in less bursty load on the interconnect, increasing performance. Brewer and Kuszmaul [28] observe a similar effect due to output port contention on the CM-5. Second, lazy invalidation allows for shared read-input data to be exploited opportunistically when two tasks share read values and execute on the same core, or in the same cluster on Rigel, during an interval.

6 Conclusion

We describe a task-centric memory model and evaluate it in the context of a work management and scheduling system, the Rigel Task Model, for MIMD compute accelerators that provide minimal hardware support for coherence and task management. Our implementation of the Rigel Task Model demonstrates that the subset of system services required by compute accelerators can be implemented efficiently without overly specialized hardware using our memory model.

Based on our evaluation, we provide a perspective on the constraints placed on the memory model in future accelerators. For scalable accelerator applications, there is a pattern to sharing that would indicate that hardware coherence may be of marginal utility, but we find efficient coherence maintenance is necessary for implementing system services and some intra-barrier sharing. The Rigel Task Model and the task-centric memory model presented take advantage of the minimal sharing while enabling inter-task data sharing when necessary.

We find that our model could restrict conventional prefetching techniques that attempt to reduce latency of access to private caches; however, our model does not limit performance since prefetching from system memory into global shared caches is not impeded, a greater concern for pin-constrained 1000+ core accelerators.

Acknowledgment

The authors acknowledge the support of the Focus Center for Circuit & System Solutions (C2S2 and GSRC), two of the five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation Program. The authors thank the Trusted ILLIAC Center at the Information Trust Institute for their contribution of use of their computing cluster. The authors also wish to thank Naveen Neelakantam, Matt R. Johnson, Aqeel Mahesri, and the anonymous referees for their input and feedback. John Kelm was partially supported by a fellowship from ATI/AMD.

References

[1] NVIDIA, "NVIDIA GeForce 8800 GPU architecture overview," November 2006.

[2] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: a many-core x86 architecture for visual computing," *ACM Trans. Graph.*, vol. 27, 2008.

[3] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, 1990.

[4] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "Treadmarks: Shared memory computing on networks of workstations," *Computer*, vol. 29, no. 2, 1996.

[5] J. K. Bennett, J. B. Carter, and W. Zwaenepoel, "Munin: distributed shared memory based on type-specific memory coherence," in *PPoPP'90*. New York, NY, USA: ACM, 1990, pp. 168–176.

[6] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel, "Rigel: An architecture and scalable programming interface for a 1000-core accelerator," in *ISCA'09*, June 2009.

[7] A. Mahesri, D. Johnson, N. Crago, and S. J. Patel, "Tradeoffs in designing accelerator architectures for visual computing," in *MICRO'08*, 2008.

[8] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," *SIGPLAN Not.*, vol. 33, no. 5, 1998.

[9] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *Queue*, vol. 6, no. 2, 2008.

[10] *OpenCL Specification*, 1st ed., Khronos OpenCL Working Group, December 2008.

[11] OpenMP Architecture Review Board, "OpenMP application program interface," May 2008.

[12] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly, 2007.

[13] P. Dubey, "Recognition, mining and synthesis moves computers to the era of tera," *Technology Intel Magazine*, Feb. 2005.

[14] C. J. Hughes, R. Grzeszczuk, E. Sifakis, D. Kim, S. Kumar, A. P. Selle, J. Chhugani, M. Holliman, and Y.-K. Chen, "Physical simulation for animation and visual effects: parallelization and characterization for chip multiprocessors," in *ISCA'07*, 2007.

[15] C. Bienia, S. Kumar, J. P. Singh, and K. Li., "The PARSEC benchmark suite: Characterization and architectural implications," Princeton University, Tech. Rep. TR-81108, January 2008.

[16] M.-L. Li, R. Sasanka, S. Adve, Y.-K. Chen, and E. Debes, "The ALPBench benchmark suite for complex multimedia applications," *IWCS'05*, Oct. 2005.

[17] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. mei W. Hwu, "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," in *PPoPP'08*, 2008.

[18] J. Leverich, H. Arakida, A. Solomatnikov, A. Firoozshahian, M. Horowitz, and C. Kozyrakis, "Comparing memory systems for chip multiprocessors," in *ISCA'07*, 2007, pp. 358–368.

[19] L. Iftode, J. P. Singh, and K. Li, "Scope consistency: A bridge between release consistency and entry consistency," in *SPAA'96*, 1996, pp. 277–287.

[20] B. Bershad, M. Zekauskas, and W. Sawdon, "The midway distributed shared memory system," *Compcon Spring '93, Digest of Papers.*, pp. 528–537, Feb 1993.

[21] M. D. Hill, J. R. Larus, S. K. Reinhardt, and D. A. Wood, "Cooperative shared memory: software and hardware for scalable multiprocessors," *ACM Trans. Comput. Syst.*, vol. 11, no. 4, 1993.

[22] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall, "Dag-consistent distributed shared memory," in *IPPS'96*, 1996, pp. 132–141.

[23] S. Kumar, C. J. Hughes, and A. Nguyen, "Carbon: architectural support for fine-grained parallelism on chip multiprocessors," in *ISCA'07*, 2007.

[24] J. Goodman, "Cache consistency and sequential consistency," SCI Working Grp., Tech. Rep. 61, March 1989.

[25] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computers*, vol. C-28, no. 9, pp. 690–691, September 1979.

[26] *SPARC Architecture Manual, Version 9*, SPARC International Inc., September 2000.

[27] S. V. Adve and M. D. Hill, "Weak ordering—a new definition," in *ISCA'90*, 1990, pp. 2–14.

[28] E. A. Brewer and B. C. Kuszmaul, "How to get good performance from the CM-5 data network," in *ISPP'94*, 1994, pp. 858–867.