A Task-Centric Memory Model for Scalable Accelerator Architectures

THIS ARTICLE PRESENTS A MEMORY MODEL FOR PARALLEL COMPUTE ACCELERATORS WITH TASK-BASED PROGRAMMING MODELS THAT USES A SOFTWARE PROTOCOL, WORKING IN COLLABORATION WITH HARDWARE CACHES, TO MAINTAIN A COHERENT, SINGLE ADDRESS SPACE VIEW OF MEMORY WITHOUT REQUIRING HARDWARE CACHE COHERENCE. THE MEMORY MODEL SUPPORTS VISUAL COMPUTING APPLICATIONS, WHICH ARE BECOMING AN IMPORTANT CLASS OF WORKLOADS CAPABLE OF EXPLOITING 1,000-CORE PROCESSORS.

John H. Kelm
Daniel R. Johnson
Steven S. Lumetta
Sanjay J. Patel
University of Illinois at Urbana-Champaign

Matthew I. Frank
Intel

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 target domain's characteristics. Current examples of compute accelerators include graphics processing units (GPUs)[1] and many-core variants of conventional microarchitectures, such as Intel's Larrabee.[2] The design goals of compute accelerators place less stringent requirements on system software, are constrained by the need for low-overhead work dispatch, and are less beholden to legacy code than contemporary CMPs. Therefore, we can optimize an accelerator for a narrower class of workloads and programming styles. Furthermore, the tendency of CMPs toward 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 architectures 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. The model exploits sharing patterns in accelerator workloads to reduce the hardware cost of coherence management. The accelerator workloads we study are drawn from the visual computing domain and were developed using a form of bulk synchronous processing.[3] In these workloads, 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. These workloads' data-access properties 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 sometimes a small amount of data that is shared among tasks and must be kept coherent within an interval.

Our model uses 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. In our context, the cost of communication through a shared global cache is orders of magnitude less costly because 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. Our model interacts with hardware managed caches with per-word dirty bits, letting us exploit fine-grained sharing and reduce the programmability burden of false sharing while retaining the hardware support provided by caches for exploiting spatial and temporal locality.

Application characteristics

Applications developed for accelerators today, including the visual computing workloads we study here, share common data sharing patterns and parallelism structure, which includes how and when interacting tasks can synchronize. We can exploit the characteristics of such scalable parallel applications when developing a memory model for accelerators.

Parallelism structure

The programming styles adopted by many developers for accelerator applications share a common structure, similar to bulk synchronous processing.[3] These large-scale parallel applications consist of interval sequences. Within each interval, a collection of concurrently executing tasks perform mostly data-parallel units of work. Tasks exchange little or no data within an interval. Each interval ends with a barrier, at which point modified shared data becomes globally visible and the next computation phase begins.

During an interval, the programmer can only assume a task's updates are visible after the current interval ends. Sharing modified data within an interval requires explicit programmer annotation. A barrier-synchronized, mostly data-parallel, task-based shared-memory programming model requires coherence management to enable sharing. However, the mechanisms found in conventional CMP architectures to support arbitrary sharing through cache coherence are of marginal utility.

Popular programming models used in developing large-scale data-parallel applications don't depend on the hardware support provided by conventional systems for arbitrary sharing. However, they do require a mechanism for allowing some data to be shared, such as work queues. Second, the common structure present in these parallel applications is rooted in the programmer's attempt to create scalable code in a conceptually simple manner. Thus, sharing is minimal and well-structured.

Sharing patterns

Emerging applications targeting accelerator systems have common data-sharing and synchronization characteristics that can guide the design of future accelerator architectures. We analyzed a set of parallel visual computing workloads from VISBench[6] and from the Rigel kernel benchmark suite.[7]

Specifically, we investigated the sharing patterns of our workloads across synchronization boundaries. We excluded work-distribution-related sharing from results to highlight application-level characteristics. Analyzing these workloads showed similarities in data sharing and synchronization patterns across the two benchmark suites. 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 the Rigel benchmark suite. The analysis results for MRI versions differ due to the larger degree of register spilling on x86, resulting in more private reads on x86 than on the Rigel variant.

Figure 1 shows the frequency of non-private loads and stores, which are data produced by one task and consumed by one or more others. Figure 2 illustrates the common sharing patterns. We further broke down nonprivate accesses 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*. Figures 1 and 2 show that most nonprivate loads are reads to data produced before the current interval began—that is, input reads. At the same time, both conflict reads and writes to data shared within an interval are rare. Output writes, which are 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 that require intrainterval synchronization. Moreover, they constitute a small fraction of overall execution. Also, the number of unique output writes is much smaller than the number of input reads due to one-to-many sharing across intervals.

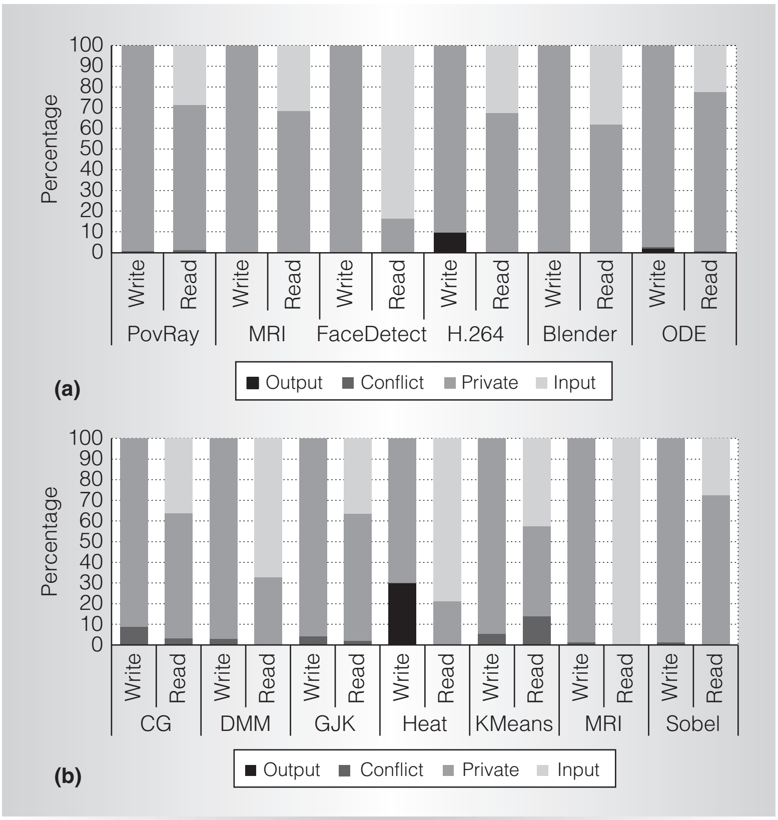
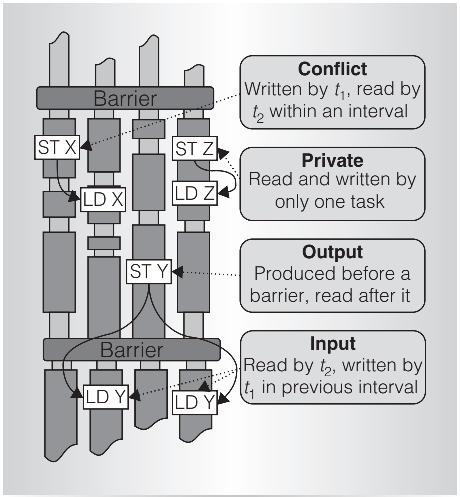


Figure 1. Read and write sharing between independent tasks in the VISBench and Rigel suites.

Accelerator workload characteristics

We observed five common characteristics of accelerator workloads:

- Large amounts of immutable, widely 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.
- Synchronization is coarse-grained, which motivates our investigation of bulk coherence management at task boundaries. Indicative of this pattern are the output writes and corresponding input reads in Figure 1, which demonstrate that modified data is often read by a task after the interval in which the data was written has ended.
- Only small amounts of write-shared data exist within an interval, which indicates that tasks are highly data parallel with few data dependences between tasks within an interval. Figure 1 demonstrates the lack of fine-grained write sharing as a lack of conflict reads and writes. Such conflicts consist primarily of collective operations, such as histogramming operations in k-means and reduction operations in conjugate gradient (CG).



Barrier
ST X
LD X
ST Z
LD Z
ST Y
Barrier
LD Y
LD Y
Conflict
Written by t_1, read by t_2 within an interval
Private
Read and written by only one task
Output
Produced before a barrier, read after it
Input
Read by t_2, written by t_1 in previous interval

Figure 2. Intertask and intratask read-to-write classifications.

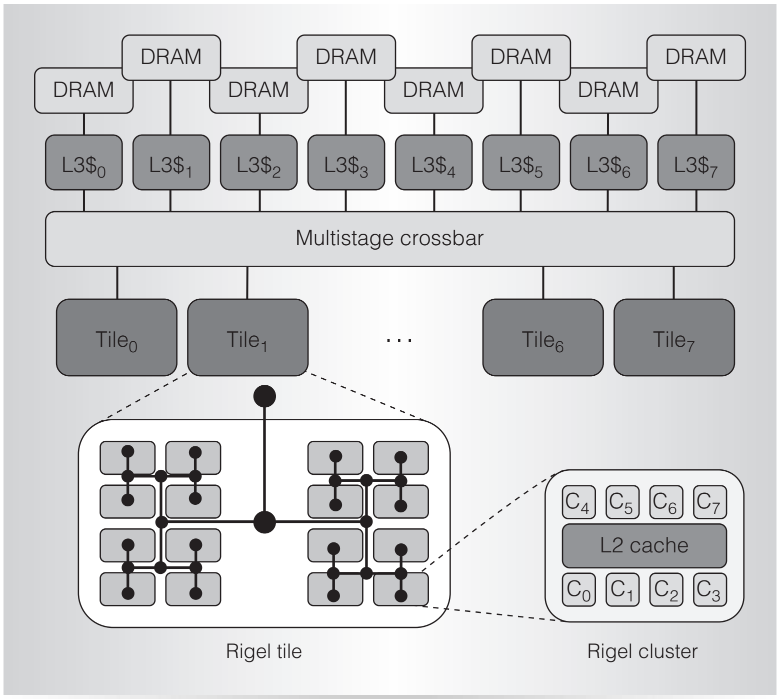


Figure 3. Diagram of the Rigel processor.

- Fine-grained synchronization is present, but rare. An example of such synchronization is atomic updates to shared data structures. Much of the fine-grained synchronization we found is used for task management and not for application code.
- When write sharing within an interval exists, 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 allow small amounts of fine-grained synchronization and data sharing within an interval to support task management and collective operations 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

We can't simply omit a mechanism for maintaining coherence from the design of future accelerators. The constraints placed on accelerators with respect to coherence differ from those of CMPs, which rely on cache coherence and global synchronization mechanisms to provide shared resource management. Alternatively, an accelerator architecture can use weakly consistent memory models, distinct local and globally visible 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 investigated the use of software enforcement with our task-centric memory model.

Rigel architecture and task model

We evaluated the task-centric memory model using Rigel,[7] which is a multiple instruction, multiple data (MIMD) compute accelerator targeting task- and data-parallel visual computing workloads in computer vision, imaging, and physical simulation that scale up to thousands of concurrent tasks. Rigel's design goal is to provide high compute density by minimizing per-core area while still enabling a conventional programming model. We can improve density by removing features found in conventional designs that are of minimal benefit to the workloads Rigel targets. Figure 3 shows a block diagram of Rigel.

Rigel's fundamental processing element is an area-optimized dual-issue in-order core with one single-precision floating-point unit and an independent fetch unit that executes a reduced-instruction-set computing (RISC) instruction set variant. Eight cores are attached to a unified L2 cache named the *cluster cache*. The cores, core-to-cluster-cache interconnect, and the cluster-to-global interconnect logic comprise a single Rigel cluster. We connect and group clusters logically into a *tile* using a bidirectional tree-structured interconnect. Eight tiles are distributed across the chip and are attached to global L3 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 intercluster shared data for maintaining a coherent view of memory. Our initial design incorporates eight memory controllers and 32 global cache banks totaling 4 Mbytes. The chip contains eight tiles, each tile contains 16 clusters, and each cluster consists of eight cores and the shared cluster cache.

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 between clusters isn't maintained by the hardware. When serialization of accesses between clusters is necessary, the global cache is the point of coherence. To access each cache level directly, Rigel implements two classes of memory operations: local and global.

Local memory operations 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 aren't kept coherent between clusters by hardware. Local memory writes follow a write-back policy at the cluster cache; on eviction from the cluster cache, modified data is written back to the global cache. From the programming model's perspective, local operations are used for accessing read-only data, private data, and data that is shared intracluster.

Global loads, stores, and atomic read-modify-write operations on Rigel bypass 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. Globally visible operations are key to supporting system resource management and synchronization for a chip that supports cache coherence in software. Global memory operations also enable fine-grained intercluster communication by way of the global caches without needing to obtain ownership as is necessary in invalidation-based coherence protocols. Global memory operations are costlier than 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.

Rigel task model

The Rigel task model is a queue-based low-level programming model that enforces coherence in software and performs synchronization using barriers. The Rigel instruction set architecture (ISA) provides instruction primitives useful for implementing task management, such as local and global atomic operations, but it doesn't explicitly support task management. By providing flexible primitives in lieu of hardware task management support, Rigel can support a wider array of task distribution and scheduling policies than could be easily implemented in a purely hardware design.

The software API for the Rigel task model consists 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 in which 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 help synchronize the execution of all cores using the queue and define a point at which all locally cached nonprivate 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 nonprivate, read-input data in the cluster cache. The programmer must specify write-shared data within an interval. The API provides intrinsics for global memory operations and atomic operations that are kept coherent across tasks within an interval.

Memory model

In the absence of hardware support for coherence, achieving consistent behavior requires a clear memory-use model. We leveraged the bulk-synchronous structure of many parallel applications as well as the implications of this structure, as we illustrated earlier, to develop the task-centric memory model. By having the programmer

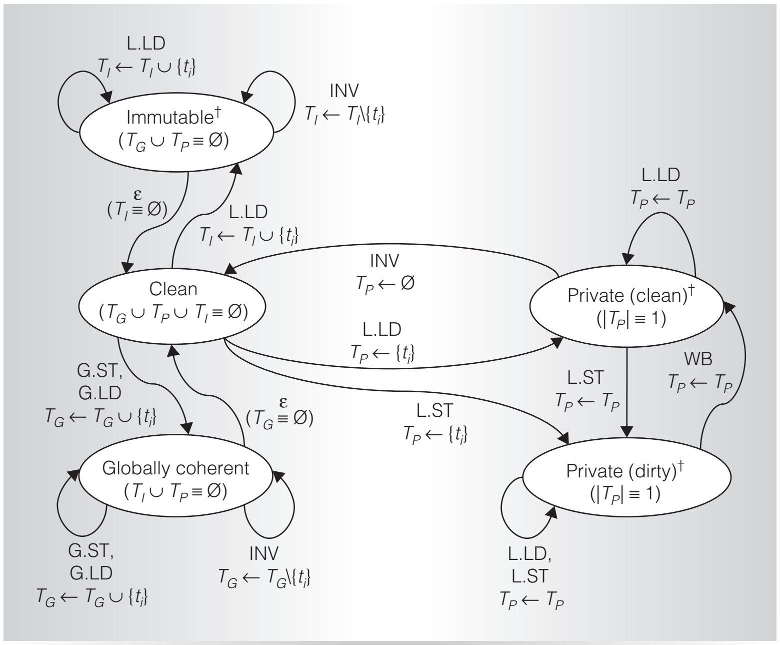

L.LD
$T_I \leftarrow T_I \cup \{t_i\}$
INV
$T_I \leftarrow T_I \backslash \{t_i\}$
Immutable†
$(T_G \cup T_P \equiv \emptyset)$
ε
$(T_I \equiv \emptyset)$
L.LD
$T_I \leftarrow T_I \cup \{t_i\}$
L.LD
$T_P \leftarrow T_P$
Clean
$(T_G \cup T_P \cup T_I \equiv \emptyset)$
INV
$T_P \leftarrow \emptyset$
Private (clean)†
$(|T_P| \equiv 1)$
L.LD
$T_P \leftarrow \{t_i\}$
G.ST,
G.LD
$T_G \leftarrow T_G \cup \{t_i\}$
ε
$(T_G \equiv \emptyset)$
L.ST
$T_P \leftarrow \{t_i\}$
L.ST
$T_P \leftarrow T_P$
WB
$T_P \leftarrow T_P$
Globally coherent
$(T_I \cup T_P \equiv \emptyset)$
Private (dirty)†
$(|T_P| \equiv 1)$
G.ST,
G.LD
$T_G \leftarrow T_G \cup \{t_i\}$
INV
$T_G \leftarrow T_G \backslash \{t_i\}$
L.LD,
L.ST
$T_P \leftarrow T_P$

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 † denotes states that can cache a block at the cluster cache. T_X denotes the set of tasks sharing a block in state X. The model disallows any transition absent from the diagram.

reason about read-only, shared, or private memory blocks during each interval, our memory model lets software achieve the behavior of a coherent design without hardware cache coherence. The task-centric memory model defines a coherence domain as a logical grouping of memory blocks for which the memory model provides coherence guarantees collectively. The model requires that software perform 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 (clean) state during an interval, it can't transition to another coherence domain until after a global synchronization point is reached.

Coherence algorithm

Figure 4 depicts the state machine that a block of memory follows. 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 (WB), and invalidate operations (INV). Write-back operations write a line back to the global cache if present in the cluster cache and mark the line unmodified at the cluster cache. Invalidate operations make a line invalid in the cluster cache if present. Each set T_x represents the collections of tasks sharing a block in a particular state: clean (T_C), globally coherent (T_G), immutable (T_I), and private (T_P). We broke the private domain into two states for clarity.

We defined six properties for the memory model:

1. All blocks start in the clean state. $(\emptyset \equiv T_I \cup T_G \cup T_P \mid \text{time} = 0)$
2. A barrier is a global point of synchronization. All memory operations performed before a barrier must be complete before any processor leaves the barrier.
3. Blocks can only transition between accessible states by first passing through the clean state, after a barrier is reached.
4. A block can be in only one (nonclean) 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 $\|T_I\| \equiv 1$.
6. Loads (G.LD) that target a block in the globally coherent state return the last write to that location. All cores in the system see the same ordering of updates to that location—that is, 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 doesn't have its dirty bit set. The cluster cache controller can 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 hadn't occurred because the core issuing the load holds sole ownership of the block (by properties 4 and 5). Global atomics are restricted to only be performed on 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 disallowed by software that obeys the memory model.

Memory ordering

We define ordering of memory operations separately for operations performed within distinct coherence domains. Ordering must be defined when conflicting accesses exist. We define a conflict 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, by definition, never have conflicting accesses. Blocks in the clean and immutable states have a single value that is visible to all cores in the system.

Property 5 of the memory model ensures that updates to private blocks are only visible to a single core; therefore, no conflicting accesses can 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. Therefore, blocks in the private state need only respect dependences implied by program order. The model disallows accesses to private blocks between cores.

Conflicts can occur for blocks in the globally coherent state. We define the ordering of all accesses to all blocks in the globally coherent state to conform to processor consistency.[8] A stricter model for globally coherent data is necessary to allow accesses to that data to be used as synchronization primitives when necessary. When used as synchronization variables, globally coherent data can impose a partial order on memory operations across coherence domains. Property 2 defines a global ordering of accesses at barriers. 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 ensures 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. The ISA provides a memory fence operation to ensure that all memory operations, including write backs and invalidates, issued by a core executing the fence complete before the fence retires. No new memory operation may be initiated until after the fence has retired. All cores issuing a memory fence prior to entering a global synchronization barrier can construct a global memory fence.

Optimizations

The task-centric memory model provides 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. We can address many of these issues by extending the baseline model. (This article briefly covers some of the issues, but we leave further analysis to future work.)

We evaluated different policies for deciding when to perform coherence actions before an interval ends. Further optimization can be performed by taking a thread-centric view of coherence management—that is, 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 task completion. As an example, we can weaken property 3 with this addition: only blocks that undergo state transition across the barrier need to be clean at barriers. Although the general problem of determining what data can 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, we can exploit locality by augmenting an underlying model assumption that a task maps to a single core. We can perform optimization 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, we can now use a low-level cache, such as the cluster cache on Rigel, as the point of data coherence. Doing so allows for data that would otherwise have to exist in the globally coherent state, thus suffering high latency to access the furthest hierarchy level, to be effectively privatized to more local caches when all tasks accessing the data can be colocated as part of a group.

Our model supports staged porting of applications initially developed assuming full hardware cache coherence and porting efforts starting from a sequential implementation. To do so, we initially use the globally coherent state for all application data to provide the appearance that all data is kept coherent at all times. Although we pay a performance penalty for this due to the restrictions on local caching, the assumption of coherence holds and thus supports correctness for ported software. With a correct implementation on the new platform serving as a baseline, we can modify software to use other states in the memory model to improve performance by relaxing coherence guarantees as needed.

Evaluation

We evaluated the task-centric memory model using an implementation of the Rigel task model running on an execution-driven 1,024-core simulator of the Rigel accelerator. The results show that the overhead of software-enforced coherence, compared to an optimistic hardware-coherent baseline, is less than 10 percent in most cases and that eager coherence actions can even improve performance in other cases by reducing the instantaneous bandwidth demands placed on the system at barriers.

A naïve memory model implementation, which strictly adheres to task-centric actions, requires that numerous write backs and invalidates occur at task completion. The added memory traffic at the start and end of each task can lead to poor bandwidth utilization. Due to queuing delays in the network and

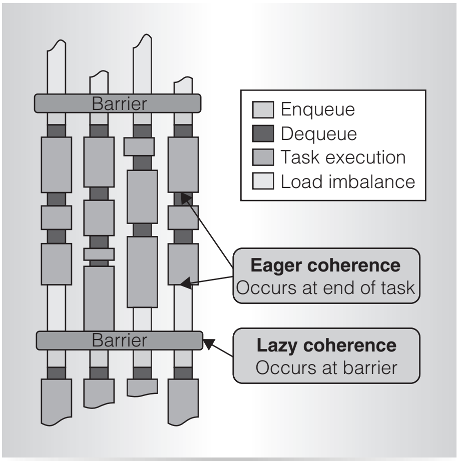


Figure 5. Timing for coherence actions.

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 are aggressively invalidated from the shared cluster caches. The combination of these effects led us to explore alternative policies for scheduling coherence actions.

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 evaluated combinations of lazy and eager policies for the write-back and invalidate components of coherence management. Figure 5 shows where these actions occur with respect to task execution. We used 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 write back (EIEW); lazy invalidate, eager write back (LIEW); eager invalidate, lazy write back (EILW); and lazy invalidate, lazy write back (LILW) relative to the optimistic baseline.

These results show that different policies provide the best performance for each benchmark. Because the model is under software control, using the best policy on a

Accelerator memory models

Compute accelerators have developed an array of memory models that emphasize compute density and parallel scalability due to the lack of legacy software constraining their design and the high degree of parallelism inherent in their workloads. Many of the prevalent models exploit the existence of coarse-grained synchronization and relative lack of fine-grained sharing in those workloads. Accelerators have achieved success relying on software for handling coherence actions and allow relaxed memory orderings, thus aiding hardware scalability and performance density. Here we survey memory models and programming models for parallel systems and compare the models with our Task-Centric Memory Model approach. With the increased interest in accelerator platforms, such as GPUs, for general-purpose computation, we see an opportunity for memory models that are less reliant on hardware to become widespread as core counts continue to rise and the distinction between chip multiprocessor (CMP) and accelerator begins to blur.

This work targets systems with a single address space and hardware-managed caches without hardware-managed coherence. Our approach contrasts with that of existing accelerators using software-managed scratchpads[1,2] or designs more similar to contemporary CMPs where caches are kept coherent transparent to software.[3] Leverich et al. investigate the implications of choosing between two different memory system configurations, hardware-coherent caches and software-managed scratchpads for future CMPs, and demonstrate that software coherence actions can provide benefit to cached systems.[4] The incoherent software-based approach, a third choice not investigated in that work, is most similar to our model. Furthermore, prototype systems with hardware caches, but without hardware coherence, such as Cedar,[5] have been built. These same techniques are being reapplied to accelerator systems today, such as the Rigel Accelerator,[6] which we used as the basis for our work.

Example parallel models for CMPs, such as Intel's threaded building blocks (TBB)[7] and Cilk,[8] use explicit task generation. These models allow for interactions between tasks and make use of parent-child communication through shared memory, which relies on the existence of a coherent address space. Underlying many of the models used by accelerators is the bulk-synchronous parallel (BSP) model.[9] BSP is reflected in accelerator languages prevalent today, including CUDA[10] from NVIDIA and OpenCL,[11] which are used to map data-parallel kernels to highly parallel systems comprising possibly hundreds of processing elements in a bulk-synchronous fashion. Although CMPs continue to support unrestricted sharing patterns and accelerators adopt share-nothing programming models, we see a potential for an intermediate design point that exploits the structure of accelerator applications by using a software-protocol and minimal hardware to provide the programmability afforded by CMPs while achieving the scalability of accelerators.

Rigel's memory model and coherence mechanisms are akin to software coherence mechanisms used to provide the illusion of a single address space for distributed shared-memory (DSM) systems.[12,13] Two DSMs, Midway[14] and Munin,[15] use flexible consistency models to achieve parallel scalability. Midway allowed for a high degree of latency tolerance by associating individual data items with synchronization operations and only guaranteeing that the data was visible after the acquiring the associated synchronization operation while also supporting multiple consistency models concurrently in one program. Munin is based on data types specified by the programmer that allows for communication-based per-type optimizations to be

per-application basis is straightforward. In general, we found four trends. First, eager write backs overlap write traffic with useful execution and should be used as much as possible to increase memory system concurrency. The coherence actions result in a less bursty load on the interconnect, increasing performance. 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. Third, eager invalidations can provide benefit to some benchmarks by aggressively removing data from the caches that is not used again, such as read-once input data, thus enabling a better replacement policy. Lastly, lazy write backs achieve

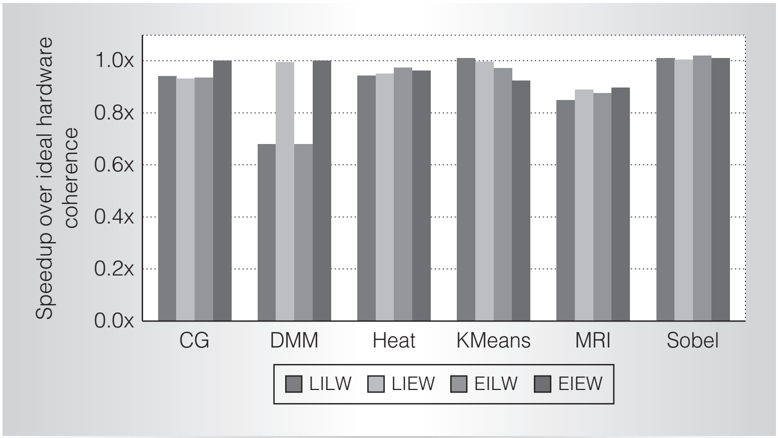


Figure 6. Speedup of the four combinations of eager/lazy and invalidation/write-back policies relative to zero-cost coherence.

exploited by the runtime. The consistency guarantees we investigate for write-output data at Rigel Task Model task boundaries are similar to Scope Consistency,[16] which makes dirty data implicitly coherent at the end of the task's scope and can defer updates until the scope is reopened. The Cooperative Shared Memory model[17] is similar to Rigel; it relies on software to properly label shared accesses for performance and achieves scalable performance using a reduced-complexity hardware coherence protocol (Dir$_1$SW).

References

1. M. Gschwind, ''Chip Multiprocessing and the Cell Broadband Engine,'' *Proc. 3rd Conf. Computing Frontiers,* ACM Press, 2006, pp. 1-8.
2. E. Lindholm et al., ''NVIDIA Tesla: A Unified Graphics and Computing Architecture,'' *IEEE Micro,* vol. 28, no. 2, pp. 39-55, 2008.
3. L. Seiler et al., ''Larrabee: A Many-Core x86 Architecture for Visual Computing,'' *ACM Trans. Graphics,* vol. 27, no. 3, 2008, article no. 18.
4. J. Leverich et al., ''Comparing Memory Systems for Chip Multiprocessors,'' *Proc. Int'l Symp. Computer Architecture* (*ISCA 07*), ACM Press, 2007, pp. 358-368.
5. D. Gajski et al., ''Cedar: A Large Scale Multiprocessor,'' *SIGARCH Computing Architecture News,* vol. 11, no. 1, pp. 7-11, 1983.
6. J.H. Kelm et al., ''Rigel: An Architecture and Scalable Programming Interface for a 1000-Core Accelerator,'' *Proc. Int'l Symp. Computer Architecture* (*ISCA 09*), ACM Press, 2009, pp. 140-151.
7. J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multicore Processor Parallelism,* O'Reilly, 2007.
8. M. Frigo, C.E. Leiserson, and K.H. Randall, ''The Implementation of the Cilk-5 Multithreaded Language,'' *SIGPLAN Notices,* vol. 33, no. 5, 1998, pp. 212-223.
9. L.G. Valiant, ''A Bridging Model for Parallel Computation,'' *Comm. ACM,* vol. 33, no. 8, 1990, pp. 103-111.
10. J. Nickolls et al., ''Scalable Parallel Programming with CUDA,'' *Queue,* vol. 6, no. 2, 2008, pp. 40-53.
11. *OpenCL Specification,* 1st ed., Khronos OpenCL Working Group, 2008.
12. C. Amza et al., ''Treadmarks: Shared Memory Computing on Networks of Workstations,'' *Computer,* vol. 29, no. 2, 1996, pp. 18-28.
13. D. Scales et al., ''Shasta: A Low Overhead, Software-Only Approach for Fine-Grain Shared Memory,'' *Proc. 7th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (*ASPLOS 96*), 1996, pp. 174-185.
14. B. Bershad, M. Zekauskas, and W. Sawdon, ''The Midway Distributed Shared Memory System,'' *Compcon Spring '93, Digest of Papers.,* Feb 1993, pp. 528-537.
15. J.K. Bennett, J.B. Carter, and W. Zwaenepoel, ''Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence,'' *Proc. 2nd ACM SIGPLAN Symp. Principles & Practice of Parallel Programming* (*PPoPP 90*), ACM Press, 1990, pp. 168-176.
16. L. Iftode, J.P. Singh, and K. Li, ''Scope Consistency: A Bridge Between Release Consistency and Entry Consistency,'' *Proc. 8th Ann. Symp. Parallel Algorithms and Architectures* (*SPAA 96*), ACM Press, 1996, pp. 277-287.
17. M.D. Hill et al., ''Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors,'' *ACM Trans. Computer Systems,* vol. 11, no. 4, 1993, pp. 300-318.

similar benefits to eager write backs, but across intervals by aggressively flushing data out of the cache at a barrier instead of waiting until a future conflict in the next interval causes the eviction.

We find that our task-centric memory model can efficiently provide a single shared address space on a system with caches, but without hardware coherence. There's still potential for improving on these efforts by adding hardware support tracking of sharing at the local caches to avoid unnecessary cache management operations. Moreover, we see opportunity for software tools support the model. Such tools could extract sharing information from applications written using parallel languages and orchestrate the memory model implicitly, thus obviating the need for explicit programmer direction. MICRO

Acknowledgments

We acknowledge the support of the Focus Center for Circuit & Systems Solutions (C2S2) and the Gigascale Systems Research Center (GSRC), two of the five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation Program. We thank the Trusted ILLIAC Center at the Information Trust Institute for their generous contribution of use of their computing cluster to help us complete our research. We also thank Naveen Neelakantam, Matt R. Johnson, Aqeel Mahesri, and the anonymous reviewers for their input and feedback. John Kelm was partially supported by a fellowship from Advanced Micro Devices.

References

1. NVIDIA, ''NVIDIA GeForce 8800 GPU Architecture Overview,'' Nov. 2006.
2. L. Seiler et al., ''Larrabee: A Many-Core x86 Architecture for Visual Computing,'' *ACM Trans. Graphics,* vol. 27, no. 3, 2008, article no. 18.
3. L.G. Valiant, ''A Bridging Model for Parallel Computation,'' *Comm. ACM,* vol. 33, no. 8, 1990, pp. 103-111.
4. C. Amza et al., ''Treadmarks: Shared Memory Computing on Networks of Workstations,'' *Computer,* vol. 29, no. 2, 1996, pp. 18-28.
5. J.K. Bennett, J.B. Carter, and W. Zwaenepoel, ''Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence,'' *Proc. 2nd ACM SIGPLAN Symp. Principles & Practice of Parallel Programming* (*PPoPP 90*), ACM Press, 1990, pp. 168-176.
6. A. Mahesri et al., ''Tradeoffs in Designing Accelerator Architectures for Visual Computing,'' *Proc. Int'l Symp. Microarchitecture,* 2008, ACM Press, pp. 164-175.
7. J.H. Kelm et al., ''Rigel: An Architecture and Scalable Programming Interface for a 1000-Core Accelerator,'' *Proc. Int'l Symp. Computer Architecture,* 2009, ACM Press, pp. 140-151.
8. J. Goodman, ''Cache Consistency and Sequential Consistency,'' tech. report 61, SCI Working Group, 1989.

John H. Kelm is a PhD candidate at the University of Illinois at Urbana-Champaign. His research interests include parallel architectures, memory system design, and cache coherence protocols. Kelm has an MS in computer engineering from the University of Illinois. He is a member of the ACM and IEEE.

Daniel R. Johnson is a PhD candidate at the University of Illinois at Urbana-Champaign. His research interests include parallel accelerators and domain-specific architectures. Johnson has a BS in electrical engineering from the University of Texas at Austin. He is a member of the ACM and IEEE.

Matthew I. Frank is a software engineer in the Performance, Analysis, and Threading Lab at Intel. His technical interests include programming tools for parallel programming. Frank has a PhD in computer science from the Massachusetts Institute of Technology.

Steven S. Lumetta is an associate professor of Electrical and Computer Engineering at the University of Illinois at Urbana-Champaign. His interests include high-performance networking and computing, hierarchical systems, and parallel runtime software. Lumetta has a PhD in computer science from the University of California, Berkeley. He is a member of the ACM and IEEE.

Sanjay J. Patel is an associate professor of Electrical and Computer Engineering and Sony Faculty Scholar at the University of Illinois at Urbana-Champaign. His interests are in high-throughput chip architectures and visual computing. Patel has a PhD in computer science and engineering from the University of Michigan, Ann Arbor. He is a member of the IEEE.

Direct questions and comments to John H. Kelm, Univ. of Illinois at Urbana-Champaign, 1306 W. Main St., Urbana, IL 61801; jkelm2@illinois.edu.