Cohesion: An Adaptive Hybrid Memory Model for Accelerators

Cohesion is a hybrid memory model that enables fine-grained temporal data reassignment between hardware- and software-managed coherence domains, allowing systems to support both. Cohesion can dynamically adapt to the sharing needs of both applications and runtimes requiring neither copy operations nor multiple address spaces.

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

•••••• Chip multiprocessors (CMPs) supporting four to 32 hardware threads are available commercially; compute accelerators, such as graphics processing units (GPUs), already support thousands of concurrent hardware threads. However, a clear divergence exists between the memory systems of general-purpose CMPs and domain-specific GPUs. While CMPs continue to espouse strict memory consistency models with a single address space and hardware cache coherence, GPUs support relaxed memory orderings, multiple address spaces, and software-managed coherence. As CMP core counts increase, a tension grows between the desire to support programming paradigms that rely on a hardware-coherent address space and the difficulty of scaling hardware cache coherence.

In our paper for the 37th Annual International Symposium on Computer Architecture, we presented Cohesion, a hybrid memory model that bridges the gap between accelerators' weak, easily scalable models and CMPs' strict, more difficult to scale models.[1] Cohesion uses a hybrid hardware/software coherence scheme. When possible, Cohesion performs coherence management in software at coarse granularity, or avoids it completely. For accesses that a weaker model would find prohibitively expensive, Cohesion incorporates hardware-supported coherence regions. Cohesion achieves scalability by exploiting scalable parallel applications' lack of fine-grained sharing, applying hardware coherence techniques when advantageous, and deferring to software-managed coherence when possible (such as in the cases of private, immutable, and read-shared data). (For more information, see the "Related work in accelerator systems" sidebar.)

Cohesion has three key benefits:

- a system with a single address space that supports various coherence domains, which are regions of memory for which the memory model collectively provides coherence guarantees;
- the ability to perform temporal reallocation between coherence domains without copies; and
- fine granularity—interleaving data managed using a software protocol and a hardware protocol at a cache-line granularity.

Cohesion aims to enable scalability by reducing reliance on hardware mechanisms without sacrificing the conventional shared-memory programming model.

Published by the IEEE Computer Society

Related work in accelerator systems

Distributed shared memory provides the illusion of a single, coherent address space across processors—in TreadMarks,[1] at a coarse grain using virtual memory and a runtime system, and in Shasta,[2] at a fine grain using compiler support. While these approaches' distributed memory architecture could trivially support incoherence, they can synthesize coherence when needed fully in software. Cooperative Shared Memory (CSM) uses software hints to reduce hardware-supported coherence's complexity and cost.[3] Software-assisted hardware schemes, such as LimitLESS,[4] trap to software when the number of sharers the hardware supports is exceeded. CSM and LimitLESS suffer from high round-trip latency between directory and cores in a hierarchically cached system and require that the coherence protocol tracks all data, resulting in unnecessary traffic for some data.

Several previous works on distributed memory multiprocessors investigate hybrid schemes that combine message passing with hardware-managed coherent shared memory. Flash used programmable protocol controllers that support customized protocols.[5] Cohesion provides a mechanism to allow such customization without a separate protocol controller. Munin used parallel program access patterns to provide different consistency guarantees to different classes of data.[6] Multiphase Shared Arrays let the programmer specify access modes for array data and change the mode for different program phases.[7] Cohesion on an integrated, shared-memory multiprocessor captures these features with modest hardware and an intuitive programming model that doesn't require a message-passing component. Researchers have proposed a hybrid x86 system with a consistent programming model for GPU and CPU cores using an address space subset.[8] Unlike Cohesion, that work doesn't investigate dynamic transitions between coherence domains. Furthermore, it focuses solely on a host-accelerator model with one CPU core,[8] while we demonstrated Cohesion using 1,024 cooperating cores.

References

1. C. Amza et al., ''Treadmarks: Shared Memory Computing on Networks of Workstations,'' *Computer,* vol. 29, no. 2, 1996, pp. 18-28.
2. D.J. Scales, K. Gharachorloo, and C.A. Thekkath, ''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,* ACM Press, 1996, pp. 174-185.
3. 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.
4. D. Chaiken, J. Kubiatowicz, and A. Agarwal, ''LimitLESS Directories: A Scalable Cache Coherence Scheme,'' *Proc. 4th Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* ACM Press, 1991, pp. 224–234.
5. J. Kuskin et al., ''The Stanford Flash Multiprocessor,'' *Proc. 21st Ann. Int'l Symp. Computer Architecture,* IEEE CS Press, 1994, pp. 302-313.
6. 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,* ACM Press, 1990, pp. 168-176.
7. J. De Souza and L.V. Kal, ''MSA: Multiphase Specifically Shared Arrays,'' *Proc. 17th Int'l Workshop Languages and Compilers for Parallel Computing,* LNCS 3602, Springer, 2005, pp. 268-282.
8. B. Saha et al., ''Programming Model for a Heterogeneous x86 Platform,'' *Proc. 2009 ACM Sigplan Conf. Programming Language Design and Implementation,* ACM Press, 2009, pp. 431-440.

Motivation

Cohesion demonstrates that software- and hardware-managed coherence both have overheads that a hybrid memory model can mitigate. Cohesion also shows promise from a programmability and optimization perspective. We find a compelling use case in heterogeneous processors with general-purpose and accelerator cores with a shared address space. In such a design, in which the coherence needs and capabilities vary by core, a hybrid approach could reduce the need for data marshalling and the cost of data copies.

The case for software cache coherence

A software-managed coherence protocol embedded in the compiler,[2] runtime,[3] or programming model[4,5] avoids hardware coherence management overheads. Directories[6] and duplicate tags[7] require no memory, and implementing and verifying the coherence protocol expends no design effort. The network experiences less traffic and relaxed design constraints. Furthermore, fine-grained software protocols can eliminate false sharing.

We simulated a 1,024-core system running multithreaded kernels that use a barrier-based task queue programming model. Figure 1 shows the messaging overhead for an optimistic implementation of hardware coherence (HW_{cc}). For this experiment, we assume a complete full-map on-die directory with infinite capacity that

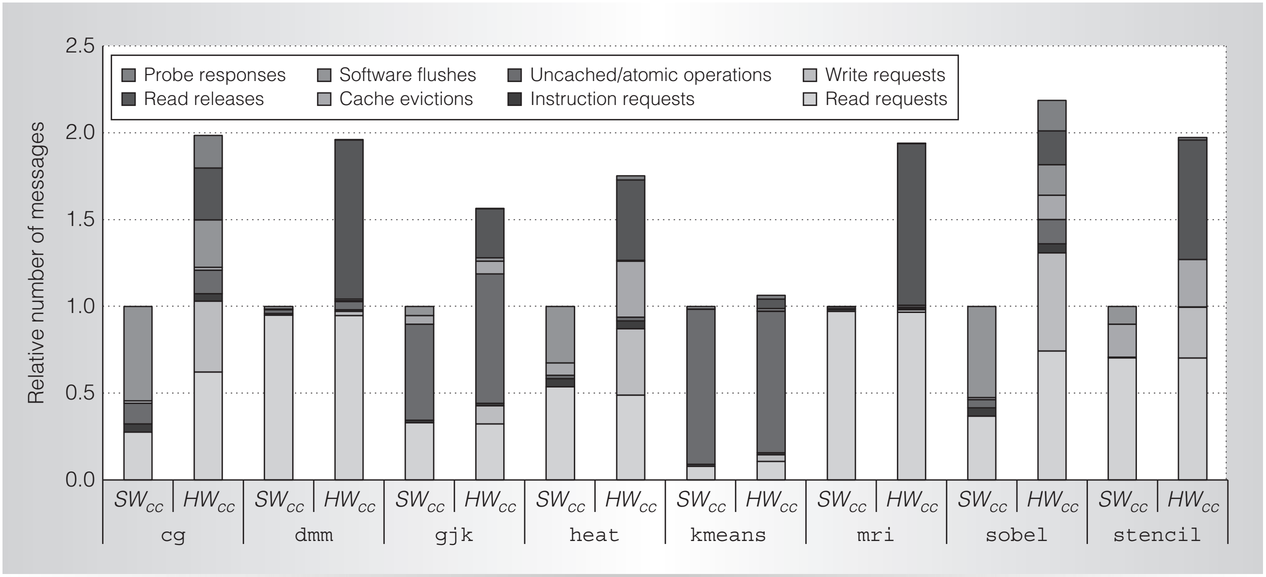

Figure 1. Number of messages sent by the local caches shared by eight cores (at level two, or L2) to the global shared last-level cache (L3) for software coherence (SW_{cc}) and optimistic hardware coherence (HW_{cc}). Results are normalized to SW_{cc}.

eliminates directory evictions and broadcasts. The software-managed protocol (SW_{cc}) uses explicit flush and invalidation instructions inserted by software to manage coherence.

Figure 1 shows significantly increased message traffic across all benchmarks for HW_{cc} except for `kmeans`, which is dominated by atomic read-modify-write histogramming operations. The additional messages come primarily from two sources: write misses and read release or invalidation operations. For SW_{cc}, software must track ownership. For a noninclusive cache hierarchy, the cache maintains per-word dirty or valid bits. The system can issue writes as write-allocates under SW_{cc} without waiting on a directory response. Also, under HW_{cc}, the system doesn't support silent evictions, and therefore issues read releases to the directory when it evicts a clean line from a local cache (L2). For SW_{cc}, there's no need to send any message, and the invalidation occurs locally. Even if the system used a protocol without read releases, HW_{cc} would still show significant message overhead for invalidation cache probes.

SW_{cc} can mass invalidate shared read data, signaling many invalidations with only a few messages. A global synchronization event, such as a barrier, helps coordinate SW_{cc} actions when necessary. The equivalent operation in hardware requires sending potentially many invalidation messages exactly when a state transition or eviction is needed. Such an invalidation mechanism lengthens coherence actions' critical path, increases contention for the network, and requires greater capacity from the network to handle the invalidation traffic. SW_{cc} can eliminate false sharing because multiple write sharers that access disjointed sets of words on a line won't generate coherence probes that would otherwise cause the line to ping-pong between sharers in HW_{cc}.

The case for hardware cache coherence

Hardware cache coherence provides several programmability benefits. HW_{cc} can enforce strict memory models whereby hardware ensures that all reads receive the latest write to a word in memory, which simplifies reasoning about sharing in some applications. HW_{cc} enables speculative prefetching and data migration. Developers can port shared memory applications to an HW_{cc} design without a full rewrite, albeit with possibly degraded performance.

SW_{cc} is a push mechanism; explicit actions must occur to make data modified by one sharer visible to other sharers.

On the other hand, HW_{cc} is a pull mechanism that lets requesters locate the latest copy of data on demand. The implication is that SW_{cc} protocols might be too conservative, pushing all data that another core might read to a globally visible point—for example, memory or a globally shared last-level cache. Furthermore, the additional traffic required for read-release messages under HW_{cc} composes a significant portion of message traffic; these messages aren't on the critical path for a waiting access, whereas a directory-sent invalidation is.

SW_{cc} reduces instruction stream efficiency by introducing explicit cache many flush instructions, which show up as additional software flush messages in Figure 1. Under a software protocol with deferred coherence actions, the state of lines must be tracked in memory, which can be less efficient than maintaining a small number of bits with each cache tag—the conventional method for tracking the coherence state. Furthermore, this wastes flush instructions, because the cache might already have evicted the targeted lines by the time the SW_{cc} actions occur. Our results show that many dynamically issued coherence instructions are superfluous, operating on lines not present in the cache.

A hybrid memory model

Supporting multiple coherence implementations means that software can dynamically select the appropriate mechanism for memory blocks. Supporting incoherent memory regions allows more scalable hardware by reducing the number of shared lines, resulting in fewer coherence messages and less directory overhead for tracking the sharer state. Furthermore, coherence lets developers make trade-offs regarding software design complexity and performance. A hybrid memory model provides a runtime mechanism for managing coherence needs across applications, even if modified applications don't need data to frequently transition between SW_{cc} and HW_{cc}.

From the system software's perspective, HW_{cc} has many benefits. It allows for migratory data patterns that SW_{cc} doesn't easily support. Threads that sleep on one core and resume execution on another must have their local modified stack data available, forcing coherence actions at each thread swap under SW_{cc}. Likewise, coherence aids task-based programming models.[8,9] HW_{cc} lets children tasks be scheduled on their parent, incurring no coherence overhead, or lets another core steal them, causing data to be pulled using HW_{cc}.

The assortment of cores on emerging systems-on-a-chip makes supporting multiple-memory models on the same die attractive. A hybrid approach lets cores without HW_{cc} support (such as accelerator cores) cooperate with cores that have HW_{cc} support and interface with coherent general-purpose cores. While heterogeneous systems don't require a single address space (as demonstrated by the Cell processor[10] and GPUs), it could aid in portability and programmability by extending the current shared memory model to future heterogeneous systems. A hybrid approach lets developers leverage HW_{cc} for easier application porting from conventional shared memory machines and easier debugging for new applications. They can then use SW_{cc} to reduce the stress on the hardware coherence mechanisms to improve performance. Table 1 lists many of the trade-offs between software- and hardware-managed coherence.

Design

Cohesion provides hardware support and a protocol that lets data migrate between coherence domains at runtime, with fine granularity, and without the need for copy operations. Cohesion's default behavior is to keep all memory coherent in the HW_{cc} domain. Software can alter the default behavior by modifying tables in memory that control how the system enforces coherence. No hardware coherence management applies to unshared data or to data for whom software can handle coherence at a coarse granularity using the SW_{cc} domain.

Baseline architecture

Figure 2 shows a diagram of our baseline 1,024-core accelerator (a variant of the Rigel architecture[11]). The processor has a single address space and three cache levels. Each core is a simple, in-order processor with private L1 instruction and data caches and a RISC-like instruction set. Eight cores form a

Table 1. Trade-offs for HW_{cc}, SW_{cc}, and Cohesion.

Approach	Programmability	Network constraints	On-die storage
HW_{cc}	Conventional CMP shared-memory paradigm; supports fine-grained, irregular sharing without relying on compiler or programmer for correctness.	Hardware (rather than extra instructions and coherence traffic) handles potential dependences.	Optimized for HW_{cc}; when desired, stores coherence data efficiently.
SW_{cc}	Used in accelerators; provides programmer and compiler control over sharing.	Eliminates probes and broadcasts for independent data, such as stack, private, and immutable data.	Optimized for SW_{cc}; minimal hardware overhead beyond hardware-managed caches.
Cohesion	Supports HW_{cc} and SW_{cc}; clear performance optimization strategies allowing $SW_{cc} \Leftrightarrow HW_{cc}$ transitions.	SW_{cc} used to eliminate traffic for coarse-grained and regular sharing patterns; HW_{cc} for unpredictable dependences.	Reduces pressure on HW_{cc} structures; enables hardware design optimizations based on HW_{cc} and SW_{cc} needs.

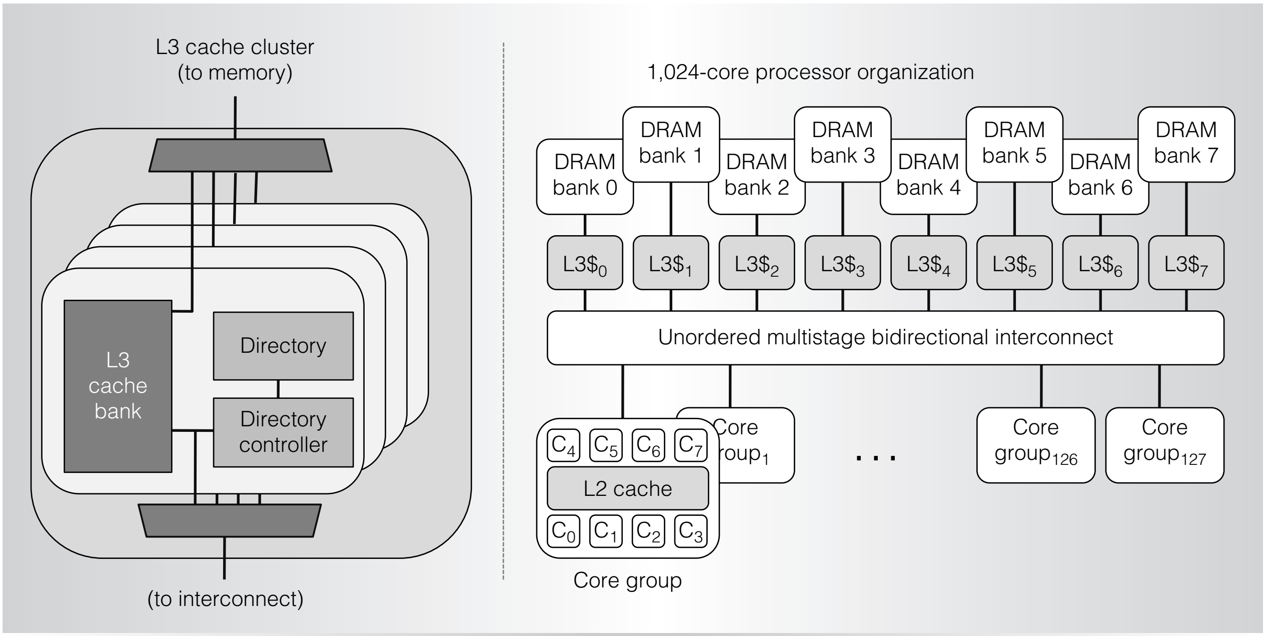

Figure 2. Baseline processor architecture. The processor comprises 1,024 simple, in-order RISC-like cores organized in a hierarchical configuration. The memory system has three levels of cache and eight independent Graphics Double Data Rate, version 5 (GDDR5) memory controllers.

cluster and share a unified L2 cache. A single, multibanked, shared last-level L3 cache serves as the chip's coherence point.

Figure 3 shows the hardware protocol (on the right). Data in the HW_{cc} domain uses a Modified, Shared, or Invalid (MSI) protocol. We employ sparse directories where the directory only holds directory entries for lines present in at least one L2. The baseline architecture is noninclusive between L2 and L3 caches. The directory is inclusive of the L2s and thus may contain entries for lines not in the L3 cache. One bank of the directory is attached to each L3 cache bank. All directory requests are serialized through a home directory bank. Associating each L3 bank with a slice of the directory allows the two mechanisms to be collocated, reducing the complexity of the protocol implementation.

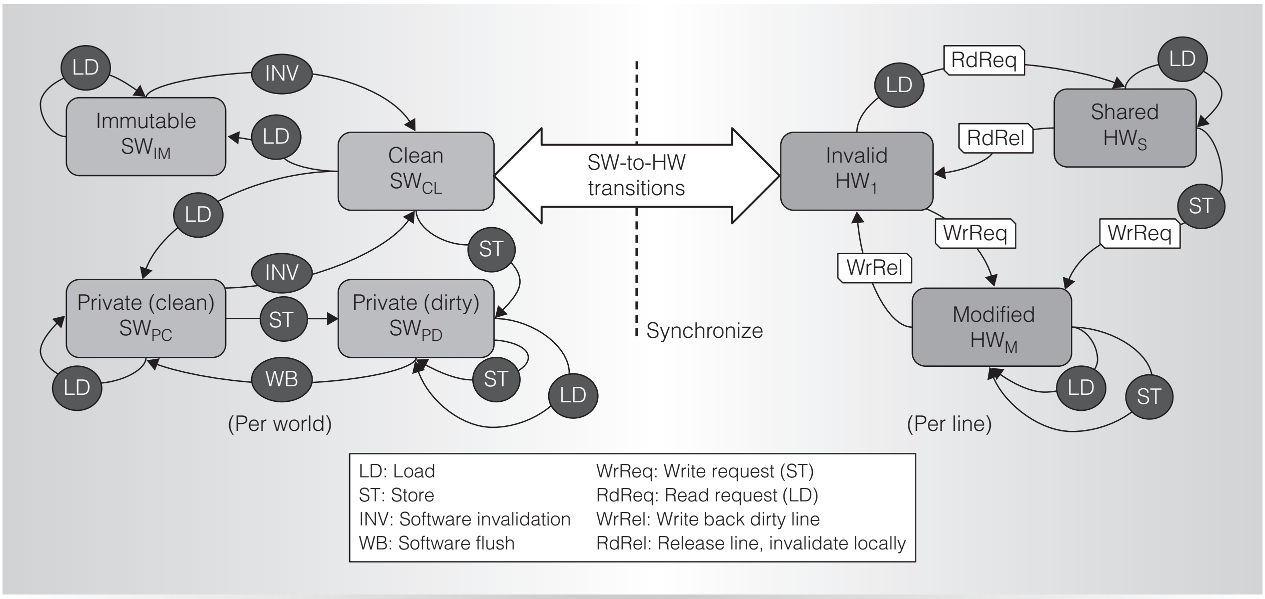

Figure 3. Cohesion as a protocol bridge. Software-managed coherence is on the left, and the hardware coherence protocol is on the right. Cohesion lets blocks of memory transition between the two coherence domains.

Our software coherence protocol is the Task-Centric Memory Model,[5] modified to support hybrid coherence (see the left side of Figure 3). The protocol leverages the bulk-synchronous (BSP) compute pattern. BSP comprises phases of mostly data-parallel execution followed by communication, with barriers separating each phase. The software protocol exploits the fact that most data isn't read-write shared across tasks between two barriers, and most intertask communication occurs across barriers.

The software protocol provides a set of state transitions (initiated explicitly by the software or implicitly by the hardware) that let programmers or compilers reason about coherence for a block of data. In a system that uses cached memory for communication, but without hardware coherence, hardware could implicitly move data into a globally visible location. The software can't control these implicit actions, so the SW_{cc} protocol—and, by extension, Cohesion—must account for them.

Cohesion: a hybrid memory model

Cohesion achieves hybrid coherence by tracking the coherence domain to which regions of memory belong and orchestrating coherence domain transitions. As Figure 4 shows, the system comprises a directory for tracking shared HW_{cc} lines, a coarse-grained region table for tracking common large memory regions that are SW_{cc}, and a fine-grained region table for tracking the remaining memory that could transition between HW_{cc} and SW_{cc}. Cohesion adds one bit of state per line (the incoherent bit) to the L2 cache to track which cached lines aren't HW_{cc}.

Cohesion queries the directory when a request arrives at the L3. If the line is a directory hit, the line is in HW_{cc} and the directory handles the response. If the requester can access the line, the directory accesses the L3 in the next cycle and returns the response to the requesting L2. A directory hit with an L3 miss results in the directory blocking access. The directory generates a response when the fill from memory occurs. A directory miss results in the region tables being examined.

The coarse-grained region table is a small on-die structure that contains a map of SW_{cc} address ranges and is accessed in parallel with the directory. The three most frequently used regions are for code, private stacks, and persistent globally immutable data. When an access misses in the directory and the address maps into one of these ranges, the L3

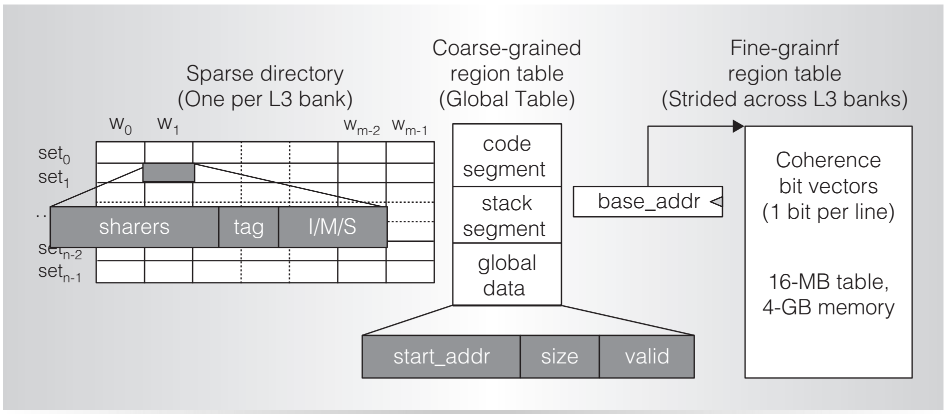

Figure 4. Cohesion architecture. The additions over a baseline hardware-coherent processor are region tables for tracking coherence domain state and additional logic for controller coherence domain transitions.

cache controller responds with the data. The message includes a bit signaling to the L2 that an incoherent access has occurred. When the response arrives, the L2 sets the incoherent bit in the L2 cache tag for the line. Under SW_{cc}, if the line is invalidated by software or evicted while in the clean state, the line is simply dropped, and the L2 doesn't send a message to the L3.

The fine-grained region table receives queries for all other accesses. The table might be cached in the L3 because the L3 is outside of the coherence protocol, which only applies between the L2 caches. We map all of memory using one bit per cache line. If the bit in the table is set, the L3 responds with the data and sends the incoherent bit along with the message. If the bit is cleared, an entry for the corresponding line is placed into the directory. The line is returned to the requester and thereafter is kept hardware coherent. Should a directory eviction occur, followed by a future access to the table, the directory entry will be reinserted.

Software interface to Cohesion

Here, we discuss Cohesion's application programming interface. When the application loads, the runtime initializes Cohesion's region tables. The coarse-grained SW_{cc} regions are set for the code segment, the constant data region, and the per-core stack region. The code segment and constant data address ranges reside in the application binary's executable and linkable format (ELF) header. Our architecture doesn't support self-modifying code, so HW_{cc} isn't required for cached instructions. Variable-sized stacks are possible by treating the stack region as a SW_{cc} heap, but we found that fixed-sized stacks were sufficient for our implementation. Our implementation has two heaps: a conventional C-style heap that's kept coherent and another that's not kept HW_{cc} by default. We use the incoherent heap for data that could transition between coherence domains during execution.

A transition between SW_{cc} and HW_{cc} is initiated by word-aligned, uncached, read-modify-write operations performed by the runtime to the fine-grained region table. The issuing core blocks until the directory completes the transition for the purposes of memory ordering. If a request for multiple line-state transitions occurs, the directory serializes the requests line by line. All lines that could transition between coherence domains are initially allocated using our implementation's incoherent heap, and these lines have an initial state of SW_{cc}. The runtime can transition SW_{cc} (HW_{cc}) lines to HW_{cc} (SW_{cc}) by clearing (setting) the state bits in the fine-grained region table.

Coherence domain transitions

The directory controller must orchestrate the $HW_{cc} \Leftrightarrow SW_{cc}$ transitions. The directory

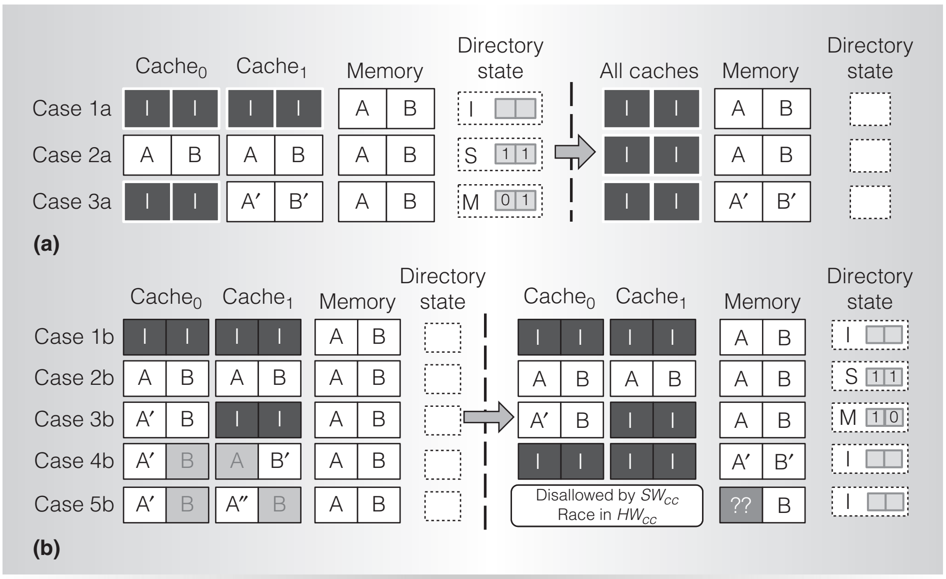

Figure 5. Potential system states on a $HW_{cc} \Leftrightarrow SW_{cc}$ transition (a) and a $SW_{cc} \Leftrightarrow HW_{cc}$ transition (b). Memory and L3 are interchangeable. Each block has two words. A and B are the initial values. On the left side, A′, A″, and B′ denote distinct modified values in the local cache. Grayed blocks are potentially valid, but software doesn't access them. The right side shows the system state after the state transition occurs.

snoops the address range for the fine-grained region table; on an access that changes a line's coherence domain, the directory performs the actions necessary to transition between SW_{cc} and HW_{cc}. The directory completes the request by sending an acknowledgment to the issuing core. Handling the transitions at the directory lets us serialize requests for a line across the system. Coherence domain transitions for a single line thus occur in a total order across the system with all other coherent and noncoherent accesses at the L3 being partially ordered by the transitions.

$HW_{cc} \Rightarrow SW_{cc}$ transitions. Moving a line out of the hardware-coherent domain requires removing any directory state associated with the line, updating the table, and putting the line in a consistent state known to software. Figure 5a shows the potential states a line can be in when software initiates a $HW_{cc} \Rightarrow SW_{cc}$ transition. Each state corresponds to a possible state allowed by the MSI directory protocol. After a transition completes, the line isn't present in any L2, and the current value is present in the L3 cache or in memory.

For Case 1a (in Figure 5a), the directory controller queries the directory and finds the entry isn't present, indicating that there are no sharers. Therefore, all the directory must do is set the bit in the region table. Case 2a has the line in the shared state with one or more sharers. The directory performs a directory eviction that invalidates all that share the line. When the directory has received all acknowledgments, it removes the directory entry, sets the bit in the region table, and returns the response to the core requesting the transition. When a line is in the modified state, as in Case 3a, some L2 has a newer version of the data. The directory sends a write-back request to the owner. When the response arrives, the directory updates the L3, modifies the table, and sends a response to the requester.

$SW_{cc} \Rightarrow HW_{cc}$ transitions. Figure 5b's left half shows two L2 caches with the potential states that a line could be in when controlled by SW_{cc}. Because the directory has no knowledge of the line state, a transition to HW_{cc} initiates a broadcast clean request from the directory. When the line is

found in an L2 in the clean state, the incoherent bit is cleared—that is, the line is now susceptible to cache probes but isn't evicted from the L2. Clean lines send an acknowledgment message to the directory, which adds the L2 to the list of sharers. If the line isn't found in the L2, the directory receives a negative acknowledgment. Cases 1b and 2b demonstrate transitions for unmodified lines.

If the system finds a dirty line, as in Cases 3b and 4b, the L2 notifies the directory. If any read sharers exist, the directory sends messages forcing all readers to invalidate the line and the owners to write back the dirty copies. If the line is dirty in only one cache, the sharer is upgraded to owner at the directory and no write-back occurs, saving bandwidth. If the system finds multiple writers, it sends write-back requests to all L2s holding modified lines, and it sends invalidations to all L2s holding clean lines. Our architecture maintains per-word dirty bits with the cache lines, letting the L3 merge multiple writers' results if the write sets are disjointed. When either operation completes, the line isn't present in any L2, and the L3 memory holds the most recent copy of the line.

The system can always force a $SW_{cc} \Rightarrow HW_{cc}$ transition to make caches consistent, such as between task swaps, but the data values might not be safe. Buggy software could modify the same line in two L2 caches concurrently when under SW_{cc}'s control (as in Case 5b, where the state represents a hardware race). To safely clean a word's state, the runtime can always turn on coherence and then zero the value. This will cause the dirty values in the separate caches to be thrown away and the zeroed value to persist. For debugging, it could be useful to have the directory signal an exception with its return message to the requesting core.

Evaluation

Compared to HW_{cc}, Cohesion has two benefits: it reduces message traffic and the on-die directory storage overhead. Cohesion shows performance comparable to pure HW_{cc}, despite optimistic assumptions for the pure model, and a large improvement over HW_{cc} with a realistic on-die directory. (For examples of high-level programming in Cohesion, see the "Programming for Cohesion" sidebar.)

Methodology

We used an execution-driven simulation of the design and ran each benchmark for at least 1 billion instructions. We structurally modeled cores, caches, interconnects, and memory controllers. The cycle-accurate dynamic RAM (DRAM) model uses Graphics Double Data Rate, version 5 (GDDR5) timing. We evaluated four design points: SW_{cc}, optimistic HW_{cc}, HW_{cc} with realistic hardware assumptions, and Cohesion with the same realistic hardware assumptions. SW_{cc}'s hardware configuration equates to our baseline, but with no directory and the software handling all sharing. For SW_{cc}, writes occur at the L2 with no delay and evictions performed to clean data don't create any network traffic. The optimistic HW_{cc} case removes all directory conflicts by making the on-die directory infinite and fully associative. The behavior equates to a full-map directory in memory,[6] but with zero cost access to the directory. The HW_{cc} realistic case comprises a 128-way sparse directory at each L3 cache bank. The Cohesion configuration uses the same hardware as the realistic HW_{cc} configurations.

Message reduction

Figure 6 shows the number of messages the L2s send to the directory, normalized to SW_{cc}. A reduction in messages occurs relative to the HW_{cc} configurations across all benchmarks. The `kmeans` benchmark is the only benchmark where SW_{cc} shows higher message counts than Cohesion. This reduction occurs because optimizations reduce the number of uncached operations that the benchmark issues by relying on HW_{cc} under Cohesion. For some benchmarks, such as `heat` and `stencil`, the number of messages is nearly identical across Cohesion and optimistic HW_{cc} configurations. We see potential to remove many of these messages by using Cohesion to apply further, albeit more complicated, optimization strategies. We leave more elaborate coherence domain remapping strategies to future work.

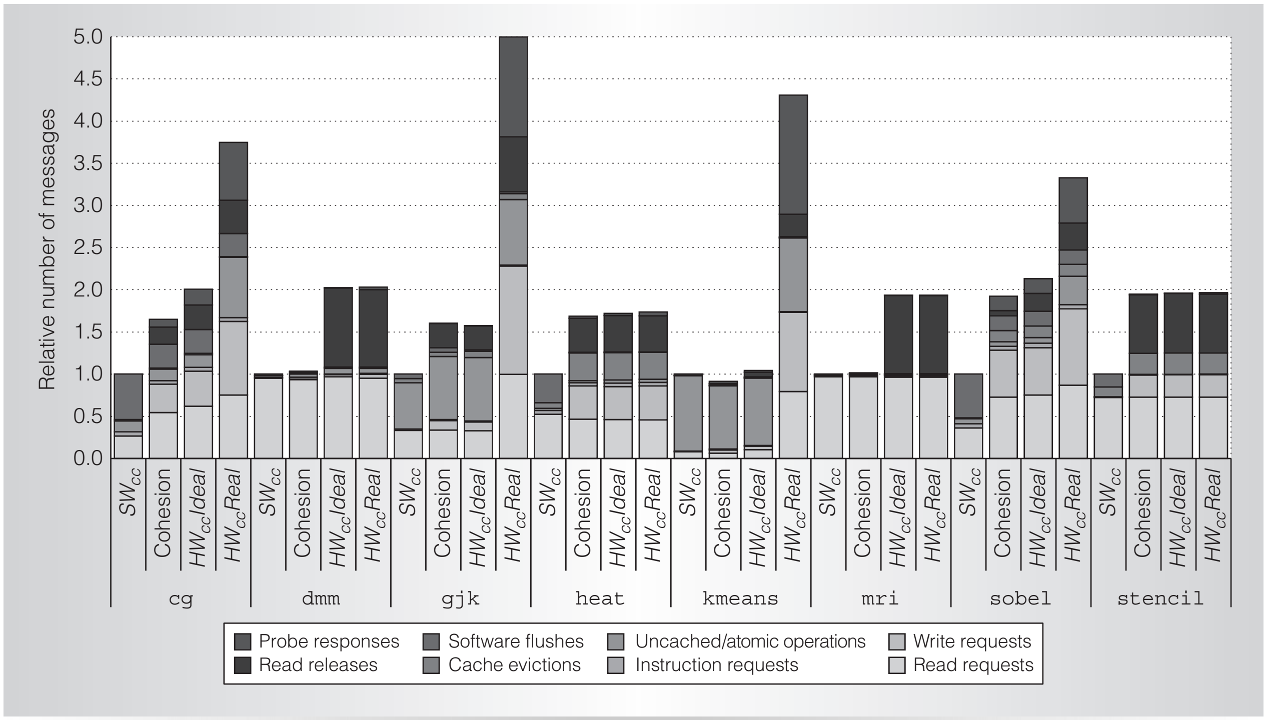

Figure 6. L2 output message counts for SW_{cc}, Cohesion, HW_{cc} with a full on-die full-map directory, and HW_{cc} with a 128-way sparse directory on-die normalized to SW_{cc}.

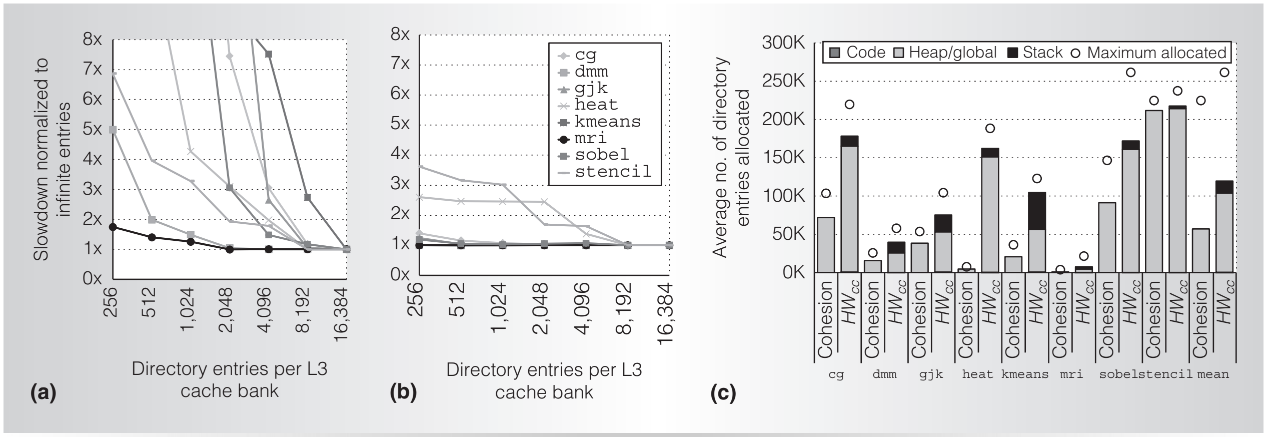

Figure 7. Slowdown for different directory sizes for HW_{cc} (a) and Cohesion (b). Performance falls off precipitously in HW_{cc}. The time average (sampled every 1,000 cycles) and maximum number of directory entries allocated for both Cohesion and HW_{cc} (c) when directories have unbounded size and associativity.

Directory entry savings

Figures 7a and 7b show the normalized runtime for different directory sizes under HW_{cc} and Cohesion, respectively, compared to an infinite directory. We make directories fully associative to isolate the capacity's influence. Figure 7a demonstrates the rapid performance drop-off for shrinking directory sizes. In Figure 7b, we show that Cohesion reduces the performance sensitivity with respect to directory sizing across all benchmarks. Figure 7c shows the mean and maximum number of directory entries that Cohesion and the optimistic HW_{cc} baseline used.

Programming for Cohesion

Accelerator systems today face a challenge. Applications with data structures or compute patterns that heavily leverage hardware cache coherence can't easily exploit available accelerators' parallel processing power, if at all. Applications that are amenable to GPU acceleration but were developed with programming models that assumed hardware cache coherence require a rewrite. A similar problem faces software that has a data-parallel kernel but requires a host application that might find coherence beneficial. Applications with shared irregular data structures that require irregular updates, such as graphs or kD-trees, might not have mechanisms to efficiently support their execution without hardware cache coherence. To make many-core accelerators and general-purpose systems efficient platforms for applications, there must be a means to support hardware cache coherence at some level.

Being able to support current programming models that require hardware coherence will open up future platforms to more developers. The key is to make coherence management an optimization choice rather than a burden for correctness. For applications that use task stealing, mutexes, reductions, mailboxes, or shared data regions, Cohesion can support those constructs efficiently with its HW_{cc} protocol while using the SW_{cc} protocol for the execution's data-parallel regions.

Here, we present three high-level programming examples using Cohesion. We break the examples into three categories, representing the predominate patterns that Cohesion's applications use. Static Cohesion divides the application's address space into HW_{cc} and SW_{cc} partitions at initialization. Throughout the application's runtime, the partitioning doesn't change. Dynamic Cohesion supports the migration of fine-grained memory regions between coherence domains throughout the application's runtime. System software is distinct from the static and dynamic patterns because it's not meant for applications' software, but is a pattern expressed in runtimes, synchronization libraries, and operating systems.

Static partitioning

For many workloads, we can partition the data into two groups: that which HW_{cc} would best serve and that which SW_{cc} would best serve. Static partitioning is similar to systems with multiple address spaces where one is coherent. However, because Cohesion covers the entire address space, it has the advantage of letting software choose the partition. A trivial case for static partitioning would put all code and stack data into SW_{cc} and the rest into HW_{cc}. Programmers could perform more elaborate partitioning on the basis of compiler analysis for read/write sets and liveness or using programmer annotations.

Figure A demonstrates an example of static partitioning. The figure shows two grid cells from a 2D stencil computation such as our heat benchmark. Each task has an allocated cell and uses two buffers to read from and write to, respectively, a number of time steps. Just one task accesses most of the data, so it doesn't need to be kept coherent. On the other hand, many tasks share the boundary regions. To enable such sharing trivially, we can statically partition each task's data set

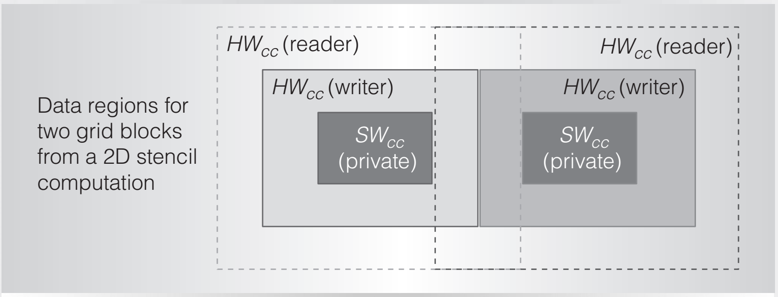

Figure A. The static Cohesion pattern demonstrated for a 2D stencil computation. The goal is to place the per-cell private regions in the software-coherence (SW_{cc}) domain and only use the hardware-coherence (HW_{cc}) domain for a small border region where communication is necessary.

We average samples taken every 1,000 cycles and classify the entries by whether they map to code (which is negligible), private stack data, or heap allocations and static global data. The HW_{cc} data in Figure 7c is a proxy for the benchmarks' on-die working set because all lines cached in an L2 have allocated directory entries and all uncached lines exit the directory when the L2 sharer count drops to zero. The data also illustrates the degree of read sharing because the ratio of directory entries to valid L2 cache lines is smaller when there are more sharers. The implication is that the system needs smaller directories.

Across all benchmarks, Cohesion reduces the average directory use by 2.1 times. For some benchmarks, simply keeping the stack incoherent achieves most of the benefit, but to have the private data kept SW_{cc} and the shared perimeter data kept HW_{cc}. This approach has two advantages: only the truly shared data must pay the cost of coherence, and the software must only change trivially to support Cohesion.

Dynamic partitioning

An advantage of Cohesion over systems that support only static partitioning is that Cohesion can transition between coherence domains at runtime. Cohesion achieves performance advantages when transitions are optimally placed by minimizing how much data the coherence protocol must track at any instant in time. Moreover, Cohesion applies to the entire address space and can be virtualized. If only static partitioning is allowed, the developer is left with a fixed partition that he or she must manage explicitly. Moreover, having only static partitions would require splitting and reorganizing data structures across HW_{cc} and SW_{cc} domains, which can cause large amounts of code to be refactored, which is undesirable. Cohesion avoids these programmability and performance bottlenecks by using fine-grained remapping between coherence domains that can occur at runtime.

Figure B provides an example of dynamic partitioning. The figure shows four cores performing a divide-and-conquer sort operation. During the divide phase, Cohesion distributes a partition of the working set to another processor using HW_{cc} while the processor performing the pivot keeps the data in SW_{cc}. Once the processors reach the sort's sequential phase, they can rely on SW_{cc} alone to avoid any coherence costs because there's no sharing. When the results are available, the processors can simply place them in the HW_{cc} domain, allowing consumers to source the data and avoiding cache flushing.

System software

Cohesion aids system software. Software-managed coherence doesn't allow for migration of tasks that might have modified data in the local caches without aggressive cache flushing. Cohesion provides the advantage of lazy, on-demand migration possible with HW_{cc} while still providing SW_{cc} benefits. Moreover, after a task or application completes, programmers can't assure that all data touched by the tasks is consistent with the last-level cache and memory without flushing the entire shared cache. Being able to force data into a hardware-tracked state, albeit at some cost, is Cohesion's advantage. Moreover, if the runtime needs to preempt a task and migrate it, the programmer can make the task's working set HW_{cc}, and the data will migrate with the task.

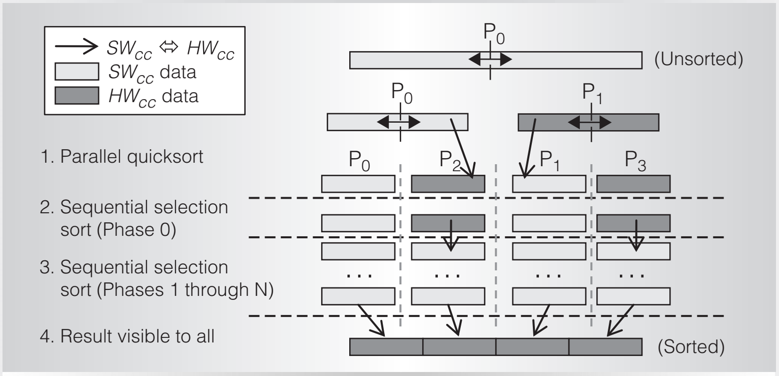

Figure B. The dynamic Cohesion pattern demonstrated using a two-phase parallel sort on four processor cores. The goal is to use HW_{cc} as needed during the divide phase and to avoid HW_{cc} during independent execution (shown as the serial sort). When the results are ready, HW_{cc} can make the produced data available to consumers.

on average, the stack alone only represents 15 percent of directory resources. Code makes up a trivial portion of the directory entries because our benchmarks have large data sets. These results show that most of the savings comes from using Cohesion to allocate globally shared data on the incoherent heap, thus avoiding coherence overhead for that data.

Application performance

Figure 8 shows our benchmarks' runtime normalized to Cohesion. Two benchmarks perform better and two do slightly worse with Cohesion relative to SW_{cc} and optimistic HW_{cc}, while the others show insignificant differences. Compared to realistic hardware assumptions, Cohesion delivers much better performance. These benefits

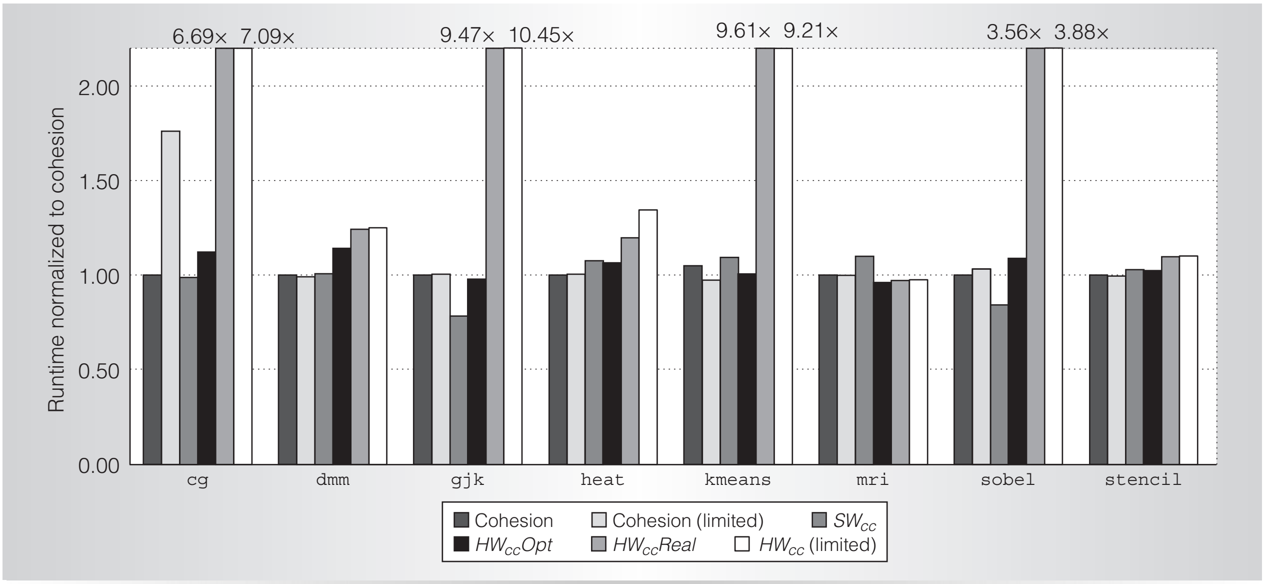

Figure 8. Relative performance for Cohesion with a full-map directory, Cohesion with a Dir_4B directory, SW_{cc}, optimistic HW_{cc}, and HW_{cc} with a full-map and Dir_4B 128-way sparse directory. We normalized all results to Cohesion with a full-map directory.

come from reduced message traffic and a reduction in the number of flush operations (many unnecessary) that the SW_{cc} configurations issue.

Microprocessor manufacturers will soon be shipping products that incorporate both general-purpose cores and an application-specific GPU. Composing the disparate memory models of these two classes or processors represents both new opportunities and new challenges. Cohesion is a proof-of-principle hybrid memory model that achieves consistency across coherence domains. The hope is that by providing a single shared address space for hybrid systems, Cohesion can increase programmability, while still leveraging the architectural benefits of existing coherent and non-coherent processors. MICRO

Acknowledgments

This work was supported in part by the Intel/Microsoft Universal Parallel Computing Research Center at the University of Illinois at Urbana-Champaign. We thank the Trusted Illiac Center at the Information Trust Institute for the use of their computing cluster.

References

1. J.H. Kelm et al., ''Cohesion: A Hybrid Memory Model for Accelerators,'' *Proc. 37th Ann. Int'l Symp. Computer Architecture,* ACM Press, 2010, pp. 429-440.
2. S.V. Adve et al., ''Comparison of Hardware and Software Cache Coherence Schemes,'' *Sigarch Computer Architecture News,* vol. 19, no. 3, 1991, pp. 298-308.
3. 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,* ACM Press, 1990, pp. 168-176.
4. 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.
5. J.H. Kelm et al., ''A Task-Centric Memory Model for Scalable Accelerator Architectures,'' *Proc. 18th Int'l Conf. Parallel Architectures and Compilation Techniques* (Pact 09), IEEE CS Press, 2009, pp. 77-87.
6. L.M. Censier and P. Feautrier, ''A New Solution to Coherence Problems in Multicache Systems,'' *IEEE Trans. Computers,* vol. 27, no. 12, 1978, pp. 1112-1118.

7. L.A. Barroso et al., ''Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing,'' *Proc. 27th Ann. Int'l Symp. Computer Architecture,* ACM Press, 2000, pp. 282-293.
8. M. Frigo, C.E. Leiserson, and K.H. Randall, ''The Implementation of the Cilk-5 Multithreaded Language,'' *ACM Sigplan Notices,* vol. 33, no. 5, 1998, pp. 212-223.
9. J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism,* O'Reilly, 2007.
10. M. Gschwind, ''Chip Multiprocessing and the Cell Broadband Engine,'' *Proc. 3rd Conf. Computing Frontiers,* ACM, 2006, pp. 1-8, doi:10.1145/1128022.1128023.
11. J.H. Kelm et al., ''Rigel: An Architecture and Scalable Programming Interface for a 1000-Core Accelerator,'' *Proc. 36th Ann. Int'l Symp. Computer Architecture,* ACM Press, 2009, pp. 140-151.

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

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

William Tuohy is a doctoral candidate in the Electrical and Computer Engineering Department at the University of Illinois at Urbana-Champaign. His research interests include parallel architectures, compiler and hardware interactions and codesign, and memory systems for heterogeneous architectures. He has a BS in electrical engineering and computer science from the University of California, Berkeley. He's a member of IEEE and the ACM.

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

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

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