COHESION*: A Hybrid Memory Model for Accelerators

John H. Kelm, Daniel R. Johnson, William Tuohy, Steven S. Lumetta, Sanjay J. Patel

University of Illinois at Urbana-Champaign
Urbana, IL 61801

ABSTRACT

Two broad classes of memory models are available today: models with hardware cache coherence, used in conventional chip multiprocessors, and models that rely upon software to manage coherence, found in compute accelerators. In some systems, both types of models are supported using disjoint address spaces and/or physical memories. In this paper we present COHESION, a hybrid memory model that enables fine-grained temporal reassignment of data between hardware-managed and software-managed coherence domains, allowing a system to support both. COHESION can be used to dynamically adapt to the sharing needs of both applications and runtimes. COHESION requires neither copy operations nor multiple address spaces.

COHESION offers the benefits of reduced message traffic and on-die directory overhead when software-managed coherence can be used and the advantages of hardware coherence for cases in which software-managed coherence is impractical. We demonstrate our protocol using a hierarchical, cached 1024-core processor with a single address space that supports both software-enforced coherence and a directory-based hardware coherence protocol. Relative to an optimistic, hardware-coherent baseline, a realizable COHESION design achieves competitive performance with a 2× reduction in message traffic, 2.1× reduction in directory utilization, and greater robustness to on-die directory capacity.

Categories and Subject Descriptors

C.1.4 [**Computer Systems Organization**]: Processor Architectures—*Parallel Architectures*

General Terms

Design, Performance

Keywords

Accelerator, Computer Architecture, Cache Coherence

*In linguistics, cohesion is the assemblage of ideas into a coherent discourse.

ISCA'10, June 19–23, 2010, Saint-Malo, France.

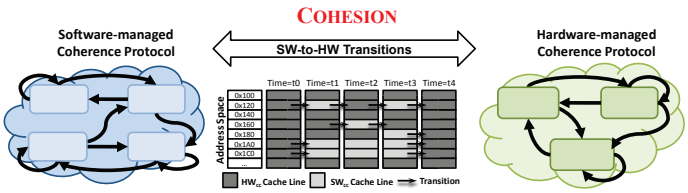


Figure 1: COHESION enables migrating data between coherence domains without copying.

1. INTRODUCTION

Chip multiprocessors (CMP) supporting four to 32 hardware threads are available commercially [11, 24, 28, 38], and compute accelerators, such as graphics processing units (GPUs), already support 1000's of concurrent hardware threads [30]. However, the memory systems of these two dominant multicore systems, general-purpose CMPs and domain-specific GPUs, indicate a clear divergence. 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 upon the existence of a hardware-coherent address space and the difficulty of scaling hardware cache coherence.

In this paper we present COHESION, a hybrid memory model that bridges the gap between weak, easily scalable models found in accelerators and strict, more difficult to scale models founds in CMPs. COHESION uses a hybrid hardware/software coherence scheme, as illustrated in Figure 1. When possible, COHESION performs coherence management in software at coarse granularity, or avoids it completely. For accesses that would otherwise be prohibitively expensive in a weaker model, COHESION incorporates hardware-supported coherence regions. COHESION achieves scalability by taking advantage of the lack of fine-grained sharing in many scalable parallel applications, 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.

The three key benefits of COHESION are (1) a system with a single address space that supports a variety of *coherence domains*, which are regions of memory for which coherence guarantees are provided collectively by the memory model,

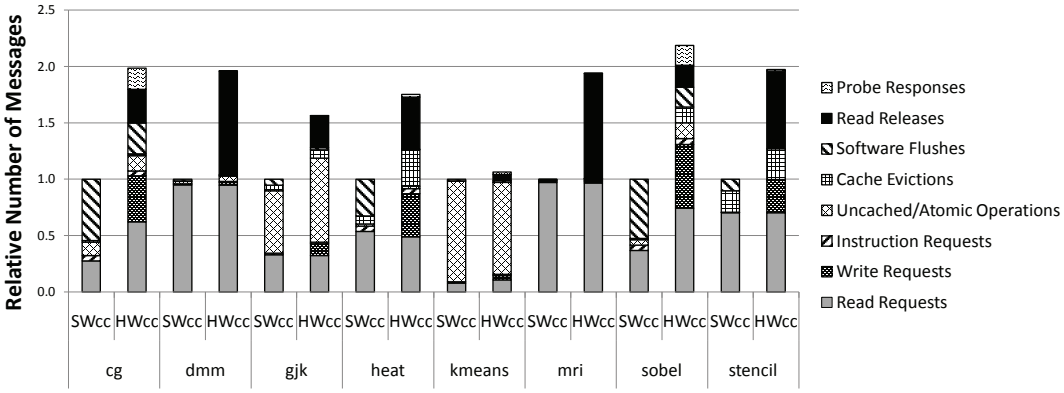


Figure 2: Number of messages sent by the local caches shared by eight cores (L2) to the global shared last-level cache (L3) for SW_{cc} and optimistic HW_{cc}. Results normalized to SW_{cc}.

(2) the ability to perform temporal reallocation between coherence domains without copies, and (3) fine granularity—interleaving data managed using a software protocol and a hardware protocol at the granularity of a cache line. The goal of COHESION is to enable scalability by reducing reliance upon hardware mechanisms without sacrificing the conventional shared memory programming model.

COHESION incorporates two components, one software and one hardware, and comprises a protocol for transitioning between the two. The software component leverages programmer knowledge of data usage, including immutable data and private data such as stacks. It is also useful for shared data with coarse-grained dependences separated by barriers, which is a common programming idiom in accelerator software [6, 19, 23, 31]. The software interface is a set of library routines that can be optimized for the underlying architecture, but do not rely on compiler support. The second part of COHESION is a hardware mechanism that allows software to indicate to hardware what regions of memory must be tracked by a hardware coherence protocol. The unifying mechanism is a protocol that allows regions of memory to transition between coherence domains at synchronization points in the code. The protocol allows for consistent behavior as coherence management of cache blocks transitions dynamically between hardware and software control.

2. MOTIVATION

We motivate COHESION by demonstrating that software-managed and hardware-managed coherence both have overheads that can be mitigated by a hybrid memory model. We also motivate COHESION 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 may be beneficial by reducing the need for data marshalling and the cost of data copies.

2.1 The Case for Software Cache Coherence

A software-managed coherence protocol embedded in the compiler [1, 25], runtime [3, 5], or programming model [18, 23] avoids the overheads of hardware coherence management. No memory is required for directories [2, 8] or duplicate tags [4, 24]. No design effort is expended implementing and verifying the coherence protocol. For the network, there is less traffic and relaxed design constraints. Furthermore, fine-grained software protocols can eliminate false sharing.

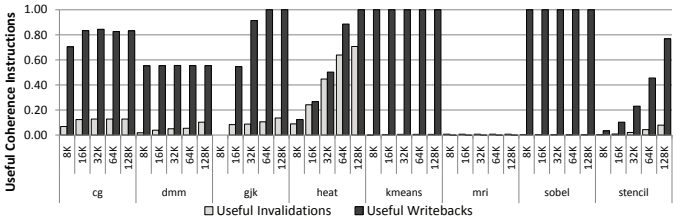


Figure 3: Fraction of software invalidations and writebacks issued that are performed on valid lines in the local cache (L2) with varied sizing. We eagerly write back dirty output data and lazily invalidate read-only input data. Operations performed to invalid lines are an inefficiency of SW_{cc}.

We simulated a 1024-core system running multi-threaded kernels (Section 4.1) that use a barrier-based task queue programming model. Figure 2 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 eliminates directory evictions and broadcasts. The software-managed protocol (SW_{cc}) uses explicit flush and invalidation instructions inserted by software to manage coherence. Figure 3 evaluates the efficiency of issued flush and invalidate instructions under SW_{cc} by counting the number of such instructions that operate on valid lines in the cache. The methodology is discussed further in Section 4.

Figure 2 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/invalidation operations. For SW_{cc}, software is responsible for tracking ownership. For a non-inclusive cache hierarchy [4, 11] per-word dirty/valid bits are maintained in cache. Writes can be issued as write-allocates under SW_{cc} without waiting on a directory response. Also, under HW_{cc} silent evictions are

not supported and thus read releases are issued to the directory when a clean line is evicted from a local cache (L2). For SW_{cc}, there is no need to send any message, and the invalidation occurs locally. Even if a protocol without read releases were used, HW_{cc} would still show significant message overhead for invalidation cache probes; the implications of a protocol without read releases are discussed in Section 3.2.

SW_{cc} has the ability to mass invalidate shared read data, signaling many invalidations with only a few messages. To coordinate this action, a global synchronization event such as a barrier is used. 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 the critical path for coherence actions, increases contention for the network, and requires greater capacity from the network to handle the invalidation traffic. SW_{cc} can eliminate false sharing since multiple write sharers that access disjoint sets of words on a line will not generate coherence probes that would otherwise cause the line to ping-pong between sharers in HW_{cc}.

2.2 The Case for Hardware Cache Coherence

Hardware cache coherence provides a number of 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 makes it easier to reason about sharing in some applications. HW_{cc} enables speculative prefetching and data migration. Shared memory applications can be ported to a 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 allowing a requester to locate the latest copy of data on-demand. The implication is that SW_{cc} protocols may be more conservative than necessary, pushing all data that *may* be read by another core to a globally-visible point, e.g., memory or a globally shared last-level cache. Furthermore, the additional traffic required for read release messages under HW_{cc} makes up a significant portion of message traffic, these messages are not on the critical path for a waiting access, whereas an invalidation sent by the directory is.

SW_{cc} reduces instruction stream efficiency since it introduces explicit cache many flush instructions which show up as additional software flush messages in Figure 2. 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 coherence state. Furthermore, flush instructions are wasted, as the lines they target may have already been evicted from the cache by the time the SW_{cc} actions occur. We quantify this effect in Figure 3, which depicts the number of SW_{cc} actions that are performed on lines valid in the L2. Indeed, many of the coherence instructions issued dynamically are superfluous, operating on lines not present in the cache.

2.3 A Hybrid Memory Model

Supporting multiple coherence implementations enables software to dynamically select the appropriate mechanism for blocks of memory. Supporting incoherent regions of memory allows more scalable hardware by reducing the number of

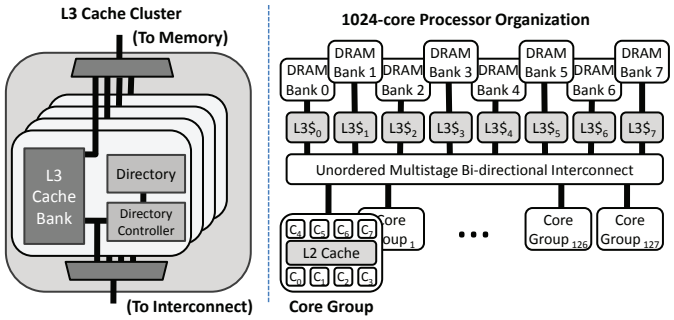


Figure 4: Baseline processor architecture.

shared lines, resulting in fewer coherence messages and less directory overhead for tracking sharer state. Furthermore, having coherence as an option enables tradeoffs to be made regarding software design complexity and performance. A hybrid memory model provides the runtime with a mechanism for managing coherence needs across applications, even if modified applications do not need data to transition between SW_{cc} and HW_{cc} frequently,

From the perspective of system software, HW_{cc} has many benefits. HW_{cc} allows for migratory data patterns not easily supported under SW_{cc}. 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, task-based programming models [12, 36] are aided by coherence. HW_{cc} allows children tasks to be scheduled on their parent, incurring no coherence overhead, or stolen by another core, which allows data to be pulled using HW_{cc}.

Systems-on-a-chip, which incorporate accelerators and general-purpose cores, are available commercially [42] and are a topic of current research [44]. The assortment of cores makes supporting multiple memory models on the same chip attractive. A hybrid approach allows for cores without HW_{cc} support, such as accelerator cores, to cooperate with cores that do have HW_{cc} support and interface with coherent general-purpose cores. While a single address space is not a requirement for heterogeneous systems, as demonstrated by the Cell processor [14] and GPUs [30], it may aid in portability and programmability by extending the current shared memory model to future heterogeneous systems. A hybrid approach allows for HW_{cc} to be leveraged for easier application porting from conventional shared memory machines and easier debugging for new applications. SW_{cc} can then be used to reduce the stress on the hardware coherence mechanisms to improve performance.

2.4 Summary

Hardware-managed and software-managed cache coherence offer both advantages and disadvantages for applications and system software. We list many of the tradeoffs in Table 1. A hybrid memory model such as COHESION leverages the benefits of each while mitigating the negative effects of the respective models. The key benefits from SW_{cc} are reduced network and directory costs. The key benefits from HW_{cc} derive from its ability to share data without explicit software actions, which, as we demonstrate, can be costly in terms of message overhead and instruction stream efficiency. A

	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	Potential dependences handled by hardware instead of extra instructions and coherence traffic	Optimized for HW_{cc}: when HW_{cc} desired, coherence data stored efficiently
SW_{cc}	Used in accelerators; provides programmer/compiler control over sharing	Eliminates probes/broadcasts for independent data, e.g., stack, private, 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-grain/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

Table 1: Tradeoffs for HW_{cc}, SW_{cc}, and COHESION.

hybrid approach can enable scalable hardware-managed coherence by supporting HW_{cc} for the regions of memory that require it, using SW_{cc} for data that does not. In comparison to a software-only approach, a hybrid memory model makes coherence management an optimization opportunity and not a correctness burden.

3. DESIGN

COHESION provides hardware support and a protocol to allow data to migrate between coherence domains at runtime, with fine granularity, and without the need for copy operations. The default behavior for COHESION is to keep all of 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 is applied to data that are not shared, or can have coherence handled at a coarse granularity by software, using the SW_{cc} domain. The rest of this section describes the protocols and hardware support that enable on-the-fly coherence domain changes.

3.1 Baseline Architecture

Figure 4 shows a diagram of the 1024-core accelerator we use as a baseline. It is a variant of the Rigel architecture [22]. The processor has a single address space and three levels of cache. 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 *cluster* and share a unified L2 cache. A single multi-banked shared last-level L3 cache serves as the point of coherence for the chip.

The interconnect between the cores and the L2 is a pipelined two-lane split-phase bus. The interconnect between the clusters and the L3 cache banks has two levels. The first level is a tree interconnect that combines the traffic of sixteen clusters. The root of each tree is connected to a crossbar that connects to L3 cache banks. Each four banks of L3 have an independent GDDR memory channel. Table 3 lists timing and sizing parameters for the design.

3.2 Hardware Coherence Protocol

The hardware protocol is represented on the right side of Figure 6. Data in the HW_{cc} domain uses an MSI protocol. An exclusive state is not used due to the high cost of exclusive to shared downgrades for read-shared data. Owned state is omitted since we use the L3 to communicate data and the directory to serialize accesses, removing much of the benefit of sourcing shared data from another L2.

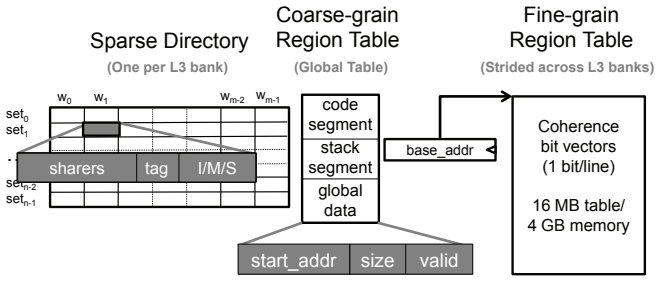


Figure 5: COHESION architecture.

For overall runtime experiments, a limited directory scheme [2] (Dir_4B) is used due to its lower storage overhead. Other experiments use a full-map scheme [8], which we use to provide a lower bound on the performance impact and message overhead of pure hardware coherence. We employ sparse directories [15, 34] where the directory only holds directory entries for lines present in at least one L2. Note that the directory storage overhead is worse for CMPs than multi-socket systems since the overhead grows linearly in the number of sharers, i.e., cores, and not sockets. Therefore, memory bandwidth does not scale with directory size as in multi-socket multiprocessors with directories. Duplicate tags [4] were not chosen due to their high required associativity (2048 ways) and difficulty supporting a multi-banked last-level cache which may require replicating the duplicate tags across L3 banks.

The baseline architecture is non-inclusive between L2 and L3 caches. The directory is inclusive of the L2s and thus may contain entries for lines not in the L3 cache. L2 evictions notify the directory and, if the sharer count drops to zero, the entry is deallocated from the directory. Entries evicted from the directory have all sharers invalidated.

One bank of the directory is attached to each L3 cache bank. All directory requests are serialized through a home directory bank, thus avoiding many of the potential races in three-party directory protocols [27]. Associating each L3 bank with a slice of the directory allows the two mechanisms to be collocated, reducing the complexity of the protocol implementation compared to a design where a directory access may initiate a request to a third party structure, such as another L2 cache as in Origin [27] or an L3 bank that is across the network.

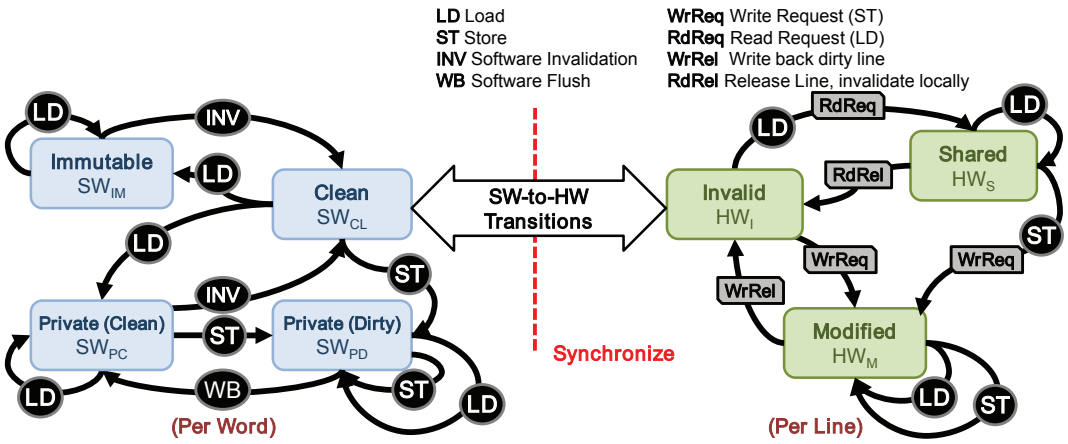


Figure 6: COHESION as a protocol bridge. Software-managed coherence is shown on the left, the hardware coherence protocol on the right. COHESION allows for blocks of memory to transition between the two.

3.3 Software Coherence Protocol

Our software coherence protocol is the Task Centric Memory Model [23] modified to support hybrid coherence, as shown on the left side of Figure 6. The protocol leverages the bulk-synchronous [41] (BSP) compute pattern. BSP comprises phases of mostly data-parallel execution followed by communication, with barriers separating each phase. The software protocol makes use of the fact that most data is not read-write shared across tasks between two barriers and most inter-task communication occurs across barriers. Other software protocols for managing coherence could be used, but we restrict ourselves to BSP for simplicity of illustration and its applicability to existing scalable programming models.

The software protocol provides a set of state transitions, which are initiated explicitly by software or implicitly by hardware, that allow for a programmer or compiler to reason about coherence for a block of data. The motivation for designing such a protocol is that in a system that uses cached memory for communication, but without hardware coherence, there is the potential for hardware to implicitly move data into a globally visible location. These implicit actions are uncontrollable by software and thus the SW_{cc} protocol, and by extension COHESION, must take them into account.

3.4 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 shown in Figure 5, the system is composed of a directory for tracking currently shared HW_{cc} lines, a coarse-grain region table for tracking common large regions of memory that are SW_{cc}, and a fine-grain region table that is used for tracking the rest of memory that may transition between HW_{cc} and SW_{cc}. One bit of state per line, the *incoherent bit*, is added to the L2 cache to track which cached lines are not HW_{cc}. A compressed hardware structure, such as the structure used in [43], was considered to move all tracking information on-die. However, we find a bitmap cached at the L3 to be a sufficient approach. If additional L3 latency for table accesses becomes a concern, the dense structure of the table is amenable to on-die caching.

The directory is queried when a request arrives at the L3. If the line is a directory hit, the line is in HW_{cc} and the response is handled by the directory. If the line is accessible to the requester, the L3 is accessed in the next cycle and the response is returned to the requesting L2. A directory hit with an L3 miss results in the directory access blocking. A response is generated when the fill from memory occurs. A directory miss results in the region tables being examined.

The coarse-grain region table is a small on-die structure that contains a map of address ranges that are in the SW_{cc} domain and is accessed in parallel with the directory. The three regions used most frequently 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 cache controller responds with the data. The message includes a bit signalling 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 no message is sent from the L2 to the L3.

The fine-grained region table is queried for all other accesses. The table may be cached in the L3 since 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, 16 MB total. To reduce the footprint of the table, a subset of memory could be designated as the COHESION-enabled region. A fine-grained region table lookup must access the L3. A minimum of one cycle of delay is incurred by fine-grained lookups and more under contention at the L3 or if an L3 cache miss for the table occurs. 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.

API Call	Description
`void * malloc(size_t sz)`	Allocate memory on coherent heap. Data is always in HW_{cc} domain. Standard libc implementation.
`void free(void * hwccptr)`	Deallocate object pointed to by `hwccptr`. Standard libc implementation.
`void * coh_malloc(size_t sz)`	Allocate memory on the incoherent heap. Data is allowed to transition coherence domains. Initial state is SW_{cc} and the data is not present in any private cache.
`void coh_free(void * swccptr)`	Deallocate object pointed to by `ptr`.
`void coh_SWcc_region(void * ptr, size_t sz)`	Make region `ptr` part of the SW_{cc} domain. Data may be HW_{cc} or SW_{cc} data.
`void coh_HWcc_region(void * ptr, size_t sz)`	Make region `ptr` part of the HW_{cc} domain. Data may be HW_{cc} or SW_{cc} data.

Table 2: Programmer-visible software API for COHESION.

The region tables are set up by the runtime at initialization. The bootstrap core allocates a 16MB region for the fine-grained region table, zeroes it, and sets a machine specific register to the base address of the table in physical memory. The process is akin to setting up a hardware-walked page table. To toggle the coherence domain of a line, the runtime uses atomic instructions, `atom.or` and `atom.and`, that bypass local caches and perform bitwise operations at the L3 to set or clear bits in the table, respectively. Atomic read-modify-write operations are necessary to avoid races between concurrent updates to the table.

Calculating an address to modify an entry in the table requires special consideration. The table is distributed across the L3 banks in our design. To remove the need for one L3 bank to query another L3 bank on a table lookup, we map the slice of the table covering one L3 bank into the same L3 bank it maps into. Since the address space is strided across L3 banks, the target address that we want to update in the table must be hashed before being added to the table base address[1]. Since the hash function is dependent upon the number of L3 banks, we choose to add an instruction to perform the hashing. The `hybrid.tbloff` instruction takes the target address and produces a word offset into the table that can be added to the base address before accessing the table from hardware. Adding such an instruction makes COHESION microarchitecture-agnostic since the number and size of L3 banks and the stride pattern can be changed without modifying software.

3.5 Software Interface to COHESION

In this section we discuss the application programming interface (API) to COHESION. The API calls are listed in Table 2. For this work we make two simplifying assumptions. First, we assume there is a single application running on the system. Second, we assume a single 32-bit address space where physical addresses match virtual addresses. The architecture we propose could be virtualized to support multiple applications and address spaces concurrently by using per-process region tables.

The COHESION region tables are initialized by the runtime when the application is loaded. The coarse-grain 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 are found in the ELF header for the application binary. Our architecture does not support self-modifying code so HW_{cc} is not required for cached instructions. Variable-sized stacks are possible by treating the stack region as a SW_{cc} heap, but fixed-sized stacks were found to be sufficient for our current implementation. There are two heaps in our implementation: a conventional C-style heap that is kept coherent and another that is not kept HW_{cc} by default. The *incoherent heap* is used for data that may transition coherence domains during execution. Note that the minimum sized allocation on the incoherent heap is 64 bytes, or two cache lines, so that the metadata for the allocation can be kept coherent. Current libc implementations support 16-32 byte minimum allocations, so we believe 64 bytes to be reasonable.

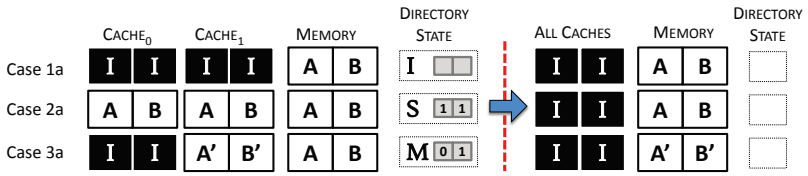
[1]DRAM row stride is used. $addr_{[10..0]}$ map to the same memory controller and $addr_{[13..11]}$ are used to stride across controllers. The hashing function for an eight controller configuration uses $addr_{[9..5]}$ to index into the word, and the table word offset address is: $addr_{[31..24]} \circ addr_{[13..11]} \circ addr_{[23..14]} \circ addr_{[10]} << 2$.

3.6 Coherence Domain Transitions

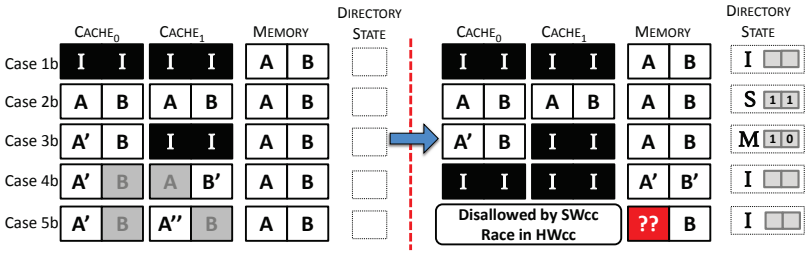
A transition between SW_{cc} and HW_{cc} is initiated by word-aligned, uncached read-modify-write operations performed by the runtime to the fine-grained region table. The issuing core blocks until the transition is completed by the directory for the purposes of memory ordering. If a request for multiple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

The directory controller is responsible for orchestrating the $HW_{cc} \Leftrightarrow SW_{cc}$ transitions. The directory snoops the address range for the fine-grained region table and on an access that changes coherence domain of a line, the directory performs the actions described below to transition between SW_{cc} and HW_{cc}. The request is completed by sending an acknowledgement to the issuing core. Handling the transitions at the directory allows for requests for a line to be serialized across the system. Coherence domain transitions for a single line thus occur in a total order across the system with all other coherent and non-coherent accesses at the L3 being partially ordered by the transitions.

$HW_{cc} \Rightarrow SW_{cc}$ Transitions To move 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 7(a) shows the potential states a line can be in when software initiates a $HW_{cc} \Rightarrow SW_{cc}$ transition. Each of the states corresponds to a possible state allowed by the MSI directory protocol. After a transition is complete, the line is not



(a) $HW_{cc} \Rightarrow SW_{cc}$ conversion



(b) $SW_{cc} \Rightarrow HW_{cc}$ conversion

Figure 7: Potential system states on a $SW_{cc} \Leftrightarrow HW_{cc}$ conversion. Memory and L3 are interchangeable. Each block shown has two words. A and B are the initial values. On the left side of the figure, A', A", and B' denote distinct modified values in the local cache. Grayed blocks are potentially valid, but are not accessed by software. The right side of the figure shows the system state after the state transition occurs.

present in any L2 and the current value is present in the L3 cache or in memory.

For Case 1a of Figure 7(a), the directory controller queries the directory and finds the entry not present indicating that there are no sharers. Therefore, no action must be taken other than to 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 sharers of the line. When all acknowledgements are received, the directory entry is removed, the bit is set in the region table, and the response is returned to the core requesting the transition. When a line is in the modified state, shown in Case 3a, a newer version of the data exists in some L2. The directory sends a writeback request to the owner. When the response arrives, the L3 is updated, the table is modified, and a response is sent to the requester.

$SW_{cc} \Rightarrow HW_{cc}$ **Transitions** The left half of Figure 7(b) shows the potential states that a line may be in for two L2 caches when a line is under SW_{cc}. Since the directory has no knowledge of the line state, a transition to HW_{cc} initiates a broadcast clean request sent from the directory. When the line is found in an L2 in the clean state, the incoherent bit is cleared, i.e., the line is now susceptible to cache probes, but the line is not evicted from the L2. Clean lines send an acknowledgement message to the directory which adds the L2 to the list of sharers. If the line is not found in the L2, a negative acknowledgement is sent to the directory. Figure 7(b), Cases 1b and 2b demonstrate transitions for lines that are not modified.

If a dirty line is found, shown in Cases 3b and 4b, the L2 sends the directory a notification. If there are any read sharers, 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 writeback occurs, saving bandwidth. If multiple writers are found, writeback requests are sent to all L2s holding modified lines and invalidations are sent to all L2s holding clean lines. Our architecture maintains per-word dirty bits with the cache lines allowing the L3 to merge the result of multiple writers if the write sets are disjoint. When either operation is complete, the line is not 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 may not be safe. It is possible for buggy software to modify the same line in two L2 caches concurrently when under the control of SW_{cc}, Figure 7(b), Case 5b. This state represents a hardware race. To safely clean the state of a word, the runtime can always turn on coherence and then zero the value. It will cause the dirty values in the separate caches to be thrown away and the zeroed value to persist. For debugging, it may be useful to have the directory signal an exception with its return message to the requesting core.

4. EVALUATION

Two benefits of using Cohesion are a reduction in message traffic and a reduction in the on-die directory storage overhead compared to HW_{cc}. 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.

4.1 Methodology

We simulate the architecture described in Section 3.1. We use an execution-driven simulation of the design and run each benchmark for at least one billion instructions. We model cores, caches, interconnects, and memory controllers. The cycle-accurate DRAM model uses GDDR5 timings.

We evaluate four design points: SW_{cc}, optimistic HW_{cc}, HW_{cc} with realistic hardware assumptions, and Cohesion with the same realistic hardware assumptions. The hardware configuration for SW_{cc} is equivalent to our baseline described in Section 3.1, but with no directory and all sharing

Parameter	Value	Unit	Parameter	Value	Unit	Parameter	Value	Unit
Cores	1024	–	Line Size	32	bytes	Directory Size (realistic)	16K	$\frac{entries}{L3\ bank}$
Memory BW	192	GB/s	Core Freq.	1.5	GHz	Directory Assoc. (realistic)	128	ways
DRAM Channels	8	–	DRAM Type	GDDR5	–	Directory Size (optimistic)	∞	$\frac{entries}{L3\ bank}$
L1I Size	2	KB	L1I Assoc.	2	way	Directory Assoc. (optimistic)	Full	–
L1D Size	1	KB	L1D Assoc.	2	way	L2 Ports	2	R/W
L2 Size	64	KB	L2 Assoc.	16	way	L3 Ports	1	R/W
L2 Size (Total)	8	MB	L2 Latency	4	clks	L3 Latency	16+	clks
L3 Size	4	MB	L3 Assoc.	8	way	L3 Banks	32	–

Table 3: Timing parameters for the baseline architecture.

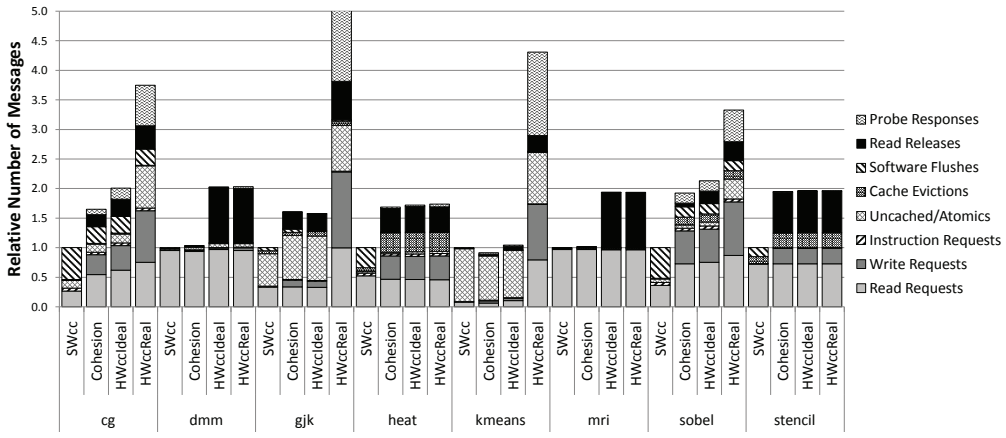


Figure 8: L2 output message counts for SW_{cc}, COHESION, HW_{cc} with a full on-die full-map directory, and HW_{cc} with a 128-way sparse directory on-die normalized to SW_{cc}.

handled by software. For SW_{cc} writes occur at the L2 with no delay and evictions performed to clean data happen without creating any network traffic. The optimistic HW_{cc} case removes all directory conflicts by making the on-die directory infinitely sized and fully-associative. The behavior is equivalent to a full-map directory in memory [8], 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.

The applications that we evaluate are optimized kernels extracted from scientific and visual computing applications. The SW_{cc} variants have explicit invalidate and writeback instructions in the code at task boundaries. The COHESION variants have such instructions when SW_{cc} is used and none for data in the HW_{cc} domain. The COHESION API, as shown in Table 2, is used to allocate the SW_{cc} and non-coherent data on the incoherent heap. The HW_{cc} versions eliminate programmed coherence actions in the benchmark code. The benchmarks are written using a task-based, barrier-synchronized work queue model. The kernels include conjugate gradient linear solver (`cg`), dense matrix multiply (`dmm`), collision detection (`gjk`), 2D stencil (`heat`), k-means clustering (`kmeans`), medical image reconstruction (`mri`), edge detection (`sobel`), and 3D stencil (`stencil`).

4.2 Message Reduction

Figure 8 shows the number of messages sent by the L2s to the directory, normalized to SW_{cc}. There is a reduction in messages 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 is due to optimizations that reduce the number of uncached operations issued by the benchmark by relying upon HW_{cc} under COHESION. For some benchmarks, the number of messages are nearly identical across COHESION and optimistic HW_{cc} configurations, such as `heat` and `stencil`. We see potential to remove many of these messages by applying further, albeit more complicated, optimization strategies using COHESION. We leave more elaborate coherence domain remapping strategies to future work.

4.3 Directory Entry Savings

Figures 9a and 9b show the normalized runtime for different directory sizes under HW_{cc} and COHESION, respectively, compared to an infinite-sized directory. We make directories fully associative to isolate the influence of capacity. Figure 9a demonstrates the rapid drop off in performance for shrinking directory sizes. In Figure 9b we show that COHESION reduces the performance sensitivity with respect to directory sizing across all benchmarks. Figure 9c shows the mean and maximum number of directory entries used by COHESION and the optimistic HW_{cc} baseline. We average samples taken every 1000 cycles. We classify the entries as to whether they map to code, which is negligible, private stack data, or heap allocations and static global data. The HW_{cc} data in Figure 9c is a proxy for the on-die working set of the benchmarks since all lines cached in an L2 have allocated directory entries and all uncached lines are removed from the directory when the L2 sharer count drops to zero. The data also illustrates the degree of read sharing since the ratio of directory entries to valid L2 cache lines is smaller

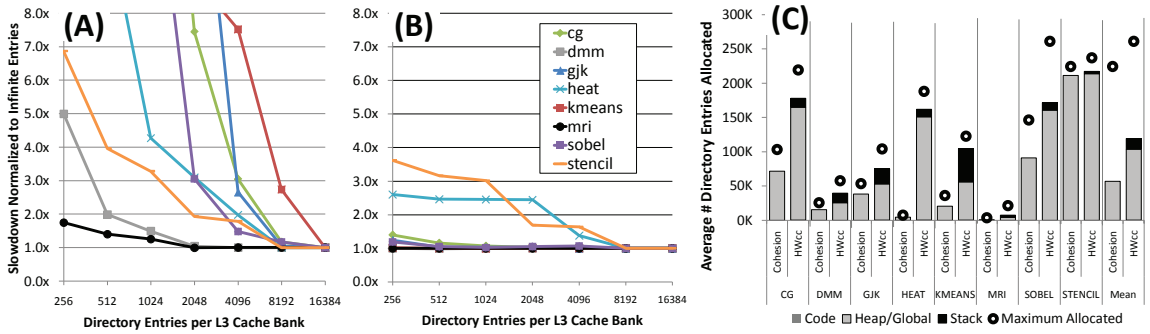


Figure 9: Slowdown for different directory sizes is shown for (A) HW_{cc} and (B) COHESION. Note that performance falls off precipitously in HW_{cc}. (C) shows the time-average (sampled every 1000 cycles) and maximum number of directory entries allocated for for both COHESION and HW_{cc} when directories have unbounded size and associativity.

when there are more sharers. The implication of more read sharing is that smaller directories are needed.

Across all benchmarks, COHESION provides a reduction in average directory utilization of 2.1×. For some benchmarks, simply keeping the stack incoherent achieves most of the benefit, but on average, the stack alone only represents 15% of the directory resources. Code makes up a trivial portion of the directory entries since 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.

4.4 Directory Area Estimates

To quantify the area savings COHESION could provide, we evaluate the on-die area costs for HW_{cc} by comparing the number of bits necessary for each scheme. The 128 L2 caches have a capacity of 2048 lines each, resulting in 256K 32-byte lines on-die, and 8 MB total. A full-map directory requires 128 bits for sharer data and 2 bits of state for each line. For a sparse scheme, 16 additional tag bits are also be required. A limited scheme with four pointers (Dir_4B) requires 28 bits per entry for sharer state and 2 bits for coherence state. The sparse directory structure is set-associative and banked to mitigate the effects port conflicts and load imbalance across L3 banks. The overhead for a full-map directory is 9.28 MB (113% of L2) and a limited scheme is 2.88 MB (35.1% of L2).

Duplicate tags [4] require 21 bits for each L2 tag. Since the directories are distributed across L3 banks, it may be necessary to replicate duplicate tag entries across banks, which leads to a 1× to 8× overhead. Duplicate tags require require 736KB * $N_{replicas}$. While a single set of duplicate tags would result in only 736 KB (8.98% of L2) overhead, the structure would still be 2048-way associative and need to service up to 32 requests per cycle on behalf of the L3 cache banks.

Clearly, none of these optimistic options is attractive for a practical design. However, COHESION has the potential to reduce hardware overhead by building smaller directories, or fewer replicas and ports for duplicate tags, compared to a pure HW_{cc} system, making known techniques more scalable and attractive. Our results indicate that a greater than 2× reduction in directory utilization is possible, which could yield 5% to 55% reduction in L2 overhead for directory storage on-die.

4.5 Application Performance

Figure 10 shows the run time of our benchmarks normalized to COHESION. As the results demonstrate, two benchmarks perform better and two 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 many times better performance. These benefits come from reduced message traffic (Section 4.2) and a reduction in the number of flush operations issued by the SW_{cc} configurations, many of which may be unnecessary, as shown in Figure 3.

The negative performance difference between COHESION and optimistic HW_{cc} is less than 5% for `gjk` and `mri` and could be improved to parity with optimistic HW_{cc} using more complex coherence domain transitions not evaluated in this work. Moreover, neither benchmark is limited by coherence costs, but rather by task scheduling overhead due to task granularity in the case of `gjk` and execution efficiency for `mri` due to its high arithmetic intensity. The `kmeans` benchmark has a large number of atomic operations that can conflict with SW_{cc} coherence actions and lead to decreased performance due to queuing effects in the network. In some cases, HW_{cc} has the effect of dispersing the coherence actions in time, thus reducing the effect of queuing delays.

4.6 Summary

COHESION reduces the number of messages sent compared to a purely hardware coherent configuration, thus reducing demand on the network. COHESION is able to reduce the pressure on the directory, thus reducing the resources needed by the directory to achieve similar performance to a purely hardware coherent design. While we find an increase in the total number of messages injected when converting regions from the SW_{cc} domain to the HW_{cc} domain, we show improved performance due to the reduction in time spent issuing coherence management instructions under SW_{cc} and the timeliness of HW_{cc} messages. Overall, COHESION provides increased performance, reduced network demands, and less directory pressure for most workloads compared to SW_{cc} or HW_{cc} alone.

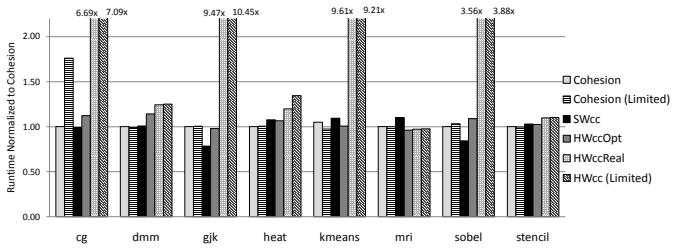


Figure 10: Relative performance for COHESION with a full-map directory, COHESION with a Dir_4B directory, SW_{cc}, optimistic HW_{cc}, and $HW_{cc}$with a full-map and Dir_4B 128-way sparse directory. All results normalized to COHESION with a full-map directory.

There is an interplay between message count, directory utilization, and execution time. Reducing the dependence on the directory by minimizing the number of lines tracked by HW_{cc} allows for fewer on-die directory resources to be provisioned and can avoid pathological cases due to directory set aliasing and capacity issues. However, doing so may lead to more messages being injected, thus reducing performance unless network capacity is increased. The programming effort is increased beyond developing an application using HW_{cc} alone as the developer must partition data accesses into SW_{cc} and HW_{cc} domains. Therefore, the tradeoff is between the level of achieved performance and the amount of programming effort required.

5. RELATED WORK

In this section, we put COHESION into context with previous work in scalable hardware-managed and software-managed coherence.

Hardware Schemes.

Snoopy coherence protocols [13] rely upon ordered networks and broadcasts to maintain consistency across caches, but are difficult to scale. Directory schemes [8] provide a scalable alternative, but full implementations suffer from high memory overhead. Limited directories [2] and sparse directories [15] can aid in reducing memory overhead with the potential cost of extra messages.

Coarse-grain coherence tracking [7] and RegionScout [33] both propose mechanisms to reduce coherence traffic in broadcast-based systems by managing coherence at a coarser granularity than a line. While these techniques can reduce storage costs, both mechanisms impose restrictions on alignment and sizing of coherence regions and may lead to increased message traffic; both are situations we wish to avoid with COHESION. Token coherence [32] attempts to increase scalability of coherence by associating a fixed number of tokens with each line and requiring a core to posses at least one token to read and all tokens to write. Token coherence requires additional messages for token requests avoided in COHESION by using software-coherence for known-private regions. Moreover, Raghavan [35] adds a directory to reduce broadcast occurrences, but still requires all lines to be tracked by the on-die directory.

Modern accelerator hardware, such as Cell [14] and NVIDIA Tesla [30], provide a variety of access modes to data located in different physical address spaces, but require explicit software management of data movement. Leverich et al. [29] demonstrate the benefit of software management of data movement on hardware cached platforms, which COHESION can facilitate with the added precision of variable coherence regions.

Software/Hybrid Schemes.

Distributed shared memory (DSM) provides the illusion of a single coherent address space across processors at a coarse grain using virtual memory and a runtime system in the case of TreadMarks [3] and at a fine grain using compiler support such as in Shasta [40]. While these approaches could trivially support incoherence due to their distributed memory architecture, they synthesized coherence when needed fully in software. Cooperative Shared Memory [18] uses software hints to reduce the complexity and cost of hardware supported coherence. Software-assisted hardware schemes, such as LimitLESS [9], trap to software when the number of sharers supported by hardware is exceeded. CSM and LimitLESS suffer from high round-trip latency between directory and cores in a hierarchically cached system and require all data to be tracked by the coherence protocol, 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 [26] and Typhoon [37] utilized programmable protocol controllers that support customized protocols. User-Level Shared Memory in Typhoon made fine-grained access control a key component of customizing protocols. COHESION provides a mechanism to allow such customization without a separate protocol controller. Munin [5] used parallel program access patterns to provide different consistency guarantees to different classes of data. Multiphase Shared Arrays [10] provide a means for the programmer to specify access modes for array data and to change the mode for different program phases. COHESION on an integrated shared memory multiprocessor captures these features with modest hardware and an intuitive programming model that does not require a message passing component.

The ability to change the strictness of regions of memory is available in a limited form on x86 processors with write combining [21] and PowerPC allows for pages to be set incoherent [20]. Both mechanisms work at page granularity and require operating system support. A hybrid x86 system with a consistent programming model for GPU and CPU cores using a subset of the address space has been proposed [39]. Unlike COHESION, that work does not investigate dynamic transitions between different coherence domains. Furthermore, while [39] focuses solely on a host-accelerator model with one CPU core, COHESION is demonstrated using 1024 cooperating cores.

WildFire [16] used operating system support to transition between line-level ccNUMA shared memory and a form of COMA known as Coherent Memory Replication (CMR). CMR and ccNUMA pages trade off coherence space overhead for coherence message overhead since all lines in the system were tracked by hardware and CMR pages require replication of coherence state. Copy operations are required by

replication under CMR, which are eliminated by COHESION. Unlike WildFire, COHESION provides symmetric access to the last-level cache, analogous to memory in DSM systems, and does not present a tradeoff between state and message overheads; it reduces both when using SW_{cc}.

Reactive NUCA [17] uses operating system remapping of page-sized regions on a distributed NUCA multicore. The analysis in [17] shows that different regions of memory possess different coherence needs, an opportunity for hybrid coherence, and illuminates many of the tradeoffs in scaling distributed versus shared cache architectures. In contrast to COHESION, the work evaluates mostly server and multiprogrammed workloads scaling to eight or sixteen cores, while we target the class of visual computing applications that tend to have higher degrees of read-sharing and more structured sharing patterns [6, 19, 23, 31] and are shown to scale to 100+ cores.

6. CONCLUSION

In this paper we present COHESION, a memory model that allows applications to combine the benefits of software and hardware coherence schemes into one unified model. We show the opportunity for message reduction and area reduction provided by a hybrid scheme relative to hardware-only cache coherence. We present a design for the hardware required to support a hybrid memory model and the protocol that the software and directory must follow to safely transition between coherence domains.

Our evaluation demonstrates that hardware coherence results in an increased number of network messages, even for data-parallel kernels. Even with optimistic hardware assumptions, hardware coherence provides poor performance for some benchmarks and can lead to thrashing behavior under more realistic hardware assumptions, resulting in significant slowdown. We find that, using COHESION, we can reduce the number of messages and thus increase performance while relaxing the design requirements for the network.

Lastly, we find that having both hardware-managed coherence and software-managed coherence as options enables developers to optimize portions of an application in a targeted way. Developers focus on data structure organization and matching coherence needs to the data access patterns across phases of execution instead of reasoning about code paths that interact and reorganizing data layouts for each phase of computation. Furthermore, we see potential for using a hybrid approach for integrating host processors and accelerators under one memory model. Ultimately, COHESION makes explicit coherence management for accelerators an optimization opportunity and not a correctness burden.

7. ACKNOWLEDGEMENTS

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.

8. REFERENCES

[1] S. V. Adve, V. S. Adve, M. D. Hill, and M. K. Vernon. Comparison of hardware and software cache coherence schemes. *SIGARCH Comput. Archit. News*, 19(3):298–308, 1991.

[2] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. In *ISCA '88: Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 280–298, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[3] 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*, 29(2), 1996.

[4] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: a scalable architecture based on single-chip multiprocessing. In *ISCA '00: Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 282–293, New York, NY, USA, 2000. ACM.

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

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

[7] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Improving multiprocessor performance with coarse-grain coherence tracking. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 246–257, Washington, DC, USA, 2005. IEEE Computer Society.

[8] L. M. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Trans. Comput.*, 27(12):1112–1118, 1978.

[9] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS directories: A scalable cache coherence scheme. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 224–234, New York, NY, USA, 1991. ACM.

[10] J. DeSouza and L. V. Kalé. MSA: Multiphase specifically shared arrays. In *Proceedings of the 17th International Workshop on Languages and Compilers for Parallel Computing*, West Lafayette, Indiana, USA, September 2004.

[11] J. Dorsey, S. Searles, M. Ciraula, S. Johnson, N. Bujanos, D. Wu, M. Braganza, S. Meyers, E. Fang, and R. Kumar. An integrated quad-core opteron processor. In *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International*, pages 102–103, Feb. 2007.

[12] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5), 1998.

[13] J. R. Goodman. Using cache memory to reduce processor-memory traffic. In *ISCA'83*, Los Alamitos, CA, USA, 1983.

[14] M. Gschwind. Chip multiprocessing and the cell broadband engine. In *Proceedings of the 3rd conference on Computing frontiers*, pages 1–8. ACM, 2006.

[15] A. Gupta, W. Weber, and T. Mowry. Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. In *In International Conference on Parallel Processing*, pages 312–321, 1990.

[16] E. Hagersten and M. Koster. Wildfire: A scalable path for smps. In *HPCA '99: Proceedings of the 5th International Symposium on High Performance Computer Architecture*, page 172, Washington, DC, USA, 1999. IEEE Computer Society.

[17] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive nuca: near-optimal block placement and replication in distributed caches. In *ISCA '09: Proceedings of the 36th Annual International Symposium on Computer Architecture*, pages 184–195, New York, NY, USA, 2009. ACM.

[18] 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.*, 11(4), 1993.

[19] 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 *Proceedings of the 34th Annual International Symposium on Computer Architecture*, 2007.

[20] IBM Staff. *PowerPC Microprocessor 32-bit Family: The Programming Environments*, February 2000.

[21] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume: 1*, November 2008.

[22] 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 *Proceedings of the International Symposium on Computer Architecture*, pages 140–151, June 2009.

[23] J. H. Kelm, D. R. Johnson, S. S. Lumetta, M. I. Frank, and S. J. Patel. A task-centric memory model for scalable accelerator architectures. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 77–87, September 2009.

[24] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[25] A. Krishnamurthy, D. E. Culler, A. Dusseau, S. C. Goldstein, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in Split-C. In *SC'93*, 1993.

[26] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The stanford flash multiprocessor. In *ISCA '98: 25 years of the International Symposia on Computer Architecture (selected papers)*, pages 485–496, New York, NY, USA, 1998. ACM.

[27] J. Laudon and D. Lenoski. The SGI Origin: a ccNUMA highly scalable server. *SIGARCH Comput. Archit. News*, 25(2):241–251, 1997.

[28] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden. IBM POWER6 microarchitecture. *IBM J. Res. Dev.*, 51(6):639–662, 2007.

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

[30] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.

[31] A. Mahesri, D. Johnson, N. Crago, and S. J. Patel. Tradeoffs in designing accelerator architectures for visual computing. In *Proceedings of the International Symposium on Microarchitecture*, 2008.

[32] M. M. K. Martin, M. D. Hill, and D. A. Wood. Token coherence: decoupling performance and correctness. In *ISCA '03: Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 182–193, New York, NY, USA, 2003. ACM.

[33] A. Moshovos. RegionScout: Exploiting coarse grain sharing in snoop-based coherence. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 234–245, Washington, DC, USA, 2005. IEEE Computer Society.

[34] B. W. O'Krafka and A. R. Newton. An empirical evaluation of two memory-efficient directory methods. In *ISCA '90: Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 138–147, New York, NY, USA, 1990. ACM.

[35] A. Raghavan, C. Blundell, and M. M. K. Martin. Token tenure: Patching token counting using directory-based cache coherence. In *MICRO '08: Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 47–58, Washington, DC, USA, 2008. IEEE Computer Society.

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

[37] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and typhoon: user-level shared memory. In *ISCA '94: Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–336, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[38] S. Rusu, S. Tam, H. Muljono, J. Stinson, D. Ayers, J. Chang, R. Varada, M. Ratta, and S. Kottapalli. A 45nm 8-core enterprise xeon processor. In *Solid-State Circuits Conference - Digest of Technical Papers, 2009. ISSCC 2009. IEEE International*, pages 56–57, Feb. 2009.

[39] B. Saha, X. Zhou, H. Chen, Y. Gao, S. Yan, M. Rajagopalan, J. Fang, P. Zhang, R. Ronen, and A. Mendelson. Programming model for a heterogeneous x86 platform. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, Dublin, Ireland, Jun 2009. ACM.

[40] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: a low overhead, software-only approach for fine-grain shared memory. In *ASPLOS'96*, 1996.

[41] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8), 1990.

[42] S. J. Vaughn-Nichols. Vendors draw up a new graphics-hardware approach. *Computer*, 42(5):11–13, 2009.

[43] E. Witchel, J. Cates, and K. Asanović. Mondrian memory protection. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 304–316, New York, NY, USA, 2002. ACM.

[44] H. Wong, A. Bracy, E. Schuchman, T. M. Aamodt, J. D. Collins, P. H. Wang, G. Chinya, A. K. Groen, H. Jiang, and H. Wang. Pangaea: a tightly-coupled ia32 heterogeneous chip multiprocessor. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 52–61, New York, NY, USA, 2008. ACM.