Proximity-Aware Directory-based Coherence for Multi-core Processor Architectures

Jeffery A. Brown
Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0404

Rakesh Kumar
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
Urbana, IL 61801

Dean Tullsen
Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0404

ABSTRACT

As the number of cores increases on chip multiprocessors, coherence is fast becoming a central issue for multi-core performance. This is exacerbated by the fact that interconnection speeds are not scaling well with technology. This paper describes mechanisms to accelerate coherence for a multi-core architecture that has multiple private L2 caches and a scalable point-to-point interconnect between cores. These techniques exploit the differences in geometry between chip multiprocessors and traditional multiprocessor architectures.

Directory-based protocols have been proposed as a scalable alternative to snoop-based protocols. In this paper, we discuss implementations of coherence for CMPs and propose and evaluate a novel directory-based coherence scheme to improve the performance of parallel programs on such processors. *Proximity-aware coherence* accelerates read and write misses by initiating cache-to-cache transfers from the spatially closest sharer. This has the dual benefit of eliminating unnecessary accesses to off-chip memory, and minimizing the distance over which communicated data moves across the network. The proposed schemes result in speedups up to 74.9% for our workloads.

Categories and Subject Descriptors

C.1.2 [**Processor Architectures**]: Multiprocessors; C.4 [**Performance of Systems**]: Performance attributes

General Terms

Design, Performance

Keywords

Chip Multiprocessors, Coherence

SPAA'07, June 9–11, 2007, San Diego, California, USA.

1. INTRODUCTION

Multi-core architectures (also called chip multiprocessors, or CMPs) are becoming increasingly popular as a means to enhance the throughput and power efficiency of processors. The initial implementations of multi-core technology from the high-performance general purpose processor industry have had a modest number of cores (two to four) [13, 2, 12, 23]; however, the industry is clearly headed to larger and larger numbers of cores on each processor. In fact, Sun already has offerings with eight cores on a die [14].

As the number of cores on a processor die grows, and as more and more shared memory programs are run on these processors, cache coherence is fast becoming a central issue for multi-core performance. Cache coherence is the mechanism that allows us to maintain the value of a given memory block in multiple processor caches at one time, and still maintain system-wide agreement about the value of that memory block at any point in program execution. Cache coherence requires that we maintain enough information about the possible locations of the data across various caches so that we can find the data when a new consumer requests a copy, and so that we can communicate the obsolescence of cached values when someone writes to a memory block which is being shared. Since cache coherence involves significant amount of communication over wires, wire speed and bandwidth are the primary limiters to the performance and scalability of cache coherence. As interconnection speeds fail to scale well with processor speeds [10] and as interconnection bandwidth overhead (in terms of area and power) worsens over time [15], novel mechanisms and policies are needed for accelerating coherence for a given wire speed and bandwidth.

This paper examines the design of effective coherence mechanisms for a multi-core architecture that has multiple L2 caches, a directory-based cache coherence protocol, and a scalable point-to-point interconnect. Multi-core implementations with small numbers of cores can use a snoop-based cache coherence protocol, which relies on a broadcast-based interconnect – typically a bus – to implicitly provide global communication of value and state changes, and to provide a single ordering of all accesses. However, broadcast-based interconnects such as buses scale poorly as the number of cores grows, and we will be forced to move to scalable interconnects and directory-based coherence solutions (which work without the need for any broadcast medium). In this paper, we begin the process of exploring directory proto-

cols for multi-core processors, and tuning the protocols for the unique needs and opportunities provided by chip multiprocessors. We propose and evaluate novel directory-based coherence schemes that improve the performance of parallel programs on such processors.

We show in this paper that simply implementing traditional directory protocols within a chip does not provide the most effective solution. This is because those protocols were designed to work under very different topological assumptions than those on a chip multiprocessor. Some examples of those assumptions, true on a multiple-chip multiprocessor but *not* on a single-chip multiprocessor, are that, for a given requester and home node:

1. The home node's main memory and the home node's directory are close to each other, and about the same distance (latency) from the requesting node;

2. Once a request has reached the home node directory, the home node memory is closer than the caches of other nodes; On a traditional multiprocessor, the biggest latency barriers are between nodes. On a CMP, the latencies between nodes are small, and the dominant latency barrier is to off-chip memory, regardless of which node it is associated with.

3. The relative distance between nodes varies little.

On a traditional multiprocessor, the distances (in latency) from a given node to the nearest node and to the farthest node are often within a factor of two of each other, because the latency is dominated in most cases by the off-chip and off-board latencies. On a chip multiprocessor, although all latencies are smaller in absolute terms, the relative latencies vary significantly. That is, a core six hops away takes significantly longer to access than one a single hop away.

This paper proposes *proximity-aware directory-based coherence*. Proximity-aware coherence is based on the observation that, while a cache line can reside in multiple caches in the shared state, there is no guarantee that the line will be present in the cache of the home node corresponding to that line. Thus, instead of making the home node always source the data, from very slow off-chip memory if it happens to not exist in the home node's caches, proximity-aware coherence enables the closest sharer to source the data on a read or write request. This results in decreased latency and bandwidth utilization, especially when the line is not present in the home node's cache but is present in the "shared" state in some other cache. Even when the line is present in the home node's cache, proximity-aware coherence can still help in reducing the bandwidth pressure on the interconnect.

Our evaluations for a 16-core chip multiprocessor with MESI coherence show that proximity-aware coherence results in up to 74.9% improvement in performance. Average performance improvement was 16% for our workloads.

The rest of the paper is organized as follows. Section 2 discusses prior related work. Section 3 describes the baseline architecture and the the baseline directory-based coherence protocol. Section 4 describes the novel coherence implementations. Methodology for our evaluations is presented in Section 5. Our evaluations and results are described in Section 6. Section 7 concludes.

2. RELATED WORK

Directory-based protocols [18, 17] have been proposed for scalable coherence on distributed shared memory multiprocessors. The coherence protocol for SGI Origin [17] was a four-state "MESI" (Modified, Exclusive, Shared, Invalid) protocol assuming sequential memory consistency. Directory coherence for the DASH multiprocessor [18], on the other hand, utilized a three-state protocol assuming weak memory consistency. Both these machines featured distributed shared memory (DSM) and implemented cache coherence with distributed directories that were stored at each node, but off-chip from the processor itself.

While directory-based coherence has been popular for DSMs, it has not been studied in much detail for CMPs utilizing private L2 caches. Most of the current CMP implementations and proposals either have shared L2 caches with directories [3] or private L2 caches with snoop-based coherence [16]; neither of those approaches scale well as the number of cores increases. Huh, *et al.* [11] discuss a CMP model with directory-based coherence for their study of optimal degree of sharing for NUCA caches. They assume a central directory with constant access time. Zhang and Asanovic [24] also consider directory-based coherence for one of their CMP models; they assume directory caches distributed by cache set indices.

Our implementations of directory-based coherence assume a distributed directory, with an on-chip directory controller and directory cache at each node. Caching the directory state was proposed [8, 21] as a means of reducing the memory overhead entailed by directories. Michael and Nanda [20] propose integrating directory caches inside the coherence controllers to minimize directory access time. Acacio, *et al.* [1] study the impact of having first level directory on-chip caches.

One of our proposed policies, proximity-aware coherence, relies on location awareness to source shared data. CC-NUMA and COMA architectures [5, 25] also use spatial awareness for minimizing latencies. However, those architectures improve performance by retaining local copies of data that would otherwise require remote access. Proximity-aware coherence, on the other hand, does not require changing the mapping of data to sharers.

While we assume a conventional interconnect, Eisley and Peh [7] move much of the coherence-related control and data storage into the network. Traditional sharer sets are replaced by *virtual trees* maintained within the routers themselves, with routers serving as active participants in coherence decisions. Through different mechanisms, their work and ours realize similar latency benefits on parallel workloads.

Chang and Sohi [4], seeking to combine the best attributes of both shared and private L2 caches, introduce a scheme for globally managing data placement, replication, and migration across the caches of all cores; even single-threaded workloads benefit through the use of neighboring cache resources. New policies manage storage through a centralized directory-like structure suitable for coordinating small numbers of cores. While their cache-management scheme is orthogonal in concept to that of a coherence protocol, both it and our own optimizations improve performance by avoiding off-chip memory accesses.

Token Coherence [19] provides a framework for decoupling policies for coherence performance and correctness, the for-

mer implemented as *performance protocols*, and the latter ensured by *correctness substrates*. The specific performance protocol considered in that work, *TokenB*, broadcasts requests in order to avoid resorting to main memory accesses unnecessarily. Our proximity-based coherence scheme seeks the same goal, and could itself be expressed as a particular performance protocol on top of a token-based system.

3. A CMP ARCHITECTURE WITH DIRECTORY-BASED COHERENCE

This section describes the processor architecture that we use for our study. We also describe the baseline implementation of directory-based coherence for this architecture.

3.1 Architecture

The architecture is a chip multi-processor consisting of 16 cores arranged as a 4×4 mesh of tiles. Each tile contains an in-order core with private Level-1 instruction and data caches, a private unified Level-2 cache, a directory controller, and a network switch connecting to the on-chip network, as shown in Figure 1. Memory – both directory and regular program memory – is accessed through on-chip memory controllers, with one located on each tile. Each memory channel provides access to a different range of physical memory addresses. The architecture resembles a conventional mesh-connected multi-chip multiprocessor. The optimizations considered in this paper exploit, among other things, the non-uniform latencies between cores inherent in a mesh architecture. This non-uniformity (in particular, the ratio of the latency for communicating with a distant node to that for communicating with an adjacent node) will only increase with larger CMPs; as wire delays increase, the absolute difference between these latencies will increase as well. Note that our baseline is only a canonical architecture – the techniques outlined in this paper can apply to any system with multiple (L2) caches, whether those caches each serve a single core, or each serve a cluster of cores.

We contrast this architecture with that of a more traditional multiprocessor (Figure 2), which is composed of multiple chips and typically multiple boards. A coherence protocol that is designed for the traditional multiprocessor will not exploit the geometries of a chip multiprocessor well; it assumes that, for a given memory address, node memory is close to the home node directory, and that remote caches are far away. Neither assumption is true on a chip multiprocessor.

We assume that the L2 cache is tightly coupled to the rest of the tile. The tag and status storage are kept separate from the data arrays and close to the core and router for quick tag resolution.

Accesses to resources on other tiles require that traffic travel through the network switch and over the on-chip network, experiencing varying access latencies depending on the distance between the tiles and the loads on the links between them.

We also assume *directory caches* (DC), one per node, which cache directory state as it is used by the directory controllers. Instead of accessing the off-chip directory memory for each coherence operation, the directory controller accesses the DC instead. All state changes are made to the contents of the DC itself. Only when there is a miss in the DC does the directory controller need to make an off-chip

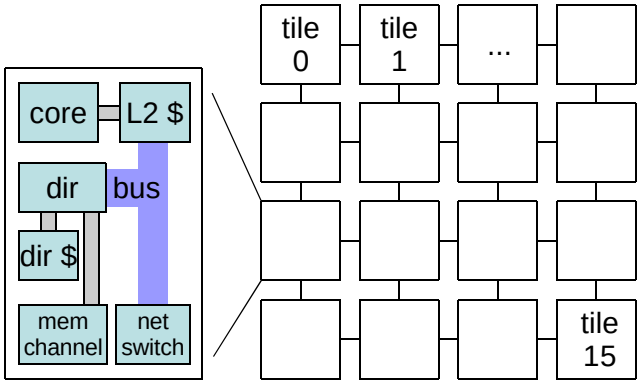


Figure 1: The baseline chip multiprocessor architecture, with 16 tiles. Each tile contains a core (with L1 caches), an L2 cache, a directory controller, a directory cache, a network switch, and a memory channel.

access to determine the coherence state of a line; given that only a single node is designated as the point of contact for directory information for any given cache line (its "home node"), the corresponding coherence state cannot exist in another node's DC, so the directory caches themselves need not be coherent.

The directory cache is organized as a set-associative cache where each cache line holds state corresponding to multiple contiguous memory blocks to exploit spatial locality. A new entry is created in the DC for every line that is loaded. The directory cache replacement policy is LRU.

Note that the above organization decouples L2 tags from the coherence directory tags. This enables low-latency access to the coherence state of a line, even when the line is not present in the L2 of the home node.

A generic four-state "MESI" protocol [17] adapted for CMPs is used as the baseline protocol for on-chip data coherence. The MESI protocol is named for the four states available to a line in a particular cache:

M (modified – this cache has a modified version of the data, and no other cache has a copy), E (exclusive – this cache has a clean copy, but no other cache has a copy), S (shared – this cache has a clean copy, but other caches may also have a copy), and I (invalid). In a directory protocol, the home node must keep track of the global state of each line, as well as the set of sharers of each line, in order to coordinate writes to shared data.

Our proposed implementation is a variant of this protocol. We now describe the details of our baseline coherence protocol.

3.2 Baseline Coherence Protocol

To illustrate the directory coherence protocol, first consider how an L1 read miss traverses the memory hierarchy:

- **Requester** - If the requested location is present in the requester's L2 cache, the cache simply supplies the data and no state change is required at the directory level. If there is an L2 miss, a request is sent to the home node which is associated with the desired memory address.

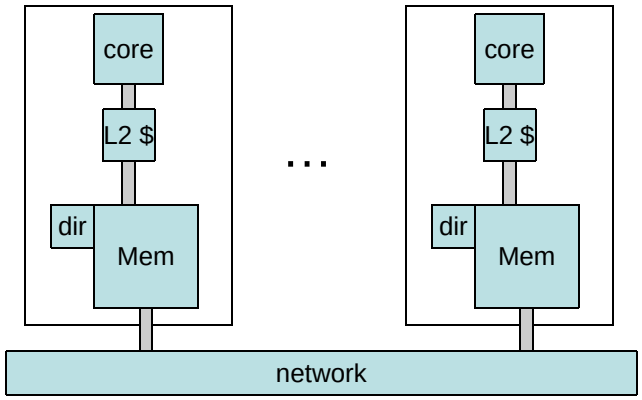


Figure 2: A more traditional multiprocessor (multi-chip, multi-board).

- **Home node** - The directory controller accesses the node's directory cache, and directory memory if necessary, to examine the coherence state for the desired cache line. If the home node itself is indicated as a sharer of the desired data, the directory controller forwards the request to the local L2 cache for service. Otherwise, if the coherence state indicates the block is shared (and hence unmodified), a read from the main memory attached to the home node is initiated, and the result subsequently sent to the requester. If the coherence state instead indicates that the block is dirty, hence held exclusively by one node, the request is forwarded to that remote node's L2 cache for service.

- **Remote node** - The node with the dirty copy replies with the most up-to-date version of the data, which is sent directly to the requester. In addition, a sharing write-back message is sent to the home node to update main memory, and to change the directory state to indicate that the requester and remote nodes now have shared copies of the data.

Next, consider the sequence of operations that occurs when a location is written. We focus here on the case of a write miss.

- **Requester** - A read-exclusive request is sent to the home node to retrieve the cache line and gain ownership.

- **Home node** - The home node can immediately satisfy an ownership request (from its L2 or memory) for a location that is in the uncached state. If a block is in the shared state, then all cached copies must be invalidated. The line in the directory cache corresponding to the request address indicates the nodes that have the block cached. Invalidation requests are sent to these nodes. For weakly consistent processors, the home node would concurrently send an exclusive data reply to the requesting node (though data need not be sent for upgrade misses), and then wait for invalidate ACKs from the other potential sharers. For strongly consistent processors, the home waits until it has received invalidate ACKs from all sharers before replying to the requester and granting ownership of the block. For both types of consistency, a request is considered serviced at the home node only after invalidate ACKs have been received from all previous sharers. If, instead of the shared state, the directory indicates that the desired block is initially dirty, then the read-exclusive request must be forwarded to the sole owner, as in the case of a read.

- **Remote node** - If the directory had initially indicated that the memory block was shared, then the remote nodes are each sent an invalidation request to eliminate their copy. Upon receiving the invalidation, each remote node invalidates the corresponding line, and then replies to the home with an acknowledgment. If the directory had instead indicated the block was initially dirty, then the sole owner is sent a read-exclusive request. As in the case of the read, the remote node responds directly to the requesting node with the data, and sends the home node a message acknowledging transfer of ownership.

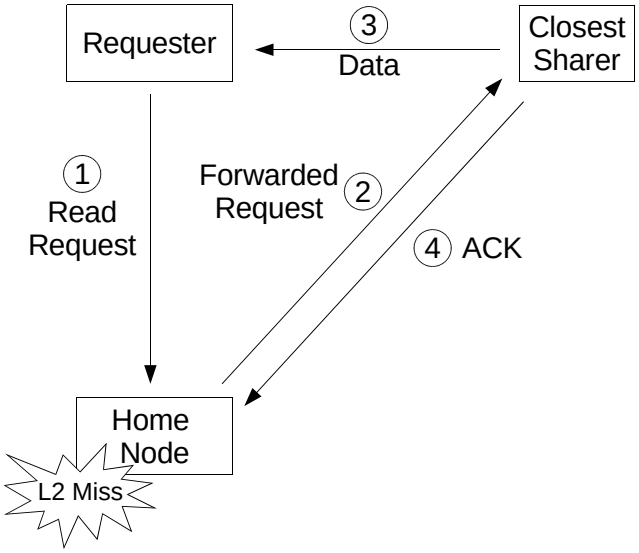
4. ACCELERATING COHERENCE VIA PROXIMITY AWARENESS

Proximity-aware coherence encompasses the recognition of two facts particular to chip multiprocessors. First, that an on-chip cache access – even to a remote node – is always closer than an off-chip memory access. Second, that if there are multiple sharers of the data, selecting the right source to provide the data (one who is *close* to the requester) can reduce both latency and bandwidth utilization. We present a novel directory-based coherence schemes that exploits these properties.

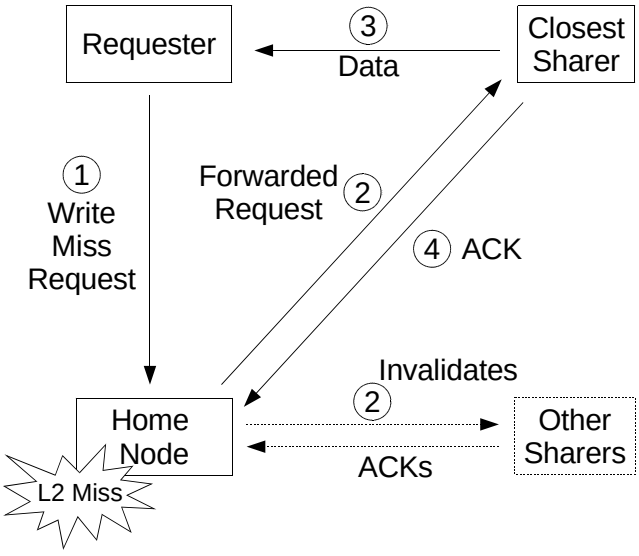
For conventional directory-based coherence, a read or write miss to a line that is in the shared or the uncached state always results in the home node sourcing the data. However, the data may not be in the home node's L2, and accesses to off-chip memory are expensive. Proximity-aware coherence relies on the observation that even if data is not present in the home node's L2, it might still exist in shared state in some other L2 on the chip. This relaxes the constraint of the home node always sourcing the data in such scenarios and instead allows other sharers to source the data.

To illustrate the proximity-aware coherence protocol, first consider how a read miss traverses the memory hierarchy. Initial actions at the requester remain the same as in the baseline. The protocol differs at the home node and at the remote node.

- **Home node** - The home node examines the directory state of the memory location. If the block is dirty, the request is forwarded to the exclusive owner, as in the baseline. If the block is uncached, the home node services the request from main memory, as in the baseline. Otherwise, the block is clean; if the home node is indicated as a sharer, the request is forwarded to the home node's L2 cache for service. If the home node is not a sharer, but other nodes are, then the directory controller selects one or more of the potential sharers to ask for the data. The home node sends a message to the closest of these sharers, requesting it forward the data to the original requester. If the protocol supports multiple sharer requests, others are contacted in turn, until one if found to have the data or the maximum number of requests has been made.



(a) Read Miss



(b) Write Miss

Figure 3: ***Proximity-aware coherence***

The directory state is not updated until the directory controller either receives an ACK from a remote node indicating that the desired data has been forwarded to the original requester, or every proximity-based request has been NACKed, at which point the controller gives up on future such requests and falls back to requesting the data from main memory.

- **Remote node** - If the remote node is in the dirty state, the same actions take place as in the baseline. However, if the data is shared and the remote node has been asked to forward the data to the requester (because it is the closest sharer), the remote node sources data directly to the requester and sends an ACK back to the home node. If the requested line is not in the remote node's L2 cache when the request from the home node reaches it, the remote node responds with a NACK.

Now consider the sequence of operations that occurs when there is a write miss. Again, initial actions remain identical at the requester.

- **Home node** - If the requested line is in the home node's L2, the same actions take place as in case of the baseline coherence implementation.

  If the directory indicates that the block is dirty, then the read-exclusive request must be forwarded to the exclusive owner, as in the case of a read.

  If the line is in shared state AND the line is not in the home node's L2, forward-exclusive requests are sent in turn to one or more potential sharers, as in the read-miss case, eventually falling back to reading from main memory if the forwarding requests fail. Any potential sharers which were not sent a forward-exclusive request are then sent invalidate requests, in parallel. Directory state is not updated until replies are received from all sharers.

- **Remote node** - If the directory had indicated a dirty state, then the exclusive owner receives a read-exclusive request. The coherence transactions in that case are identical to the baseline coherence implementation.

  If the directory had indicated that the memory block was shared, and the remote node is the subject of a forward-exclusive request, it forwards a copy of the data (if present) to the original requester, invalidates its own copy, and responds with an ACK to the home node. If the remote node does not have the data, a NACK is sent.

Proximity-aware coherence attempts to ensure that if data is anywhere in the CMP in the appropriate state, a read or write request can be satisfied without the need to do off-chip memory access at the home node. This decreases the latency of coherence. Also, proximity-aware coherence should result in decreased overall bandwidth utilization since the control messages are much smaller than data messages: even though the number of control messages increases, the larger data-carrying responses will travel shorter distances. The latency/bandwidth tradeoffs depend on the spatial location of the nodes and the relative size of the data messages versus control messages.

The implementation of proximity-aware coherence is a straightforward, safe extension of the mechanisms present in the baseline system; no additional storage is required specifically to support it, and the additional state transitions within the directory and cache controllers don't require significant complexity to handle. Correctness of the underlying protocol is not affected, since 1) the proximity-aware extensions are applied only to clean data (possibly after invalidates), 2) the resulting cache blocks are left clean, and 3) the corresponding directory entry is always updated with a superset of the actual sharers before processing the next request for the subject memory block.

Note that proximity-aware coherence is not applicable for upgrade misses, as no data blocks need to be transmitted in that case. Proximity-aware coherence will work whether the processor supports strong (e.g., sequential) or weak consistency.

While proximity-aware coherence as introduced forwards a single data request to the sharer nearest each requester, a variety of request policies are possible. We explore two additional considerations: how many sharers to send proximity forwarding requests before giving up, and what metric is used to order candidate sharers.

Forwarding requests to multiple potential sharers pits the benefits of avoiding unnecessary off-chip memory accesses when some of the advertised sharers lack copies of the data, against the bandwidth and latency costs of the additional control messages. (It is possible for nodes listed as sharers at the directory to no longer contain copies of the data, because it has been evicted from the cache). We consider forwarding requests to the "nearest" one, two, or three nodes before falling back to an external memory access.

We consider three node selection policies, which are used to order the potential recipients of a proximity forwarding request. The first, *near*, orders candidates by the Manhattan distance from each remote node to the requester. The second, *via*, orders candidates by the sum of the Manhattan distances from the home node to each remote node, and from each remote node to the requester. The final policy, *rand*, simply chooses nodes from the sharer set at random.

In reporting results, we combine the node selection policy and try-count, e.g., a policy which attempts to source data from two remote nodes nearest to the requester is referred to as *near2*. We refer to the policy of consulting a single sharer at random before reverting to main memory simply as *rand*.

5. METHODOLOGY

We perform all our evaluations using a modified version of RSIM [22], a discrete event-driven multiprocessor simulator. RSIM has detailed models of the core, the primary caches, the L2 cache and the 2D mesh network. However, RSIM models a distributed shared-memory multiprocessor. Appropriate changes were made to simulate on-chip multiprocessing with on-chip networks. Significant changes were required to model the proposed directory-based coherence implementations. Home nodes are assigned based on a "first-touch" policy [17]. This policy ensures that the home node is assigned to a node that is likely to be an active sharer of the data.

We assume 70 nm technology (based on BPTM [6]) and model a 3-cycle network hop – that includes the router latency and an optimally-buffered 5 mm inter-tile copper wire on a high metal layer. The latencies are modeled by assuming a 24 FO4 processor clock cycle [9]. Memory channels are assumed to have RDRAM interfaces. Table 1 lists the important system parameters used in the experiments.

Component	*Parameter*
Processor Model	in-order
Issue-width	dual-issue
Instruction window (entries)	16
Load/store queue (entries)	16
Branch predictor	bimodal (2K)
Number of integer ALUs	2
Number of FP ALUs	1
Cache Line Size	64B
L1 I-Cache Size/Associativity	32KiB/4-way
L1 D-Cache Size/Associativity	32KiB/4-way
L1 Load-to-Use Latency	1 cycle
L1 Replacement Policy	Pseudo-LRU
L2 Cache Size/Associativity	256KiB/8-way
L2 Load-to-Use Latency	6 cycles
L2 Replacement Policy	Pseudo-LRU
Directory Cache Size/Associativity	16KiB/4-way
Directory Cache Load-to-Use Latency	1 cycle
Directory Cache Replacement Policy	LRU
Network Configuration	4×4 mesh
One-hop latency	3 cycles
Worst-case L2 hit latency (contention-free)	48 cycles
Number of Memory Channels	16 (1 per L2)
Directory Memory Latency	30 cycles
External Memory Latency	256 cycles

Table 1: Architecture Detail

The workloads we use to evaluate our coherence mechanisms are listed in Table 2. These are all parallel workloads and represent a wide variety in their computation-communication ratio. The applications also have varying degrees of sharing and synchronization, and represent diverse application domains.

6. ANALYSIS AND RESULTS

In this section, we present our evaluation of the proposed *proximity-aware coherence* policies. All our evaluations assume a sequentially consistent processor with MESI baseline protocol as described in Section 3.2. Evaluations are presented for 256 KB L2 caches unless otherwise noted.

Proximity-aware coherence seeks to eliminate accesses to memory for any data held in an on-chip cache, and to minimize the distance traveled for any cache-to-cache transfers. The first goal, in particular, exploits the unique property of chip multiprocessors (over traditional distributed shared memory multiprocessors) that the latency of communication between compute nodes (and hence the latency of seeking data from a peer core's cache) is significantly lower than the latency of seeking data from one's own local memory.

The effectiveness of this technique, then, will depend in large part on how often a requested line does not reside in the queried home node (which would typically result in a memory access), yet does reside elsewhere on chip in another cache. Figure 4 shows the fraction of read misses to a shared line (i.e., the corresponding directory entry has the line in shared state) that do not have the home node listed as a sharer. The higher the bars, the higher the opportunity for initiating proximity-aware cache-to-cache transfers. These results include the effect of the first-touch home node assignment policy, which increases the chance that the home

Benchmark	*Benchmark Suite*	*Problem Size*
APPBT	NAS	$64 \times 64 \times 64$, 30 iterations
FFT	SPLASH2	64K points
LU	SPLASH2	256×256 matrix, 8×8 blocks
MP3D	SPLASH	48000 nodes, 20 timesteps
Ocean	SPLASH2	130×130 array, 10^{-9} error tolerance
QuickSort	TreadMarks	512K integers
Unstructured	Wisc	Mesh.2K, 5 timesteps

Table 2: Workloads

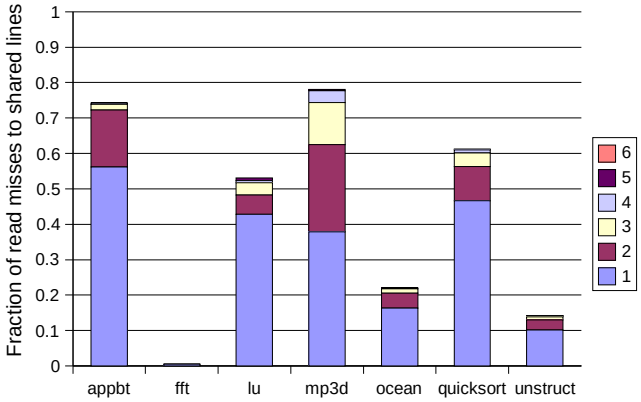


Figure 4: Fraction of reads misses to shared lines for which the home node is not a sharer, but another node is. The higher the bars, the more potential benefit from proximity-aware coherence. The segments of each bar indicate the minimum number of hops from a sharer to the requester, for each miss.

node is an active sharer. In the absence of this policy, the potential for proximity-aware coherence would be even greater.

Note also, in Figure 4, that the results for each benchmark are broken down in terms of the distance from the requester of the closest node that is listed in the directory as a sharer. So, if the requester id is 0 (top left corner of the chip), and the closest sharer for the requested line, as listed in the directory, is node 15 (bottom right corner of the chip), it adds to the stacked bar corresponding to 6 (because the two nodes are six network hops away).

There are two things to note in this graph. First, we can see that it is quite common for shared data to not be found in the home node, but exist elsewhere on the chip. In fact, this happened for nearly half of read misses to shared lines (43%). The actual percentage does depend significantly on the data access patterns, however, and therefore we observe a significant variance by benchmark. For example, for *unstruct* and *ocean*, the requester is often the home node as well and thus only 14% and 22% of read misses to shared lines, respectively, have the home node not listed as a sharer. *fft* experiences a very small number of read misses to shared lines, essentially all of which are shared at the home node. Data migration patterns are more aggressive for *appbt* and *mp3d*, resulting in a high fraction of read misses to shared lines with non-home sharers (74% and 78%, respectively).

Another observation from the graph is that most of the requests can be satisfied by nodes that are one hop away. While this is not very surprising (as the average distance between two nodes in a 16×16 tiled processor is only three hops), it does mean that proximity-aware coherence, when done right, can potentially result in significantly reduced average L2 miss latency.

While Figure 4 shows the potential for proximity-aware coherence, the numbers do represent an upper bound (albeit likely a tight one), because there is no guarantee we will find the closest sharer on the first try. Evictions in the individual caches cause the sharer set in the directory to always be a superset of the actual sharers. Thus, while this result gives an accurate account of how often a sharer exists, there will be times (depending on the eviction rate) when finding that sharer is not easy.

An more direct measure of success for this technique coherence is the reduction in average L2 miss latency, when proximity-aware coherence is applied. Figure 5 shows the average latency of an L2 miss for a multi-core processor enhanced with proximity-aware coherence. The results are normalized against L2 miss latencies for the processor with baseline coherence. As can be seen, proximity-aware coherence can often result in significant reduction in latency of coherence operations. Latency reductions of up to 79% (*quicksort*) were observed. Average latency reduction was 24.6%.

Proximity-aware coherence has two distinct enhancements – elimination of unnecessary memory accesses, and the minimization of distance traveled by shared data. These results indicate that the former is clearly the more important factor, in these experiments, for performance. This is reflected in the fact that all three distance protocols achieve strong gains, but the difference between them is slight.

However, it is still worth noting that *via* performs slightly better than *random* and *near*. *Random (rand)* represents a baseline distance-oblivious heuristic. *Near* minimizes the distance from the sharer to the requester, but in many cases the total hops traveled from the home node to the requester (via the sharer) is greater than the minimal number of hops from the home node to the requester. This is because the control message must travel from the home node to the sharer (which, although presumably close to the requester, may be on the opposite side from the home node). So while the distance traveled by the data is minimized, the total distance is not. The *via* enhancement greatly increases the likelihood that the combined distance traveled by the control message and the data is no greater than the minimal distance. This results in reduced overall L2 miss latency.

An associated effect of performing a spatial optimization (e.g., *near* and *via*) is that is that the average bandwidth pressure on the interconnect is reduced. This is because the sourced data now traverses a shorter distance, on average, from the sharer to the requester. Figure 6 shows the average number of bytes of data transferred per L2 miss for the three proximity-aware policies. We observe up to 6% reduction in bandwidth requirements for *via* and *near* over *random*. In a system where contention for some links was high, we would

expect that reduced bandwidth to translate more directly into latency reductions (due to reduced queueing). However, the SPLASH2 benchmarks put very little overall pressure on our assumed interconnect.

In fact, there are several reasons why we believe these results understate the potential gains of the proximity-aware coherence, and the spatial optimizations (e.g., *via*) especially. The SPLASH2 benchmarks have small working sets relative to our caches. This has two effects – contention for links is low (as mentioned), and the number of L2 misses per instruction is generally quite small. This means that the sensitivity of these results to the actual L2 miss latency is much lower than most realistic parallel commercial workloads. Additionally, we will see even greater gains as the number of cores (and the maximum and average distance between cores) increases. With future many-core systems (with tens or possibly hundred of cores), the importance of proximity-aware coherence and the right policy to choose will only increase.

In spite of the above limitations, the significant reduction in average L2 miss latency through proximity-aware coherence does translate into improved system performance. Figure 7 shows the performance of the three proximity-aware policies normalized against the baseline coherence protocol. As can be seen, proximity-aware coherence can result in speedups up to 75% with an average speedup of 16%. The *via* policy results in the highest system performance.

We also evaluate the impact of the *near* and *via* policies with try-counts of two and three, i.e., retrying one or two additional potential sharers, after the failure of the first proximity-read request, before falling back to the use of main memory. The use of additional requests increases the success rate of proximity-read sequences by a mean of 7% for *near2* and *via2*, and 9% for *near3* and *via3*. Unfortunately, these gains are offset by increases in bandwidth utilization, as well as some increases in the L2 miss-service latency (since retries are handled serially). While *lu* benefits from retries with an additional 1% decrease in mean L2 miss-service time and an additional 0.2% increase in speedup, the suite-wide impact of retries is negligible.

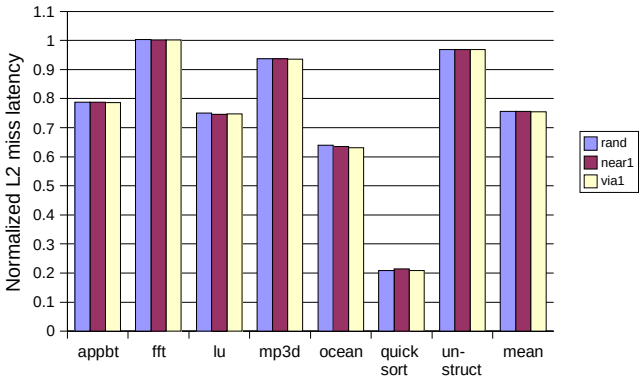


Figure 5: Mean L2 miss-service latency with proximity-aware coherence, normalized ($base = 1.0$)**. The *rand* policy queries a random on-chip sharer, *near1* queries one sharer closest to the requester, and *via1* queries one sharer with a minimum distance along the home-requester path.**

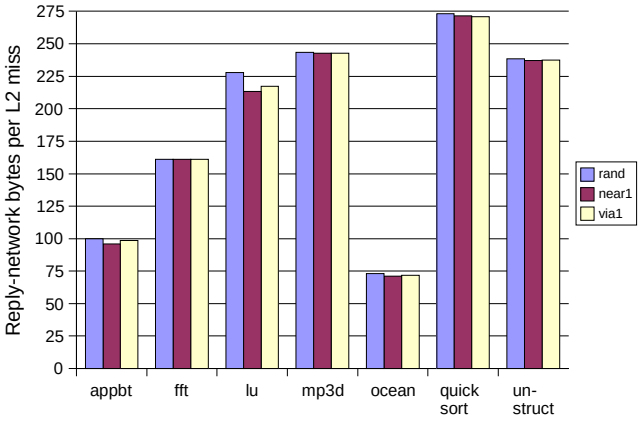


Figure 6: Reply-network utilization, in terms of *total* network traffic generated for each miss. (Each individual link transit counts toward the total.)

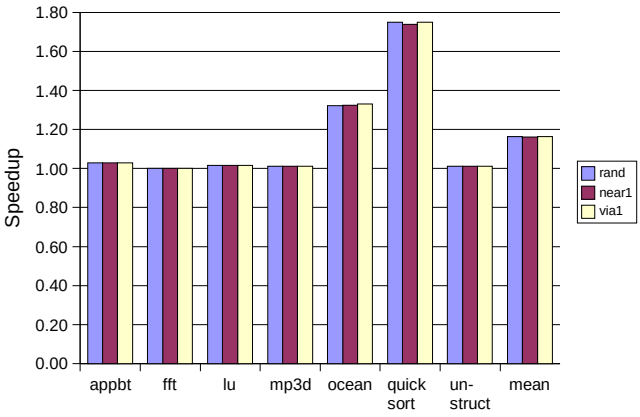


Figure 7: Speedup from using proximity-aware coherence for a sequentially consistent processor

7. SUMMARY AND CONCLUSIONS

This paper takes a first step in exploring the design of directory coherence protocols for chip multiprocessors. Future multi-core designs will feature multiple L2 caches and scalable interconnects. However, this work shows that simply implementing traditional directory protocols on a multi-core architecture does not provide the best design.

In particular, we show that multi-core specific customization of directory coherence (we call it *proximity-aware coherence*) can result in speedups up to 75% over a traditional directory coherence protocol applied directly to a multi-core processor. Average speedup of 16% was observed. Reduction in average L2 miss latency (for coherence misses) was even greater. Average miss latency reduction of up to 79% (and a suite-wide average reduction of 25%) was observed. More importantly, the results suggest that as multi-cores scale (both in terms of number of cores on the dies as well as the size of the dataset or working set of applications), the potential for proximity-aware coherence is going to only increase with time.

These results also demonstrate two other important points. First, that as we seek to use the coming wave of chip multiprocessors more effectively with parallel programs, perfor-

mance will become increasingly sensitive to coherence. Second, we find that it is beneficial to reconsider the design of coherence mechanisms to account for the unique characteristics of multi-core architectures.

Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful insights. This research was supported in part by NSF grant CCF-0541434 and Semiconductor Research Corporation Grant 2005-HJ-1313.

8. REFERENCES

[1] M. E. Acacio, J. Gonzalez, J. M. Garcia, and J. Duato. A novel approach to reduce l2 miss latency in shared-memory multiprocessors. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 25, Washington, DC, USA, 2002. IEEE Computer Society.

[2] AMD. http://www.amd.com/us-en/processors/productinformation/0 30_118_9484% ,00.html.

[3] L. 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-27*, 2000.

[4] J. Chang and G. S. Sohi. Cooperative caching for chip multiprocessors. In *Proceedings of the 33rd International Symposium on Computer Architecture*, pages 264–276, Washington, DC, USA, 2006. IEEE Computer Society.

[5] F. Dahlgren and J. Torrellas. Cache-only memory architectures. *Computer*, 32(6):72–79, 1999.

[6] Device Group. Predictive technology model. In *UC Berkeley Technical Report*, 2001.

[7] N. Eisley, L.-S. Peh, and L. Shang. In-network cache coherence. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 321–332, Washington, DC, USA, 2006. IEEE Computer Society.

[8] A. Gupta, W.-D. Weber, and T. C. Mowry. Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. In *ICPP (1)*, pages 312–321, 1990.

[9] A. Hartstein and T. R. Puzak. The optimum pipeline depth considering both power and performance. *ACM Trans. Archit. Code Optim.*, 1(4):369–388, 2004.

[10] R. Ho, K. Mai, and M. Horowitz. The future of wires. *Proceedings of the IEEE*, 89(4):490–504, 2001.

[11] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler. A nuca substrate for flexible cmp cache sharing. In *Proceedings of the 19th ACM International Conference on Supercomputing (ICS 05)*, June 2005.

[12] IBM. Power5: Presentation at microprocessor forum. 2003.

[13] Intel. http://www.intel.com/products/processor/coreduo/.

[14] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. In *IEEE MICRO Magazine*, Mar. 2005.

[15] R. Kumar, V. Zyuban, and D. M. Tullsen. Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling. In *Proceedings of International Symposium on Computer Architecture*, 2005.

[16] R. Kumar, V. Zyuban, and D. M. Tullsen. Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling. In *Proceedings of International Symposium on Computer Architecture*, June 2005.

[17] J. Laudon and D. Lenoski. The SGI Origin: a ccNUMA highly scalable server. In *ISCA '97: Proceedings of the 24th annual international symposium on Computer architecture*, pages 241–251, New York, NY, USA, 1997. ACM Press.

[18] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Henessy, M. Horowitz, and M. Lam. The stanford DASH multiprocessor. In *IEEE Computer*, 1992.

[19] M. M. K. Martin, M. D. Hill, and D. A. Wood. Token coherence: decoupling performance and correctness. In *Proceedings of the 30th annual international symposium on Computer architecture*, pages 182–193, New York, NY, USA, 2003. ACM Press.

[20] M. M. Michael and A. K. Nanda. Design and performance of directory caches for scalable shared memory multiprocessors. In *HPCA '99: Proceedings of the 5th International Symposium on High Performance Computer Architecture*, page 142, Washington, DC, USA, 1999. IEEE Computer Society.

[21] 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 Press.

[22] V. S. Pai, P. Ranganathan, and S. V. Adve. RSIM: An Execution-Driven Simulator for ILP-Based Shared-Memory Multiprocessors and Uniprocessors. In *Proceedings of the Third Workshop on Computer Architecture Education*, February 1997. Also appears in IEEE TCCA Newsletter, October 1997.

[23] Sun. UltrasparcIV: http://siliconvalley.internet.com/news/print.php/3090801.

[24] M. Zhang and K. Asanovic. Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 336–345, Washington, DC, USA, 2005. IEEE Computer Society.

[25] Z. Zhang and J. Torrellas. Reducing remote conflict misses: Numa with remote cache versus coma. In *HPCA '97: Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture*, page 272, Washington, DC, USA, 1997. IEEE Computer Society.