Memory coherence in the age of multicores

Mieszko Lis Keun Sup Shim Myong Hyon Cho Srinivas Devadas

Massachusetts Institute of Technology, Cambridge, MA, USA

***Abstract*—As we enter an era of exascale multicores, the question of efficiently supporting a shared memory model has become of paramount importance. On the one hand, programmers demand the convenience of coherent shared memory; on the other, growing core counts place higher demands on the memory subsystem and increasing on-chip distances mean that interconnect delays are becoming a significant part of memory access latencies.**

In this article, we first review the traditional techniques for providing a shared memory abstraction at the hardware level in multicore systems. We describe two new schemes that guarantee coherent shared memory without the complexity and overheads of a cache coherence protocol, namely execution migration and library cache coherence. We compare these approaches using an analytical model based on average memory latency, and give intuition for the strengths and weaknesses of each. Finally, we describe hybrid schemes that combine the strengths of different schemes.

I. Introduction

Although in the past decade heat dissipation limits have halted the drive to higher and higher core frequencies, transistor density has continued to grow [1], and CPUs with four or more cores have become common in the commodity and server-class general-purpose processor markets [2]. To further improve performance and use the available transistors more efficiently, architects are resorting to medium and large-scale multicores both in academia (e.g., Raw [3], TRIPS [4]) and industry (e.g., Tilera [5,6], Intel TeraFLOPS [7]), and industry pundits are predicting 1000 or more cores in a few years [8].

How will these massively-multicore chips be programmed? A shared memory abstraction stands out as a *sine qua non* for general-purpose programming: while architectures with restricted memory models (most notably GPUs) have enjoyed immense success in specific applications (such as rendering graphics), most programmers prefer a shared memory model [9], and small-scale commercial general-purpose multicores have supported this abstraction in hardware. The main question, then, is how to efficiently provide coherent shared memory on the scale of hundreds or thousands of cores.

The main barrier to scaling current memory architectures is the *off-chip memory bandwidth wall* [8,10]: off-chip bandwidth grows with package pin density, which scales much more slowly than on-die transistor density [1]. Since rising core counts mean higher memory access rates, these bandwidth limitations require that more of the data be stored on-chip to reduce the number of off-chip memory accesses: today's multicores integrate large shared (monolithic) last-level on-chip caches [2]. Such shared caches, however, do not scale beyond relatively few cores, and their power requirements of large caches (which grow quadratically with size) exclude their use in chips on the scale of 100s of cores (the Tilera Tile-Gx 100, for example, does not have a shared cache). Large multicores, then, are left with relatively small per-core caches; for programmability, these must present the abstraction of a unified addressing space, and, for efficiency, this abstraction must be managed automatically at the hardware level.

Architects implementing shared memory in large-scale multicores face a dilemma. On the one hand, the large scale implies that data should be cached close to the core that is operating on it, since even relatively efficient interconnects can take a long time to transfer data among per-core caches (in a 1024-core processor connected with a 32×32 2D mesh, 62 hops—likely of two or three clock cycles each not counting congestion—are required to cross the die). On the other hand, off-chip memory bandwidth limits mean that the implementation should keep as much data as possible in on-chip caches: since on-chip cache capacity is necessarily limited, this implies minimizing replication of the same data in multiple caches.

In the remainder of this article, we describe two traditional shared memory implementations—one which permits shared data to be replicated and one which does not—and two novel approaches: execution migration and library cache coherence. We then compare them using analytical models based on per-access *average memory latency*, and illustrate the conditions under which they perform well. Finally, we outline hybrid approaches which balance the strengths and weaknesses of each scheme.

II. Shared Memory Implementations

When multiprocessors consisted of a few separate processors on a common memory bus, keeping caches coherent was relatively simple: since all per-processor caches observed the memory-bus activity of other processors at the same time, caches in a write-through system could "snoop" on the bus and automatically invalidate or update entries when other processors modified them (e.g., [11]). But, with higher core counts and many clock cycles required to cross the chip, such atomic broadcasts have become impractical, and as the number of cores has become large, snoopy caches are no longer viable [12].

A. Directory Cache Coherence

On scales where bus-based mechanisms fail, the traditional solution to this dilemma has been directory-based cache coherence (DirCC): a logically central directory coordinates sharing among the per-core caches, and each core cache must negotiate shared or exclusive access to each cache line via a coherence protocol [13].

The simplest classical instance of directory coherence is the MSI protocol, in which each line in the cache can live in one of three states: **M**odified (meaning that the cache has the most up-to-date data and everyone else's copies are invalid), **S**hared (meaning that the cache has the most up-to-date data but can only access it for reads as there may be other copies in the system), or **I**nvalid (the data may not be accessed). The (logically central) directory keeps track of every line cached anywhere in the system, together with a list of sharers (caches that contain a copy of the data) who must be contacted if the cache line is to be written. From the standpoint of a single memory operation, accessing memory address *A* in the core cache *C* looks like this:

1) if the state of the cache line in *C* containing *A* allows the type of access requested (i.e., if the request is a LOAD and the cache line is in the S state, or the request is a STORE and the cache line is in the M state), complete the request;
2) otherwise:
    a) send a request from *C* to the directory *D* for the cache line containing *A* in S (for LOADs) or M (for STOREs) state; then,

b) if there are sharers incompatible with the request (any sharers for an M-state request or an M-state sharer for an S-state request),
   i) send a request from D to each sharer requesting that the cache line containing A be invalidated,
   ii) wait for an invalidate acknowledgement or writeback from each sharer, and
   iii) if a writeback was received, write the new value to the backing DRAM or the next-level cache;
c) retrieve the data from DRAM or the next-level cache (optionally bypass writeback data if a writeback was received);
d) send a reply message from D to C granting the requested access level;

3) complete the memory access and continue execution.

The main advantages of a directory-based coherence protocol are that (a) when data is only used by one core and fits in its cache, both reads and writes are fast as they can be accomplished locally, and (b) when data is written very infrequently but read often and concurrently by many cores, the fast local reads amortize the relatively high cost of the infrequent writes.

Outside of those scenarios, however, directory coherence protocols incur significant overheads. Consulting the directory requires an indirection (especially when multiple sharers must be invalidated), which significantly contributes to the latency incurred on a cache miss. Write requests especially can not only require invalidations in all other sharers but also cause cache misses (and the associated latency) should the sharers access the relevant cache line again. In addition, replication of data in per-core caches effectively reduces the total capacity of the on-chip caches in the system and increases the off-chip memory access rate, since significant parts of the caches store data that are also cached elsewhere on the chip.

Directories themselves also have significant costs in hardware. Since they must know about every line cached in any private core cache in the system, directory sizes must equal a significant portion of the *combined* size of the per-core caches: a directory that is too small will experience frequent evictions (which cause invalidations in all of the caches that contain the relevant data) that limit performance [13]. The protocol amounts to a complex set of cooperating state machines with many actors (directories, caches) and, because a wide range of combinations of final and in-progress states must be considered, the protocol can be tricky to implement and costly to verify [14].

An alphabet soup of protocols has followed MSI to optimize specific inefficiencies. **O**wned or **F**orward states allow data for new S-state requests to be satisfied not by the main memory but by cache-to-cache transfers from a designated on-chip cache responsible for serving requests for the given cache line. **E**xclusive and **R**ecent states allow a cache line that has been read by only one core to be modified without interacting with the directory if no other cache has accessed it since the read, reducing write costs especially for systems where cached data tends to be private to one thread. While these extra states offer performance improvements in specific cases, they combinatorially expand the state space of the whole system, and their benefits must be carefully weighed against the additional costs in implementation and verification efforts.

B. Remote Cache Access

Remote Access (RA) memory architectures eschew capacity-eroding replication entirely by combining the per-core caches into one large virtual non-uniform cache access (NUCA) cache [15]. The address space is divided among the cores in such a way that each address is assigned to a unique *home core* where it can be cached; when a thread wishes to access an address not assigned to the core it is running on, it contacts the home core with a request to perform the memory operation on its behalf [16].

Since accesses to addresses cached at remote cores involve a round-trip on the interconnect, good address-to-core assignment is critical for performance. For maximum data placement flexibility, each core might include a Core Assignment Table (CAT), which stores the home core for each page in the memory space. Akin to a TLB, the per-core CAT serves as a cache for a larger structure stored in main memory. In such a system, the page-to-core assignment might be made when the OS is handling the page fault caused by the first access to the page; the CAT cache at each core is then filled as needed.

For each memory operation under RA, the system must:

1) compute the *home* core H for A (e.g., by consulting the CAT or masking the appropriate bits);
2) if $H = C$ (a *core hit*),
   a) forward the request for A to the cache hierarchy (possibly resulting in a DRAM or next-level cache access);
3) if $H \neq C$ (a *core miss*),
   a) send a remote access request for address A to core H;
   b) when the request arrives at H, forward it to H's cache hierarchy (possibly resulting in a DRAM access);
   c) when the cache access completes, send a response back to C;
   d) once the response arrives at C, continue execution.

Compared to DirCC, RA is easy to implement and simple to verify, since the protocol consists of simple peer-to-peer messages between two core caches rather than a complex dance involving the directory and other core caches. Because only the home core can read or write any address, write atomicity and sequential consistency are trivially ensured. Furthermore, since pure RA restricts any address to be cached in only one location, it can potentially greatly increase the effective capacity of the total on-chip cache and steeply reduce off-chip memory access rates compared to directory-based coherence, ameliorating the off-chip memory bandwidth wall problem.

This comes at a price, however, as accessing data cached on a remote core requires a potentially expensive two-message round-trip: where a directory-based protocol would take advantage of spatial and temporal locality by making a copy of the block containing the data in the local cache, remote-access NUCA must repeat the round-trip *for every remote access* to ensure sequential memory semantics. This makes effective *thread assignment* and *data placement* critical for RA architectures. Optimally, to reduce remote cache access costs, data private to a thread should be assigned to the core the thread is executing on or to a nearby core; threads that share data should be allocated to nearby cores and the shared data assigned to geographically central cores that minimize the average remote access delays. In some cases, efficiency considerations might dictate that critical portions of shared read-only data be replicated in several per-core caches to reduce overall access costs.

Data assignment, migration, and replication techniques have been previously explored in the NUMA context (e.g., [17]), and variants of NUCA and RA are able to move private data to its owner core and replicate read-only shared data among the sharers at the OS level (e.g., [10, 18, 19]) or aided by hardware (e.g., [2022]). While these schemes improve performance on some kinds of data, they still rely on an underlying mechanism (like directory coherence) to provide a shared memory abstraction, and therefore need to be modified to be

applicable to the pure RA case.

C. Execution Migration

Like RA architectures, the Execution Migration Machine (EM^2) architecture [23, 24] maximizes the effective on-chip cache capacity by dividing the address space among per-core caches and allowing each address to be cached only at its unique *home core*. EM^2, however, exploits spatiotemporal locality by bringing *computation* to the locus of the data instead of the other way around: when a thread needs access to an address cached on another core, the hardware efficiently migrates the thread's execution context to the core where the memory is (or is allowed to be) cached and continues execution there. Unlike schemes designed to improve performance of cache-coherence-based designs (e.g., [25,26]) or schemes that require user-level intervention (e.g., [27]), the thread *must* migrate to access memory not assigned to the core it is running on; in EM^2, migration is the *only* mechanism that provides sequential semantics and memory coherence.

If a thread is already executing at the destination core, it must be evicted and migrated to a core where it can continue running. To reduce the necessity for evictions and amortize the latency of migrations, EM^2 cores duplicate the architectural context (register file, etc.) and allow a core to multiplex execution among two (or more) concurrent threads. To prevent deadlock, one context is marked as the *native context* and the other is the *guest context*: a core's native context may only hold the thread that started execution on that core (called the thread's *native core*), and evicted threads must migrate to their native cores to guarantee deadlock freedom [28].

Briefly, when a core C running thread T executes a memory access for address A, it must

1) compute the *home* core H for A (e.g., by masking the appropriate bits);
2) if $H = C$ (a *core hit*),
    a) forward the request for A to the cache hierarchy (possibly resulting in a DRAM access);
3) if $H \neq C$ (a *core miss*),
    a) interrupt the execution of the thread on C (as for a precise exception),
    b) migrate the microarchitectural state to H via the on-chip interconnect:
        i) if H is the native core for T, place it in the native context slot;
        ii) otherwise:
            A) if the guest slot on H contains another thread T', evict T' and migrate it to its native core N'
            B) move T into the guest slot for H;
    c) resume execution of T on H, requesting A from its cache hierarchy (and potentially accessing backing DRAM or the next-level cache).

Although EM^2 requires more invasive hardware changes than RA (since the core itself must be designed to support efficient migration), it shares with it the simplicity of protocol and is easy to reason about: since memory operations for a given address can only be executed on its home core, write atomicity and sequential behavior are trivial, and, since only at most three actors are involved in a migration (source and destination cores, plus potentially an evicted thread's native core), arguments about properties such as end-to-end deadlock freedom can be succinctly and rigorously made [28].

By migrating the executing thread, EM^2 takes advantage of spatiotemporal locality: where RA would have to make repeated round-trips to the same remote core to access its memory, EM^2 makes a one-way trip to the core where the memory resides and continues execution, and, unless every other word accessed resides at a different core, EM^2 will make far fewer network trips. Indeed, since DirCC requires multiple round-trips for highly contested data (between the requesting cache and the directory and between the directory and sharers that must be invalidated), migrations can outperform cache coherence protocol messages on data that is often written and highly shared.

Although migrations are one-way, given a 1-2Kb thread context size, they require a high-bandwidth interconnect to be efficient. Further, core misses pause execution until the thread completes its migration to where the data is. Therefore, just like RA relies on efficient data placement to keep memory references local and reduce the average distance of remote accesses, good data placement is required to reduce migration rates and improve performance in EM^2 architectures.

D. Library Cache Coherence

The Achilles heel of RA and EM^2 lies in their lack of support for replicating temporarily read/write data or permanently read-only data; indeed, data replication with conscious programmer or compiler intervention results in significant performance improvements [29]. At the same time, directory cache coherence incurs multiple round-trip delays when shared data is also written, and relies on protocols that are tricky to implement and expensive to verify.

Library Cache Coherence (LCC) allows automatic replication of shared data without the need for complex invalidation protocols. Instead of a directory, caches wishing to request access to shared data contact a logically central *library*, which allows the relevant cache line to be "checked out" by multiple core caches for reading (i.e., a read-only copy to be stored in a core's private cache) for a fixed period of time; when this time expires, the core caches automatically invalidate the cache line without any invalidate messages from and to the library. Writes are performed in a way similar to remote cache accesses under RA: they are sent to the appropriate library, where they are queued until all the checked out copies have expired, and only then committed to memory. As with distributed directories under DirCC, libraries can be physically distributed among the cores, with each core responsible for a fraction of the address space. In contrast to schemes built on top of traditional coherence protocols (e.g., [30,31]) or implemented at the software level (e.g, [32, 33]), LCC requires neither underlying coherence protocols nor software cooperation to ensure a sequentially-consistent shared memory abstraction.

To efficiently implement timeouts, LCC architectures keep a global clock (or, equivalently, a counter in each core that increments with each global clock cycle); when a copy of a cache line is checked out, the line travels to its client cache with its expiration timestamp, and this timestamp is stored with the cache line to determine whether the line is still valid at any point in time.[1] Although this somewhat increases the number of bits that must be stored in each cache line in the system (an overhead of between 32 and 64 bits for each, say, 64-byte cache line, depending on implementation), the library system scales with the number of cores: unlike DirCC's directories, which must know *which* cores have a valid copy of a given cache line and

[1]For brevity of exposition, we will assume that the timestamps are large enough that they never "roll over," while noting that smaller timestamps can easily be accommodated by, for example, the libraries never giving out timestamps that cross the "roll over" boundary.

in what state, the library stores only a fixed-size timestamp regardless of the number of sharers.

To access address A from core C at global time T the system must, then, perform the following steps:

1) if the request is a LOAD and the timestamp t of the cache line in C containing A has not expired (i.e., $t \leq T$), *or* if the current core is the home library for A ($C = L$), complete the request;
2) otherwise, if the request is a LOAD:
   a) send a request from C to the library L for the cache line containing A; then
   b) determine the new lending timestamp t as follows:
      i) if A is already checked out, the maximum timestamp of checked out copies $t_{\max}$ is stored in the library. If there are write requests for the cache line containing A waiting in the queue, set $t = t_{\max}$;
      ii) else if A is not checked out, determine t using a suitable heuristic and set $t_{\max} = t$;
      iii) else if A has been checked out, determine t using a suitable heuristic and set $t_{\max} = MAX(t,\ t_{\max})$.
   c) send a response from L to C containing the cache line with A and the expiration timestamp t;
   d) if the timestamp has not already expired when the response arrives at C (i.e., if $t \leq T$), store the line containing A in C's cache;
   e) complete the LOAD request and resume execution;
3) otherwise, if the request is a STORE (or an atomic read-modify-write):
   a) send a request from C to the library L with the data to be written to A;
   b) if there are copies currently checked out of the library (i.e., the library timestamp $t_{\max} \geq T$), queue the write request until all the copies expire (i.e., until $t_{\max} < T$);
   c) complete the write request at L, possibly writing to the backing DRAM or next-level cache;
   d) send a write acknowledgement message from L to C;
   e) after receiving the acknowledgement, continue execution at C.

By relying on shared knowledge (the logically global time value), LCC obviates the need for multicast or broadcast invalidations and the resulting multiple network roundtrips that can make DirCC inefficient, and, by using simple timestamps in place of complex cache and directory states, simplifies the implementation and reduces verification costs.

On the other hand, LCC efficiency depends heavily on how the "lending period" for each read request is chosen (in steps 2(b)ii and 2(b)iii): too short of a lending period will result in premature expirations and repeated requests from core caches to the library, while too long of a lending period will delay write requests and unnecessarily suspend execution at the writing cores.

III. Analytical models

To develop intuition about the relative strengths and weaknesses of each scheme, this section formulates analytical models based on *average memory latency* (AML). This model abstracts away multi-actor system interactions to focus on the average cost of a single memory access in terms of experimentally measurable quantities like off-chip memory and on-chip cache access delays, fractions of the various memory requests in programs, or cache miss rates for each request type. By clearly identifying the various sources of latency contributing to each memory scheme, the AML model allows us to explore not only the avenues of optimization available in each protocol, but also to bound the maximum potential of each technique. In the remainder of this section, we describe the AML models for the four memory implementations we have discussed.

Parameter	Value
$cost_{L1\$access}$	2 cycles
$cost_{L1\$insert/inv/flush}$	3 cycles
$cost_{L2\$access}$	7 cycles
$cost_{L2\$insert}$	9 cycles
$cost_{dir\$lookup}$	2 cycles
$size_{ack/address/value}$	32 bits
$size_{cacheline}$	512 bits
$size_{context}$	1088 bits
$cost_{DRAM}$	250 cycles (200 latency + 50 serializ.)
$flit\,size$	256 bits
$cost_{avg\,net\,dist}$	36 cycles (12 hops × 2 cycles/hop + 50% congestion overhead)
$rate_{read}, rate_{write}$	70%, 30%
$rate_{rdI,wrI,rdS}$	40% + 40% + 5%
$rate_{wrS}$	5%
$rate_{rdM}$	10%
$rate_{wrM}$	negligible
$rate_{L1\$miss}$	6%
$rate_{L2\$miss}$	1%
$rate_{core\,miss}$	2%
$cost_{expiration\,wait,LCC}$	3 cycles

TABLE I
DEFAULT PARAMETERS FOR THE ANALYTICAL AML MODELS.

A. Interconnect latency

We adopted a uniform network model in which, on average, each type of core-to-core message travels the same number of hops (12 for an 8×8 mesh) and experiences similar congestion (50%). Packets are divided into equal-size (256-bit) flits, and comprise a header flit with several data flits; wormhole routing is assumed, so each packet experiences a one-cycle serialization delay in addition to the latency of delivering the first flit:

$$cost_{\rightarrow,data\,size} = cost_{avg\,net\,dist} + \left\lceil \frac{data\,size}{flit\,size} \right\rceil,$$

where $data\,size$ depends on the packet size, and grows for cache lines (64 bytes) and the EM^2 context migrations (1088 bits = 32×32-bit registers, a 32-bit instruction pointer register, and a 32-bit status register):

$$\begin{aligned} cost_{\rightarrow,ack} &= cost_{\rightarrow,address} = cost_{\rightarrow,value} \\ &= (12 \times 2 + 50\%) + \left\lceil \frac{32}{256} \right\rceil = 36 + 1 = 37 \text{ cycles}, \\ cost_{\rightarrow,addr\&value} &= (12 \times 2 + 50\%) + \left\lceil \frac{32+32}{256} \right\rceil = 36 + 1 = 37 \text{ cycles}, \\ cost_{\rightarrow,cacheline} &= (12 \times 2 + 50\%) + \left\lceil \frac{512}{256} \right\rceil = 36 + 2 = 38 \text{ cycles, and} \\ cost_{\rightarrow,context} &= (12 \times 2 + 50\%) + \left\lceil \frac{1088}{256} \right\rceil = 36 + 5 = 41 \text{ cycles}. \end{aligned}$$

Only for $cost_{\rightarrow,context}$ we add an additional 3 cycles corresponding to restarting a 5-stage pipeline with a new instruction at the migration destination core, resulting in $cost_{\rightarrow,context} = 44$.

B. Last-level cache and DRAM latencies

The L2 cache was modeled as a shared non-uniform access (NUCA) L2 data cache distributed in slices divided equally among

all the cores, with each slice handling a subset of the memory space. Directories (for DirCC) and libraries (for LCC) are similarly distributed, with each per-core slice handling the same predefined address range as the L2 cache slice on the same core; therefore requests to L2 did not incur additional network costs above those of contacting the directory or library, but only the L2 access itself and the amortized cost of accessing off-chip memory:

$$\begin{aligned} cost_{L2\$\,request} &= cost_{L2\$\,access} \\ &\quad + rate_{L2\$\,miss} \times (cost_{DRAM} + cost_{L2\$\,insert}) \\ &= 7 + 1\% \times (250 + 9) = 9.6 \text{ cycles.} \end{aligned}$$

C. First-level cache miss effects

Under EM^2 and RA, L1 misses at the home core of the address being accessed result directly in L2 requests, and have identical costs. Under LCC, writes are also executed at the home core, and writes that miss L1 incur additionally only the L2 request latency and the L1 insert penalty (using the data from L2 or DRAM):

$$\begin{aligned} cost_{L1\$\,miss,RA} &= cost_{L1\$\,miss,EM^2} = cost_{L1\$\,write\,miss,LCC} \\ &= cost_{L2\$\,request} + cost_{L1\$\,insert} = 9.6 + 3 = 12.6 \text{ cycles.} \end{aligned}$$

LCC reads that miss L1, however, require that a copy of the cache line be brought in from the home core, possibly causing network traffic if the home core is not the current core, and written back to L1:

$$\begin{aligned} cost_{L1\$\,read\,miss,LCC} &= cost_{L2\$\,request} \\ &\quad + rate_{core\,miss} \times \left(cost_{\rightarrow,addr} + cost_{\rightarrow,cacheline}\right) \\ &\quad + cost_{L1\$\,insert} \\ &= 9.6 + 2\% \times (37 + 38) + 3 = 14.1 \text{ cycles.} \end{aligned}$$

We assumed that address ranges are assigned to per-core L2 cache slices (as well as directories and libraries) at 4 KB-page level using a heuristic that keeps data close to its accessors (such as first-touch), leading to a fairly low core miss rate of 2%.

In all L1 miss cases under DirCC, the directory must first be contacted, which may involve network traffic if the relevant directory slice is not attached to the current core. The relevant cache line must be brought to the L1 cache for all types of access, but the protocol actions that must be taken and the associated latency depend on the kind of request (read or write), as well on whether any other L1 caches contain the data. Reads and writes for lines that are not cached in any L1 (i.e., the directory entry is *invalid*) simply query the directory and concurrently access the next level in the cache hierarchy to retrieve the cache line; the same is true for reads of lines cached in *shared* state in some per-core L1, because the directory does not store the actual cache line:

$$\begin{aligned} cost_{rdI,wrI,rdS} &= rate_{core\,miss} \times cost_{\rightarrow,addr} \\ &\quad + \max(cost_{dir\,lookup}, cost_{L2\$\,request}) \\ &\quad + rate_{core\,miss} \times cost_{\rightarrow,cacheline} + cost_{L1\$\,insert} \\ &= 2\% \times 37 + 9.6 + 2\% \times 38 + 3 = 14.1 \text{ cycles.} \end{aligned}$$

Exclusive access (write) requests to lines that are in *shared* state elsewhere additionally contact the sharer(s) to invalidate their copies and wait for their invalidate acknowledgements; assuming that the messages to all sharers are all sent in parallel, we have:

$$\begin{aligned} cost_{wrS} &= rate_{core\,miss} \times cost_{\rightarrow,addr} \\ &\quad + \max(cost_{dir\,lookup}, cost_{L2\$\,request}) \\ &\quad + cost_{\rightarrow,addr} + cost_{L1\$\,inv} + cost_{\rightarrow,ack} \\ &\quad + rate_{core\,miss} \times cost_{\rightarrow,cacheline} + cost_{L1\$\,insert} \\ &= 2\% \times 37 + 9.6 + 37 + 3 + 37 + 2\% \times 38 + 3 = 91.1 \text{ cycles.} \end{aligned}$$

Read requests for data that is cached in *modified* state by another L1 cache must flush the modified line from the cache that holds it, write it back to L2 (to satisfy future read requests from other caches), and only then send the cache line back to the client (note that, in this case, the directory access cannot be amortized with the L2 request because the data to be written must first arrive from the L1 cache):

$$\begin{aligned} cost_{rdM} &= rate_{core\,miss} \times cost_{\rightarrow,addr} + cost_{dir\,lookup} \\ &\quad + cost_{\rightarrow,addr} + cost_{L1\$\,flush} + cost_{\rightarrow,cacheline} + cost_{L2\$\,write} \\ &\quad + rate_{core\,miss} \times cost_{\rightarrow,cacheline} + cost_{L1\$\,insert} \\ &= 2\% \times 37 + 2 + 37 + 3 + 38 + 9 + 2\% \times 38 + 3 \\ &= 93.5 \text{ cycles.} \end{aligned}$$

Writes to data in modified state are similar but can skip the L2 write (because the data is about to be written and the L2 copy would be stale anyway) and directly send the flushed cache line to the client cache:

$$\begin{aligned} cost_{wrM} &= rate_{core\,miss} \times cost_{\rightarrow,addr} + cost_{dir\,lookup} \\ &\quad + cost_{\rightarrow,addr} + cost_{L1\$\,flush} + cost_{\rightarrow,cacheline} \\ &\quad + rate_{core\,miss} \times cost_{\rightarrow,cacheline} + cost_{L1\$\,insert} \\ &= 2\% \times 37 + 2 + 37 + 3 + 38 + 2\% \times 38 + 3 = 84.5 \text{ cycles.} \end{aligned}$$

To estimate the occurrence rates of these state transitions, we simulated several SPLASH-2 benchmarks [34] in the cycle-accurate multicore simulator HORNET [35], and obtained the rates shown in Table I. Given those, the overall L1 miss penalty for DirCC becomes:

$$\begin{aligned} cost_{L1\$\,miss,CC} &= rate_{rdI,wrI,rdS} \times cost_{rdI,wrI,rdS} \\ &\quad + rate_{wrS} \times cost_{wrS} + rate_{rdM,wrM} \times cost_{rdM,wrM} \\ &= 85\% \times 14.1 + 5\% \times 91.1 + 10\% \times 93.5 + 0\% \times 84.5 \\ &= 25.9 \text{ cycles.} \end{aligned}$$

D. Overall average memory latency

Under DirCC, the overall average memory latency (AML) comprises the L1 cache access and the pro-rated cost of the L1 miss:

$$\begin{aligned} AML_{CC} &= cost_{L1\$\,access} + rate_{L1\$\,miss,CC} \times cost_{L1\$\,miss,CC} \\ &= 2 + 6\% \times 25.9 = 3.56 \text{ cycles.} \end{aligned}$$

For EM^2, the AML incorporates the cost of the L1 request (be it a hit or a miss) and the context migration delay in the event of a core miss:

$$\begin{aligned} AML_{EM^2} &= cost_{L1\$\,access} + rate_{L1\$\,miss} \times cost_{L1\$\,miss,EM^2} \\ &\quad + rate_{core\,miss} \times cost_{\rightarrow,context} \\ &= 2 + 6\% \times 12.6 + 2\% \times 44 = 3.63 \text{ cycles.} \end{aligned}$$

Under RA, the core miss overhead involves a round-trip message that depends on the type of access (read or write):

$$\begin{aligned} cost_{core\,miss,RA} &= rate_{read} \times \left(cost_{\rightarrow,addr} + cost_{\rightarrow,value}\right) \\ &\quad + rate_{write} \times \left(cost_{\rightarrow,addr\&value} + cost_{\rightarrow,ack}\right) \\ &= 70\% \times (37 + 37) + 30\% \times (37 + 37) = 74 \text{ cycles,} \end{aligned}$$

giving a total average latency of:

$$\begin{aligned} AML_{RA} &= cost_{L1\$\,access} + rate_{L1\$\,miss} \times cost_{L1\$\,miss,RA} \\ &\quad + rate_{core\,miss} \times cost_{core\,miss,RA} \\ &= 2 + 6\% \times 12.6 + 2\% \times 74 = 4.23 \text{ cycles.} \end{aligned}$$

Finally, LCC reads can be cached locally, and their latency comprises the L1 access costs and the amortized L1 miss:

$$\begin{aligned} cost_{read,LCC} &= cost_{L1\$\,access} + rate_{L1\$\,miss} \times cost_{L1\$\,read\,miss,LCC} \\ &= 2 + 6\% \times 14.1 = 2.84 \text{ cycles.} \end{aligned}$$

Writes, on the other hand, must be performed at the home core, and may incur network transit latencies if the home core is remote. In addition, writes can complete only when all outstanding checked out copies of the relevant cache line have expired, and the L1 cache must be accessed again (since the first access indicated that not all copies have expired); based on our SPLASH-2 simulations we estimated that this expiration penalty will not exceed 3 cycles. The write delay, then, becomes:

$$\begin{aligned} cost_{write,LCC} &= cost_{L1\$access} \\ &+ rate_{L1\$miss} \times cost_{L1\$write miss,LCC} \\ &+ rate_{core miss} \times (cost_{\rightarrow,addr\&value} + cost_{\rightarrow,ack}) \\ &+ cost_{expiration wait,LCC} \\ &= 2 + 6\% \times 12.6 + 2\% \times (37+37) + 3 = 7.23 \text{ cycles.} \end{aligned}$$

Our SPLASH-2 simulations indicated, on the average, a 70%/30% split between reads and writes, giving:

$$\begin{aligned} AML_{LCC} &= rate_{read} \times cost_{read,LCC} + rate_{write} \times cost_{write,LCC} \\ &= 70\% \times 2.84 + 30\% \times 7.23 = 4.16 \text{ cycles.} \end{aligned}$$

In the next section, we vary some of the model parameters from their defaults in Table I, and examine how this affects the AML for each implementation.

IV. Results

We first examined the effect of different L1 cache miss rates on the performance of each scheme. Varying $rate_{L1\$miss}$ has the most striking effect under the DirCC model, which incurs relatively expensive coherence protocol costs (of two or more interconnect round-trips plus directory and often invalidation overheads) for every L1 miss (Figure 1a); while the other three protocols also degrade with increasing miss rates, the differences are not as dramatic.

Of course, L1 cache miss rates under these protocols are not *directly* comparable: on the one hand, DirCC and LCC can make multiple copies of data and are more often able to hit the local cache; on the other hand, EM^2 and RA do not pollute their L1 caches with multiple copies of the same data, and feature higher total L1 cache capacities. Nevertheless, comparisons within each protocol are valid, and we can safely use Figure 1a to examine how the protocols tend to *degrade* when cache miss rates grow.

In all protocols, keeping core miss rates low via efficient data placement is critical to accessing the shared NUCA L2 cache efficiently. In EM^2 and RA, however, a core miss may occur even if the data resides on-chip in some L1 cache (because each address may only be cached at a specific *home core*); the same is true for stores under LCC. Figure 1b varies $rate_{core miss}$ and illustrates that EM^2 and RA perform well as long as the core miss rate is low (below about 2%); when core miss rates are high, however, the high rate of migrations (for EM^2) and remote accesses (for RA) means that the replication-based protocols (DirCC and LCC) dominate.

Again, comparing core miss rates directly can be somewhat misleading, especially for RA and EM^2. This is because under RA threads do not move from their original cores, and each access to a remote core's cache constitutes a core miss; under EM^2, however, the thread migrates to the relevant home core on the first access and subsequent accesses to the same core are no longer core misses but an access to the thread's original core would now become a core miss. The effect, then, depends on the amount of spatiotemporal locality in the access pattern: with high locality, EM^2 core miss rates will be substantially lower than the RA equivalent. To examine the potential, we varied $rate_{core miss}$ for EM^2 *only*, keeping it at the default of 2% for the other protocols: as Figure 1c illustrates, lower core miss rates favor EM^2, while with higher core miss rates EM^2 migration overheads cause its performance to deteriorate.

The execution contexts migrated in EM^2 are substantially larger than both coherence messages and cache-line sizes; this scheme is therefore particularly sensitive to both architectural context size and network bandwidth. To examine the context size variations, we directly varied $size_{context}$ in Figure 1d; to simulate different network bandwidths, we varied the size of a flit that can be transferred in one cycle (*flit size*) in Figure 1e. Both illustrate that EM^2 is sensitive to network bandwidth: for both very large contexts and very low on-chip network bandwidths, EM^2 migration latencies become too large to offer much benefit over the round-trips required by the other protocols.

Finally, since LCC allows "checked out" cache lines to be read without contacting the library until their timestamps expire but must both incur a round-trip to the library and wait until all the checked-out copies of the given cache line expire, we reasoned that it should work very well for workloads highly skewed to load instructions. To evaluate this, we varied the ratio of loads to stores in our model ($rate_{read}$ and $rate_{write}$): as shown in Figure 1f, LCC performance steadily improves relative to the other protocols as the read-to-write ratio grows.

V. Hybrid Approaches

Providing efficient shared memory abstraction in large-scale multicores remains a difficult problem, and there is no clear winner among the approaches we have discussed. RA and LCC have the simplest implementation, but their performance is subpar except for niche applications. DirCC and EM^2 tend to perform better, but EM^2 is sensitive to context sizes and energy limitations, while DirCC protocols are complex cooperating state machines that require careful design and expensive verification.

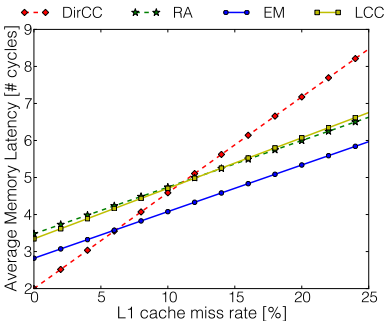
One option is to combine several of those protocols to create a scheme suitable for a wide variety of circumstances. In a sense, the various directory-based cache coherence protocols already behave like this: they comprise a variety of current-state-dependent decisions and the AML description of even the simplest MSI variant is appreciably larger than the other schemes. In this section, we briefly contemplate variations on the simpler protocols and their potential.

A. EM^2 + RA

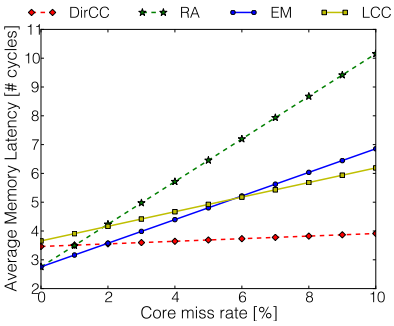
While EM^2 performs well on the per-memory-access level captured by our AML model, it depends on high core hit rates resulting from spatiotemporal locality for good performance: when a thread is migrated to another core, it needs to make several local memory accesses to make the migration "worth it." While some of this can be addressed via intelligent data layout [29] and memory access reordering at the compiler level, occasional "one-of" accesses and the resulting back-and-forth context transfers seem inevitable.

If such an access can be predicted (by, say, the hardware, the compiler, the OS, or even by the programmer), however, optimization is straightforward: adopt a hybrid approach where "one-of" accesses are executed under the RA protocol, and migrations handle sequences of accesses to the same core. Similarly, when accesses are relatively few and the distance to the home core is very short, executing the operations via RA offers latency and energy benefits.

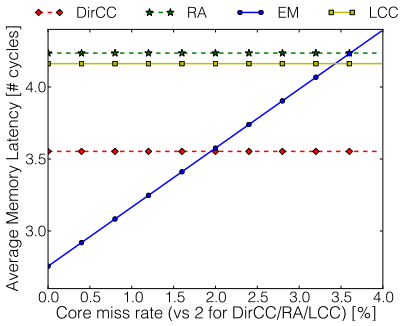
The crux of an efficient hybrid approach, however, is a close cooperation among three actors: the programmer, the compiler, and the operating system. Today's compilers are implicitly optimized for DirCC architectures, and do not pay attention to careful data layout and memory operation order that would reduce EM^2 and RA



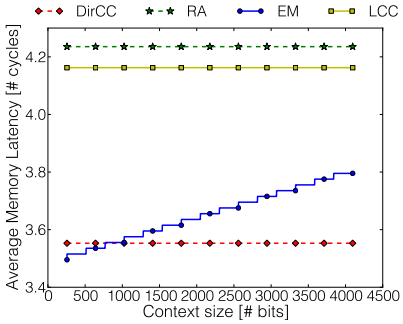
(a) AML versus L1 cache miss rates.



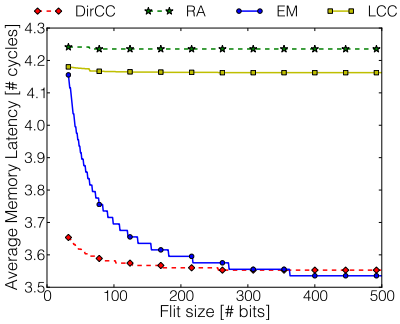
(b) AML versus core miss rates.



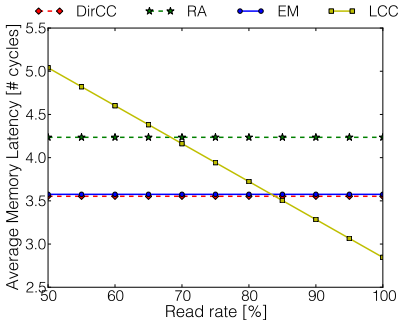
(c) Effect of spatiotemporal locality on AML: EM^2 core miss rates vs. DirCC/RA/LCC.



(d) AML versus EM^2 context size: for very large contexts, the network overhead limits EM^2 performance.



(e) AML versus network bandwidth (flit size).



(f) AML versus the fraction of reads among memory accesses.

core miss rates to a minimum, much less exploiting memory access locality at a level of an EM^2 vs. RA decision. Operating systems and runtime-support libraries like `libc` tend to allocate memory contiguously without regard to where the requesting thread is running and where its local memory is located. Finally, benchmark suites have long been optimized by hand to run fast on existing DirCC architectures, painstakingly allocating data in suitably-sized chunks and avoiding such pitfalls as false write sharing. Applying the same focused research effort to a problem like EM^2/RA could result in an approach which significantly outperforms any known coherence methods.

B. EM^2 + LCC

One of the advantages of EM^2 is that it eschews replication: in many cases, this allows L1-level caches to be used more efficiently and neatly avoids the problem of invalidating copies resident in remote caches when data must be written. But in some cases, such as for frequently accessed read-only data, limited-duration copying actually benefits performance [29].

Since the complexity of DirCC makes it a poor candidate for a hybrid, we consider providing automatic copying using the much simpler LCC mechanism. Like the EM^2/RA hybrid above, this approach depends on a per-access decision on whether a given memory operation should perform a migration or check out a read-only limited-time copy. A successful hybrid would reduce core miss rates and network traffic by obviating migrations needed to read shared data, resulting in significant performance improvements. A specific hybrid scheme may always migrate for writes.

C. LCC with read/write checkouts

As we have seen, LCC works very well for workloads with high load-to-store ratios, and is limited in performance by its centralized writes. To address this, we might contemplate bringing LCC closer to DirCC: that is, allowing cache lines to be "checked out" not only for reads but also for writes. Although the read/write copies would have to be checked out to one client cache at a time, contiguous read-write accesses to the same location by the same thread would entirely avoid the round-trip to the library and waiting for read-only timestamps to expire.

Like the other hybrids, this approach relies on finer-granularity per-access decisions: while a read-only copy can be checked out by multiple caches before any of the timestamps expire, read-write copies can only be checked out to one client and would need more carefully tuned (and probably shorter) timestamps.

VI. Conclusion

In the preceding sections, we have outlined four methods of providing shared memory abstraction in massive-scale multicores: two traditional approaches (directory-based cache coherence and a remote-access architecture) and two novel designs (shared memory via execution migration and library cache coherence). To compare the relative strengths of the protocols, we have developed an analytical model for each based on per-access average memory latency, and examined the design and application parameter ranges that favor each protocol; Table II summarizes the pros and cons of each approach.

Both novel schemes we described introduce opportunities for further optimization. Operating systems can ensure efficient data placement among core caches to minimize core miss rates; compilers can help by laying out data structures according to anticipated sharing patterns and reordering memory accesses to maximize spatiotemporal locality. This is especially true of the hybrid approaches we sketched in Section V: these rely on a per-memory-access decision as to which protocol to use (e.g., in EM^2/RA, whether to migrate or perform a memory access), and provide ample topics for further research.

Protocol	Pros	Cons
DirCC	little additional OS or compiler support (e.g., data placement) required; many programs already optimized for DirCC	complex to implement and verify; requires large private caches to overcome negative effects of high cache miss rates
RA	simple to implement and verify; efficient when data placements keep most accesses on the local core	requires multiple round-trip access to non-local cores even if data exhibits high spatiotemporal locality; does not even replicate read-only data
EM^2	simple to analyze and verify; efficient when good data placement and high spatiotemporal data access locality can amortize the cost of thread migrations	thread migrations must transfer large execution contexts across the network, requiring high-bandwidth interconnects when poor data placement results in high core miss rates; does not even replicate read-only data
LCC	leverages temporary data replication to complete most loads locally without the performance and complexity overheads of a cache coherence protocol	writes performed similarly to RA without benefit of private-cache locality; efficiency depends on effective heuristics to determine optimal timestamp values

TABLE II
THE PROS AND CONS OF EACH OF THE DESCRIBED SHARED MEMORY IMPLEMENTATIONS.

VII. ACKNOWLEDGEMENTS

The authors would like to express gratitude to Dr. Omer Khan for helpful discussions as well as valuable suggestions and feedback on early versions of this manuscript. We also thank Rachael Harding for valuable feedback.

REFERENCES

[1] I. T. R. for Semiconductors, "Assembly and Packaging," 2007.
[2] S. Rusu, S. Tam, H. Muljono, D. Ayers, J. Chang, R. Varada, M. Ratta, and S. Vora, "A 45nm 8-core enterprise Xeon® processor," in *A-SSCC*, 2009.
[3] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal, "Baring it all to Software: Raw Machines," in *IEEE Computer*, September 1997, pp. 86–93.
[4] K. Sankaralingam, R. Nagarajan, H. Liu, J. Huh, C. K. Kim, D. B. an d S. W. Keckler, and C. R. Moore, "Exploiting ILP, TLP, and DLP using polymorphism in the TRIPS architecture," in *ISCA*, 2003, pp. 422–433.
[5] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. Brown III, and A. Agarwal, "On-Chip Interconnection Architecture of the Tile Processor," *IEEE Micro*, vol. 27, pp. 15–31, September 2007.
[6] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook, "TILE64 - Processor: A 64-Core SoC with Mesh Interconnect," in *ISSCC*, 2008.
[7] S. R. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar, "An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS," *IEEE J. Solid-State Circuits*, vol. 43, pp. 29–41, 2008.
[8] S. Borkar, "Thousand core chips: a technology perspective," in *DAC*, 2007.
[9] A. C. Sodan, "Message-Passing and Shared-Data Programming Models—Wish vs. Reality," in *HPCS*, 2005.
[10] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: near-optimal block placement and replication in distributed caches," in *ISCA*, 2009.
[11] J. R. Goodman, "Using cache memory to reduce processor-memory traffic," in *ISCA*, 1983.
[12] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz, "An Evaluation of Directory Schemes for Cache Coherence," in *ISCA*, 1988.
[13] A. Gupta, W. Weber, and T. Mowry, "Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes," in *ICPP*, 1990.
[14] D. Abts, S. Scott, and D. J. Lilja, "So Many States, So Little Time: Verifying Memory Coherence in the Cray X1," in *IPDPS*, 2003.
[15] C. Kim, D. Burger, and S. W. Keckler, "An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches," in *ASPLOS*, 2002.
[16] C. Fensch and M. Cintra, "An OS-based alternative to full hardware coherence on tiled CMPs," in *HPCA*, 2008.
[17] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum, "Operating system support for improving data locality on CC-NUMA compute servers," *SIGPLAN Not.*, vol. 31, pp. 279–289, 1996.
[18] S. Cho and L. Jin, "Managing Distributed, Shared L2 Caches through OS-Level Page Allocation," in *MICRO*, 2006.
[19] M. Awasthi, K. Sudan, R. Balasubramonian, and J. Carter, "Dynamic hardware-assisted software-controlled page placement to manage capacity allocation and sharing within large caches," in *HPCA*, 2009.
[20] M. Zhang and K. Asanović, "Victim replication: maximizing capacity while hiding wire delay in tiled chip multiprocessors," in *ISCA*, 2005.
[21] M. Chaudhuri, "PageNUCA: Selected policies for page-grain locality management in large shared chip-multiprocessor caches," in *HPCA*, 2009.
[22] K. Sudan, N. Chatterjee, D. Nellans, M. Awasthi, R. Balasubramonian, and A. Davis, "Micro-pages: increasing DRAM efficiency with locality-aware data placement," *SIGARCH Comput. Archit. News*, vol. 38, pp. 219–230, 2010.
[23] O. Khan, M. Lis, and S. Devadas, "EM^2: A Scalable Shared-Memory Multicore Architecture," *MIT-CSAIL-TR-2010-030*, 2010.
[24] M. Lis, K. S. Shim, M. H. Cho, C. W. Fletcher, M. Kinsy, I. Lebedev, O. Khan, and S. Devadas, "Brief Announcement: Distributed Shared Memory based on Computation Migration," in *SPAA*, 2011.
[25] P. Michaud, "Exploiting the cache capacity of a single-chip multi-core processor with execution migration," in *HPCA*, 2004.
[26] M. Kandemir, F. Li, M. Irwin, and S. W. Son, "A novel migration-based NUCA design for Chip Multiprocessors," in *SC*, 2008.
[27] M. D. Noakes, D. A. Wallach, and W. J. Dally, "The J-Machine Multicomputer: An Architectural Evaluation," in *ISCA*, 1993.
[28] M. H. Cho, K. S. Shim, M. Lis, O. Khan, and S. Devadas, "Deadlock-Free Fine-Grained Thread Migration," in *NOCS*, 2011.
[29] K. S. Shim, M. Lis, M. H. Cho, O. Khan, and S. Devadas, "System-level Optimizations for Memory Access in the Execution Migration Machine (EM^2)," in *CAOS*, 2011.
[30] A. R. Lebeck and D. A. Wood, "Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors," in *ISCA*, 1995.
[31] A.-C. Lai and B. Falsafi, "Selective, accurate, and timely self-invalidation using last-touch prediction," in *ISCA*, 2000.
[32] S. L. Min and J. L. Baer, "Design and Analysis of a Scalable Cache Coherence Scheme Based on Clocks and Timestamps," *IEEE Trans. Parallel Distrib. Syst.*, vol. 3, pp. 25–44, 1992.
[33] X. Yuan, R. Melhelm, and R. Gupta, "A Timestamp-based Selective Invalidation Scheme for Multiprocessor Cache Coherence," in *ICPP*, 1996.
[34] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, "The SPLASH-2 programs: characterization and methodological considerations," in *ISCA*, 1995.
[35] M. Lis, P. Ren, M. H. Cho, K. S. Shim, C. W. Fletcher, O. Khan, and S. Devadas, "Scalable, accurate multicore simulation in the 1000-core era," in *ISPASS*, 2011.