DDCache: Decoupled and Delegable Cache Data and Metadata

Hemayet Hossain, Sandhya Dwarkadas, and Michael C. Huang
University of Rochester
Rochester, NY 14627, USA
{hossain@cs, sandhya@cs, huang@ece}.rochester.edu

***Abstract*—In order to harness the full compute power of many-core processors, future designs must focus on effective utilization of on-chip cache and bandwidth resources. In this paper, we address the dual goals of (1) reducing on-chip communication overheads and (2) improving on-chip cache space utilization resulting in larger effective cache capacity and thereby potentially reduced off-chip traffic. We present a new cache coherence protocol that decouples the logical binding between data and metadata in a cache set. This decoupling allows data and metadata for a cache line to be independently delegated to any location on chip. By delegating metadata to the current owner/modifier of a cache line, communication overhead for metadata maintenance is avoided and communication can be effectively localized between interacting processes. By decoupling metadata from data, data space in the cache can be more efficiently utilized by avoiding unnecessary data replication. Using full system simulation, we demonstrate that our decoupled protocol achieves an average (geometric mean) speedup of 1.24 (1.3 with microbenchmarks) compared to a base statically mapped directory-based non-uniform cache access protocol, while generating only 65% and 74% of the on-chip and off-chip traffic respectively, and consuming 74% of the corresponding energy (95% of the power) in the on-chip memory and interconnect compared to the base system.**

***Keywords*-Cache Coherence; Chip Multiprocessors; Decoupled Cache; Delegable Cache; DDCache;**

I. Introduction

If the full compute power of current multi-core processors and future many-core processors [18], [21], [28], [29], [36] is to be unleashed, conventional on-chip cache designs must be revised to support both partitioned use and fine-grained sharing. In order to scale to the many on-chip cores, directory-based protocols seem the most viable option [2], [23], [24]. In a conventional directory-based protocol, cache lines are distributed among the nodes in a straightforward interleaved fashion. Such an interleaving allows easy determination of the home node, but also creates overheads in coherence activities. First, communicating processor cores may be close to each other physically and yet have to route their invalidation and fetch requests indirectly via the arbitrarily designated home node. Given the presence of dynamically changing fine-grain access patterns, it is unlikely that any simple mapping schemes can ensure that most cache lines are mapped close to or at the accessing nodes. Second, the home node retains a copy of data even when the copy is stale. These overheads are a natural byproduct of the data structure in directory, where the metadata keeping track of cache line state and sharers for coherence is bound with the data storage and pinned to the home node.

In this paper, we propose an alternative implementation of the directory-based protocol, *Decoupled and Delegable Data and Metadata*, with the dual goals of reducing on-chip communication overheads and improving on-chip cache space utilization. This design allows the metadata to be transferred to one of the communicating cores so that coherence can be delegated to the core. This way, coherence is still maintained by the same logic, but in a directory assist closer to the communication, not artificially bound to the home node. Such an arrangement can further improve the efficiency of designs that already move the indirection through the directory off the critical path [16]. In addition to decoupling coherence maintenance from the home node, the design also relaxes the one-to-one binding between data and metadata, allowing the home node to avoid wasting storage keeping stale copies of delegated lines. We expect the following direct benefit from this mechanism.

- **Faster Cache Coherence:** In a canonical directory-based protocol, consulting the directory of a line on the home node is on the critical path for a number of situations such as transferring a line from the last writer to the next reader. Proposals like Owner Prediction [1] and ARMCO [16] take a first step to addressing this indirection problem by predicting (close-by) owners/sharers and sending coherence messages directly to the targets. Nevertheless, the designs still require a substantial involvement of directory, and even when some of the directory accesses are off the critical path, they can indirectly appear on the critical path for future accesses [16].

  To further expedite communication, we allow the ownership of the metadata (and the responsibility to maintain coherence) to migrate to accessing nodes – typically the predominant writer nodes. This allows the owner node to perform 2-hop invalidations instead of using 3- or 4-hop requests.
- **Reduced Coherence Traffic:** When the home node delegates the coherence responsibility and sends the metadata to another node, the home node still keeps a pointer to the current keeper of coherence in order to redirect any new requester to the current coherence keeper. The home node does not maintain an up-to-date sharer list. As a result, it is "short-circuited" from the coherence loop. In the steady state, all indirections via the home node in a conventional design are avoided, including the often unnecessary writebacks to the home node (in a protocol without the owner state) when a producer repeatedly performs a cache-to-cache transfer to update the consumer with the newer version of the cache line.
- **Increased Cache Capacity:** When the home node delegates coherence responsibility, it also gives up the responsibility to provide data for a request. This means that the data

storage allocated for the cache line is no longer needed. By decoupling data and metadata, we can free the data block and use it for other data. The increase in effectiveness of the storage reduces capacity misses and thus the demand on off-chip access bandwidth. In the extreme case, no line in the L1 caches has a replica in the L2 cache, and the system essentially mimics a non-inclusive cache structure. However, since the home node still maintains a degenerate copy of the metadata (pointing to the current coherence keeper of the line), the L2 cache can still filter out unrelated (chip-to-chip) coherence checks as an inclusive cache system can. Of course, this is achieved with extra storage just for the degenerate metadata.

- **Timely Update for Producer Consumer Data:** As described above, the metadata owner has the complete list of sharers of the cache line. In the case of a producer-consumer data access pattern, the producer, being the owner of the metadata, can perform data pushing in order to disseminate the data in a more timely fashion to potential consumers.

Overall, our protocol achieves faster coherence while reducing traffic on both on-chip and off-chip interconnects. Both effects lead to reduced energy consumption. Our simulation results show that on a 16-core CMP with 64KB L1 split-caches and a 4MB L2 cache (in 16 different banks, 256KB each), speedup over a baseline directory-based design ranges from 1.01 to 2.11 for a suite of 12 commercial [34], [37], scientific [39], mining [7], and branch-and-bound benchmarks, and two microbenchmarks. The geometric mean of the speedup is 1.24 excluding the microbenchmarks and 1.30 including the microbenchmarks. Our decoupled protocol also generates only 65% of the on-chip interconnection network traffic (flits) of the base protocol on average, 74% of the off-chip traffic, and 74% of the dynamic energy consumption in the on-chip memory hierarchy (95% of the dynamic power consumption). We also compare our decoupled protocol to ARMCO [16] and Victim Migration (VM) [40]. The decoupled protocol outperforms both by a significant margin (speedup of 1.22 (1.15 without microbenchmarks) when compared to VM and 1.13 (1.13 without microbenchmarks) when compared to ARMCO).

II. Design Overview

Figure 1 presents a high-level view of a generic, modular CMP architecture we base our design on. Each tile in the CMP contains a processor core, private L1 caches, and a slice of the globally-shared L2 cache statically-mapped (line-interleaved) by physical address. To optimize for fine-grain sharing, this baseline CMP is augmented with a set of architectural support (similar to those used in ARMCO [16]) to identify access patterns and predict coherence targets.

Additionally, the metadata portion of the L1 data cache is augmented with a sharers list in order to assume the delegated role of coherence keeper. When an L1 cache miss occurs, the coherence logic works as usual by sending a request (*e.g.*, read or upgrade) over the interconnect to the coherence keeper. The only difference is that the keeper is not necessarily the home node.

At the L2 cache banks, extra ways of metadata are provided. If a line is delegated to some other node, the L2 cache line in the home node gives up its data space. When the line is undelegated from the current keeper, new data space will be requested.

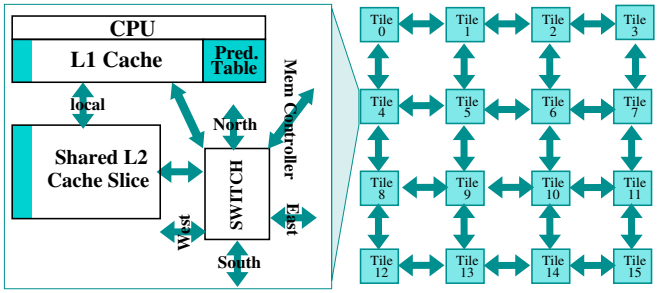

Fig. 1. Schematic of the underlying CMP architecture depicting a processor with 16 cores with additions for DDCache and ARMCO identified via shading.

III. Protocol and Hardware Design

A. Decoupled Data and Metadata Structures

1) Support for delegation: To support delegation, the L1 cache is augmented with a sharers list ($Sharers$ in L1 data of Figure 2). Note that only the sharers list on the coherence-keeper node (henceforth referred to as the *keeper* for short) tracks the sharers of the line. In all other sharers' L1 caches, the sharers list merely points to the keeper, if there is one. If the line is not delegated, the list remains empty. The sharers list at the home slice of the L2 cache is used conventionally when the line is not delegated. When it is delegated, the home L2's $Sharers$ points to the keeper. For example, if line A at an L2 bank is delegated to P0 and that line is also shared by P1 and P2, then for line A, $Sharers$ at P0's L1D will be $\{1, 2\}$ $Sharers$ at P1's and P2's L1D and at the home L2 bank will all be $\{0\}$. P0 does not store its id in its own sharers list since it is implicitly a sharer. For undelegated lines, each core can thus store its own id in the sharers bit vector to identify that it needs to communicate with the corresponding L2 bank (rather than with some other L1).

We currently represent the sharers list using a presence bit vector, resulting in the directory overhead being linear in the number of cores/tiles. Previously proposed techniques [13] to reduce directory overhead and make it more scalable are equally applicable to this design.

2) Decoupling data and metadata: At the L2 level, the storage is organized to handle data and metadata in a decoupled manner. Specifically, within a set, there are more metadata ways than data ways. When coherence responsibility is delegated to an L1 cache, the home node's L2 only needs to keep the metadata portion to forward requests to the keeper. Thus, the freed-up data storage can be used for another cache line.

Data and metadata association can be implemented using pointers and free lists. We chose instead to avoid the complexity of the corresponding management and create a static pairing between metadata and data blocks for each data block in the set.

Placement and replacement are performed using a tree-based pseudo-LRU algorithm that uses $associativity - 1$ bits per set to track recently accessed lines in a binary partitioned way. We use two trees: one for data with paired metadata (N-way in Figure 2) and the other for only metadata (t-ways in Figure 2).

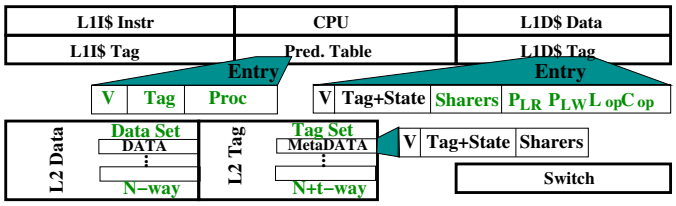

Fig. 2. Block diagram showing the details inside a core. Structures with changes and extra structures are represented using a lighter color.

When a line is fetched in normal undelegated state, we select a replacement candidate from the first tree. On the other hand, housing a new delegated line requires only metadata and can select a victim from either tree. In order to avoid unfairly favoring eviction of delegated lines, we choose either the first or second tree in a ratio proportional to the number of entries in each tree, using the last few bits of the clock timer to randomize the selection. Finally, when a line is being undelegated and returns home, its metadata may not have a paired data way and thus some shuffling is needed: A replacement candidate is selected from the first tree; If the selected line is in delegated state (and thus does not need its data storage now), swapping the two lines is enough; Otherwise, the selected line is evicted.

3) Destination predictor: When we let the keeper maintain coherence, one issue that arises is that the location of the keeper is not fixed as the home is. If each requester has to go through the home node to locate the keeper, there will be little if any benefit from delegation. We use an address-based destination predictor based on [16] to predict the location of the owner of the data or a closest sharer that can also provide the data. This way, for most cases, we avoid indirection via the home.

The predictor table is similar to the one used in [16] and is a cache-like structure that keeps a valid bit, a tag, and a processor ID for the predicted destination node (Figure 2). The table is updated when an invalidation is received: the tag of the cache line and the ID of the invalidating processor are recorded in the predictor table. This captures the fixed producer-consumer relationship very well. Additionally, when a request is serviced by the keeper or home, the reply piggybacks information about the closest node that also has a copy of the data. This information is also updated in the predictor table, enabling prediction of the closest sharer in case of invalidation. Each L1D cache line tracks the last reader and writer of the cache line. During invalidation or eviction of the line, it also lets the last reader and writer (if not this L1) know that the line is being evicted so that they may update their predictor tables.

On L1 cache misses, the predictor table is accessed like a cache. If there is a hit, the request is sent to the predicted potential owner/sharer; otherwise it is sent to the home node. The sensitivity analysis conducted in [16] on predictor table size (256, 512, 1K, and 2K entries) and associativity (8 and 16 way) pointed to the use of an 8-way associative 1K-entry predictor table as a good compromise between performance and complexity. The accuracy of the prediction ranges from 63% to 93% (74% on average) for our benchmark suites, similar to earlier results [16].

B. Protocol Actions

Our general guideline is to supply data from a location close to the requester. When an L1 cache miss occurs, the predictor is consulted to find a nearby node who can potentially supply the data. This nearby node can be the keeper or just a sharer. As a result of this distributed data supply system, the coherence ordering point in our design is also distributed across the nodes, depending on the supplier of the data:

- **At the home L2 bank:** When there is a cache miss in both the L1 and the predictor table, the request is sent to the home L2. If the line is not delegated, the request is serviced just as in the conventional design. The only difference is that if the request is for exclusive access, delegation is performed: after sending invalidations to the current sharers and collecting invalidation acknowledgments, the line enters delegated (D) state at the L2.

  A delegated line will be surrendered back to the home L2 due to eviction either at the owner L1 or at the home L2. In the latter case, the owner L1 will receive a request from the home to surrender and invalidate the line.
- **At the keeper:** When a cache line is delegated, the keeper will be the coherence ordering point for many requests. A request may arrive at the keeper in one of three ways. First, the request may arrive directly from the requester if the requester correctly predicts the location of the keeper or has the line in shared state (and thus has metadata that tracks the keeper).

  Second, the request may be forwarded from the home: When the predictor table misses, or when the prediction is incorrect, the request is sent either directly to the home node or is indirectly forwarded to the home by the predicted target node that does not have the line in its L1 cache.

  Finally, a read-exclusive request can be forwarded to the keeper by a sharer of the line: When the request is initially sent to a sharer, the sharer can directly forward it to the keeper if the line is in delegated mode. Recall that in delegated mode, the line can be shared among multiple L1 caches. The metadata of the keeper L1 maintains the sharers list, while all other nodes' sharers lists point to the keeper. Note that if a read request is sent to a sharer, like in the ARMCO design, the data is supplied by the sharer and coherence information is updated in the background (as discussed later).

  All exclusive requests (upgrade or read-exclusive) entail a transfer of coherence responsibility unless it is classified as a write-in-place operation to reduce the ping-pong effect. The requester gets two separate replies. The first one is a data or access grant reply for the request. After getting this reply, the line goes into a transient state that allows the requester itself to read and write but does not allow the node to supply the line to another node (unless the consistency model does not require write atomicity). While the change of ownership is under way, requests coming to the old keeper will be forwarded to the new keeper. After supplying the data or upgrade grant, the old keeper sends invalidations to all sharers (except the new keeper) and, in parallel, sends a notice of ownership change to

the home. After getting the notice, the home node updates the metadata to point to the new keeper. After collecting invalidation acknowledgments and the acknowledgment from the home, the old keeper confirms ownership transfer to the new keeper. At this time, the line transitions into stable state in the new keeper's cache and the new keeper can service other nodes' requests.

- **At a sharer L1:** To expedite communication, we allow an L1 cache to supply data without ownership of the cache line as in ARMCO [16]. Specifically, if a read miss request is sent to a node identified by the predictor as a possible sharer and the node is indeed a sharer, it will provide a reply with data and metadata info – the identity of the keeper. Meanwhile, a notice to update the sharers list is sent to the keeper (if the sharers list points to a keeper) or home (if the sharers list is empty, i.e., holds its own id indicating that the line is not delegated). Until the keeper acknowledges this notice, the supplier node is temporarily responsible for the coherence of the requester node and becomes a coherence ordering point. The keeper also sends an acknowledgment to the requester, indicating that the transfer is complete. One possible race condition is if the keeper initiates an invalidation for the line prior to receiving the sharers list update. In this case, the keeper acknowledges the data transfer with an indication that an invalidation is pending. The supplier delays any received invalidation until the ongoing data transfer is acknowledged by the keeper. At this point, the supplier applies the received invalidation. (If the invalidation message arrives out of order with respect to the acknowledgment, the supplier will wait for the invalidation and NACKs any further read request in the mean time.) Note that a sharer cannot service any request other than a read. Exclusive requests are simply forwarded to the keeper or to the home node depending on whether the line is delegated.

C. Delegation Decision

Broadly, we can classify data accesses into private, shared read-only, and shared read-write. The general guideline is to pick the delegation decision that best suits the access pattern. Of course, at run-time, given a particular access request, we can only approximately determine the access pattern for the data.

- **Private data:** Private data only concerns the accessing thread and should always be delegated to the accessing node. This allows faster upgrade requests and reduction in L2 capacity usage. The predictor and the directory information combined can help us identify private data. Specifically, when an access misses in the cache, we use the line address to check the destination predictor table. Since the predictor tries to capture the location of other shared copies in the system, a miss in the table suggests that the line could be a private line. We then send the request with a hint bit indicating the line is probably private. When the request eventually arrives at either the home or the keeper and the metadata shows that it is indeed not shared by other nodes, then the line is treated as private and is thus delegated. If there is a miss at the L2, the line will be brought in from memory and delegated – the home will only keep the metadata.
- **Shared data:** When data is shared, the appropriateness of delegation becomes more complex. It depends on the accuracy of the prediction table, which in turn depends on whether the read-write patterns are stable. In general, when a line is delegated, correct prediction speeds up transactions, while misdirections add to the delay. For read-write data, the home node is likely to have stale data and has to forward the read request. Therefore, delegating the line makes more sense as it avoids unnecessary storage use in the L2 and repeated traffic to update the home version. For read-only data, maintaining a version at the home node does not incur extra traffic and allows the home to provide data without forwarding to the keeper. As such, it favors not delegating the line. For implementation simplicity, we delegate for read-write shared data and do not delegate read-only shared data. Thus, if a line is first believed to be private and thus delegated, and subsequently read by another node (thus suggesting shared read-only), we undelegate from the first requester node.

D. Sharing Pattern Optimization

When data is shared, unnecessary communication and indirection via the directory can be avoided by tailoring the coherence protocol to the access pattern. We use ARMCO [16] as our base design, which uses $2log_2\,N + 2$ bits (N being the number of tiles in the system) in each cache line to track the access patterns and perform in-place accesses when warranted. The bits are P_{LR} ($log_2\,N$ bits to identify the last reader), P_{LW} ($log_2\,N$ bits to identify the last writer), C_{op} (1 bit flag to track whether multiple accesses happened from the local tile), and L_{op} (1 bit flag to determine whether the last access in the cache line was read or write), as shown in Figure 2. Hossain et al. [16] describe how these bits are used to identify and optimize the communication for migratory, producer-consumer, and false-shared access patterns. In addition to optimizing the data communication as in this protocol, we also delegate the metadata to the current modifier as described in Section III-C, thereby avoiding metadata updates to the directory, even though such an update would be off the critical path.

E. Forwarding for Producer-Consumer Data

One final benefit of delegating the coherence responsibility is that in producer-consumer access patterns, the producer has knowledge of the previous sharers and thus can proactively push data to them. This consumer list is simply the sharers list before the producer sent the invalidations. Rather than clear the list when performing an invalidation, we keep it. Previous sharers are distinguished from current sharers by the state of the cache line.

One policy issue with forwarding is when to send the update. We use a simple, albeit reactive approach which is to update all consumers when one of them generates an on-demand read. To cleanse the sharers list of nodes no longer consuming the data, we leverage the tracking bit C_{op} (used in [16]) included in the L1D cache lines, which is set when multiple accesses occur to the line from the local processor. If C_{op} is 0 when the next invalidation is received, it means the line has not been

accessed since the last update (at most one access, in which case read-in-place is better than copying the whole cache line). The acknowledgment message piggybacks this information and the producer will reset the bit corresponding to this consumer, preventing future updates to that node. Finally, when the line is in dirty state and the producer receives a read request from outside the current sharers list, we treat it as a signal of a changing producer-consumer pattern and flash-clear the sharers list.

IV. Performance Evaluation

A. Evaluation Framework

For evaluation of our new DDCache, we used a Simics-based [25] full-system execution-driven simulator that models the SPARC architecture faithfully. We used Ruby from the GEMS toolset [27], modified to encode the decoupled data and metadata protocol, for cache memory simulation. We simulated a 16-core chip multiprocessor (CMP) using a 4x4 mesh interconnect to connect the tiles. Each tile contains private L1 instruction and data caches and a slice of the 16-way shared L2 as in the base system. The L1 cache and local L2 cache can communicate directly without going through the switch (Figure 1). The main parameters are summarized in Table I.

For the baseline coherence protocol, we used a statically-mapped line-interleaved non-uniform-shared L2 (L2S) MOSI-style directory-based protocol. We encode all stable and transient states and all required messages for a detailed network model simulation of DDCache, ARMCO [16], Victim Migration [40], and L2S using SLICC [27].

For interconnect modeling, we use GEMS's [27] network model. We employ virtual cut-through switching. The network link width is 16 Bytes and so is the flit size. For L2S, we have 8B and 72B message sizes. For DDCache, we have 8B, 16B, and 72B messages. We use a configuration where the network link is shared at an 8B granularity, *i.e.*, two 8B messages (or one 8B message and part of a 16B or 72B message) can be transmitted simultaneously, assuming both messages are ready for transmission. Messages sent between L1s and L2s are treated as on-chip flit-hops and messages communicated between L2 and memory controller are treated as off-chip flit-hops.

For power consumption modeling, we use Cacti 6.0 [31] to model power, delay, area, and cycle time for the individual cache banks as well as the interconnect switches. All process-specific values used by Cacti 6.0 are derived from the ITRS roadmap. We use a 45 nm process technology and focus on dynamic energy.

For our evaluation, we use a wide range of benchmarks, which include commercial, scientific, mining, branch and bound, and microbenchmarks. In order to demonstrate efficiencies for specific access patterns, we have developed microbenchmarks with producer-consumer and migratory access patterns. As commercial workloads, we use the Apache webserver with the surge [5] request generator and SPECjbb2005. Alameldeen et al. [3] described these commercial workloads for simulation. As scientific benchmarks, we have a large set of applications and kernels from the SPLASH2/SPLASH suites [39], which includes *Barnes*, *Cholesky*, *FFT*, *LU*, *MP3D*, *Ocean*, *Radix*, and *Water*. Our benchmark suite also includes a graph mining application [7] and a branch-and-bound based implementation of the traveling salesman problem (TSP). All these applications are thread-based except Apache, which is process-based. Table II lists the problem sizes, access patterns, and L1 miss rates for L2S at 16 processors.

TABLE I
System parameters

16-way CMP, Private L1, Shared L2	
Processor cores	16 3GHz in-order, single issue, non-memory IPC=1, sequentially consistent
L1 (I and D) cache	64KB 2-way each, 64-byte blocks, 2-cycle
Predictor table	1K entry 8-way associative
L2 cache	4MB in L2S, 20-way tag and 15-way data for DD-Cache (16-way tag and data for others), 16 banks, 64-byte blocks, Sequential tag/data access, 14-cycle
Memory	4GB, 300-cycle latency
Interconnect	4x4 mesh, 4-cycle link latency, 128-bit link width, virtual cut-through routing

B. Effect of Delegation on Speed of Coherence

The first benefit of the DDCache protocol is a reduction in the coherence latency. Figure 3 shows the effect in terms of speedup with respect to L2S. DDCache builds on top of ARMCO [16], which is also shown for comparison. In this comparison, we do not take advantage of the capacity benefit of delegation, so all configurations use 16 way tags and data (T16D16).

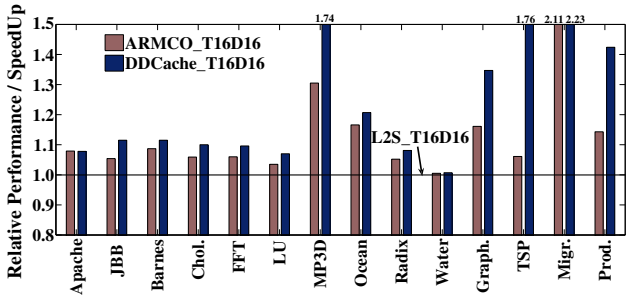

Fig. 3. Effect of metadata delegation to some L1 on execution speedup normalized to L2S. Note that the Y axis begins at 0.8 for all speedup charts in order to more clearly see the differences among protocols.

From Figure 3, we see that for some workloads (*e.g.*, *MP3D*, *TSP*, and *Producer-Consumer*), the benefit of delegation is significant. Delegation benefits are due to the faster coherence communications, especially in stable producer-consumer sharing patterns: the producer can invalidate consumers directly without going through the home node and similarly consumers can get the data directly from the producer.

For certain applications, such as *Apache*, the sharing is mostly on read-only data. In that case, cache lines are not delegated and DDCache provides no benefit. In fact, some read-shared data would be identified as private initially and would have to be undelegated in a subsequent read from another node. That slows down the second read access. This accounts for lack of performance gain from DDCache compared to ARMCO.

Overall, we see significant performance improvement: the speedup of enabling delegation over ARMCO is 1.11 (geometric mean).

We also measured the frequency of keeper transfers while a cache line is delegated. We count the number of times the

TABLE II
PROBLEM SIZE, DATA ACCESS PATTERNS, AND MISS RATES FOR THE BENCHMARKS EVALUATED ON A 16-CORE CMP WITH THE BASE PROTOCOL.

Benchmark	Simulated problem size	Major data access pattern	L1 miss rate (L2S)
Apache	80000 requests fastforward, 2000 warmup, and 3000 for data collection	mostly read-shared,read-write-shared	11.2%
JBB2005	350K Tx fastforward, 3000 warmup, and 3000 for data collection	read-shared and producer-consumer	7.3%
Barnes	8K particles; run-to-completion	single producer-consumer and read-shared	1.9%
Cholesky	lshp.0; run-to-completion	migratory, read-shared, read-write-shared	1.5%
FFT	64K points; run-to-completion	read-shared	3.7%
LU	512x512 matrix,16x16 block; run-to-completion	producer-consumer, false-sharing	2.0%
MP3D	40K molecules; 15 parallel steps; warmup 3 steps	migratory, read-shared, read-write-shared	16.6%
Ocean	258x258 ocean	producer-consumer	6.9%
Radix	550K 20-bit integers, radix 1024	read-shared, producer-consumer	3.2%
Water	512 molecules; run-to-completion	little read-shared	1.3%
GraphMine	340 chemical compounds, 24 different atoms, 66 atom types, and 4 types of bonds; 200M instructions; warmup 300 nodes exploration	migratory and false-sharing	4.3%
TSP	18 city map; run-to-completion	false-sharing	13.8%
Migratory	512 exclusive access cache lines	migratory	5.2%
ProdCon	2K shared cache lines and 8K private cache lines	single producer-multiple consumer	7.1%

keeper changes for a delegated line in a time period where there is at least one L1 sharer including the keeper. Interestingly, we found that most of the time the keeper is not changed. Across all benchmarks, 96.9% of the delegations remain at one keeper, while for 1.7%, 0.5%, and 0.9% of the delegations the keeper changes once, twice, or more than twice respectively.

C. Effect of Decoupling Data and Metadata on L2 Cache Capacity

1) Unused lines in L2: As discussed in Section III-A, delegated lines do not need data storage. To see how many lines are being delegated in general, we perform a sampling run where at every 25K cycle interval (10K cycles for microbenchmarks), we take a snapshot of the L2 cache and count the number of sets with one, two, or more lines delegated. Thus this sampling is both in space (all sets) and time. Figure 4 shows the breakdown of frequency of the number of lines delegated.

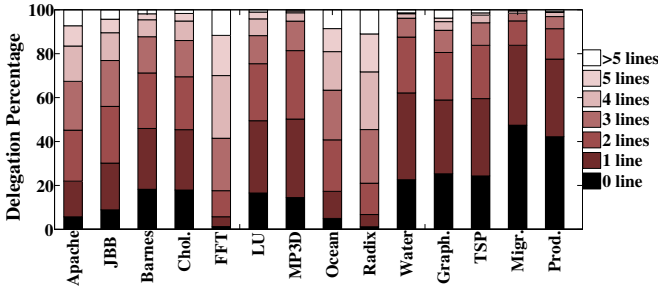

Fig. 4. Percentage of total execution time specific number of lines per L2 cache set were delegated to L1s. Numbers are collected for each set using a 25K cycle sample interval.

From the figure, we can see that sometimes a significant number of lines in a set are delegated. In general, most of the time, only a handful of lines are delegated: except for three applications, in 80% of the cases, the number of lines delegated is less than 4. The average number of delegated lines per set ranges from 0.8 to 3.82 lines (with a suite-wide average of 2.1). This suggests that having a few extra metadata blocks per set would be an economic approach to improving cache capacity utilization. For the rest of the evaluation, we choose to add 4 extra metadata blocks per set, and reduce one data block from the base configuration in order to compensate for the metadata area overhead.

2) Effect on L2 misses: Clearly, the benefit of better cache space utilization is a (highly non-linear) function of the working sets. If cache size is increased just enough to capture the next working set size, L2 misses would reduce dramatically. The bars in Figure 5 show the L2 misses under different configurations normalized to that of L2S. For example, L2S_T20D20 increases the number of ways from 16 to 20, adding 25% capacity. We also show the absolute L2 miss rate of the baseline configuration (L2S) (measured in MPKI – misses per 1K non-sync instructions) in the form of a curve.

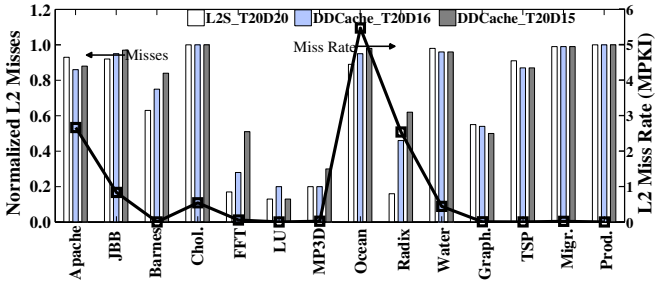

Fig. 5. Normalized L2 misses with respect to base L2S protocol (bar chart) and L2 miss rates for base protocol (square markers in dashed line). TxDy (in protocol names) indicates x tag and y data lines.

We see that for some applications (*e.g.*, *LU* and *MP3D*), an increase in capacity substantially reduces the number of L2 misses (by about 80%). We also see that by adding only 4 metadata ways to DDCache (DDCache_T20D16), DDCache can largely match the reduction in misses as a result of adding 4 data and metadata ways in L2S, but with a much smaller extra storage cost. In fact, even if we *reduce* 1 way of data to (over)compensate for the storage cost of adding 4 metadata ways (it uses 4% less space than the baseline L2S), the resulting DDCache_T20D15 still compares favorably against the baseline. For Apache, Ocean, and Radix, workloads with substantial L2 misses, on average, L2S_T20D20 still eliminates 19% of the L2 misses incurred by L2S_T16D16, while DDCache_T20D16 and DDCache_T20D15 eliminate 14% and 10% of the L2 misses

respectively.

3) Performance impact of cache capacity: To quantify the performance impact of just the capacity saving feature of our design, we simulate two DDCache configurations: both with 16 data ways, but one with 16 tag ways and the other with 20 tag ways. The difference in execution speed is therefore due to the cache capacity improvement. For contrast, we also show the impact of increasing the cache from 16 ways to 20 ways in L2S. The results are shown in Figure 6.

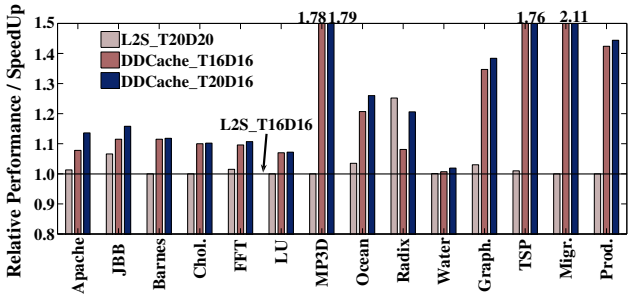

Fig. 6. Effect of extra tags in terms of speedup with respect to L2S.

As can be expected, a significant performance gain is only possible when both the absolute miss rate is high and the reduction in that miss rate is significant. In our suite of applications, only *Radix* falls into this category as can be seen by the large performance gain of adding 4 extra ways to the baseline L2S. In DDCache, at a much lower cost of adding only 4 extra tags, we also achieve significant performance gain for *Radix*. It is worth noting that for some applications (*e.g.*, *Apache*), the effect of adding 4 extra tags to DDCache is even more pronounced than that of adding 4 full-blown ways to L2S. This is because L2 evictions will cause mispredictions in our keeper/sharer predictor. The additional effective capacity reduces such mispredictions and thus also helps the acceleration of coherence.

D. Overall Performance Improvement

We now perform an evaluation of the complete system. For DDCache, we choose the configuration that has 20 metadata ways and 15 data ways (DDCache_T20D15). Considering all the extra storage needed for our design, this configuration still takes less total storage than the baseline L2S. We start with a performance analysis when all features are turned on, including the producer forwarding feature discussed in Section III-E. Figure 7 shows the speedup achieved by the target DDCache configuration relative to the baseline L2S.

We see that the DDCache design achieves significant performance improvement. Among the applications, *MP3D*, *Graph-Mine*, *TSP*, and *Producer-consumer* benefit mainly from delegation and direct L1-L1 communications. *Apache*, *JBB*, *Ocean*, and *Radix* mainly benefit from extra L2 capacity. *Barnes*, *Cholesky*, *FFT*, and *LU* have small additive contributions from extra capacity, delegation, and direct L1-L1 communications. *Water* does not benefit from any of these features as it is not L2 capacity hungry nor does it have any predictable access patterns that can utilize delegation and direct L1-L1 communications. *Migratory* has the advantage of direct L1-L1 communications for migratory, read-modify-write accesses.

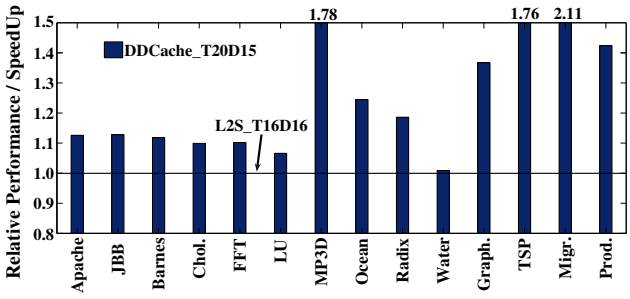

Fig. 7. Performance improvement of DDCache (20-tag and 15-data) normalized to L2S.

Overall, DDCache_T20D15 achieves a speedup over the baseline L2S of up to 1.78 with a geometric mean of 1.24. Counting the two microbenchmarks, the mean speedup goes up to 1.30.

E. Interconnect Traffic Reduction

Figure 8 shows the breakdown of the number of on-chip flit-hops for baseline L2S and DDCache_T20D15, normalized to L2S (Table I). The flit-hop numbers are accumulated by adding the number of hops traversed by all the flits. The flits are classified as (1) Data traffic between tiles, (2) Control traffic including requests and non-data responses between caches, and (3) Misprediction induced extra request traffic in DDCache. As we can see, while DDCache incurs some overhead in the form of extra control traffic, the amount of data traffic is cut almost by half. Since data traffic dominates, the end result is a significant reduction of overall on-chip traffic. Although not shown in the figure, the off-chip traffic is also significantly lower thanks to the higher effective on-chip cache capacity. Across all benchmarks, DDCache_T20D15 generates only 65% on-chip and 74% off-chip traffic (both geometric mean) compared to L2S. Bandwidth or traffic generation rate is also reduced for DDCache despite its faster execution speed; on average, DDCache demands only 82.6% of the bandwidth of L2S.

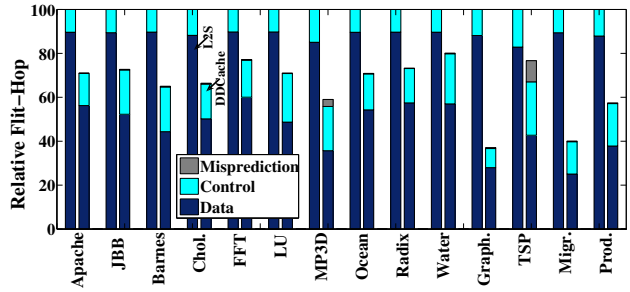

Fig. 8. Breakdown of interconnect flit-hops normalized to L2S.

F. Dynamic Energy and Power Requirements

Table III lists the energy consumption per access derived from Cacti 6.0. These numbers, along with collected access statistics, are used to derive dynamic energy numbers for DDCache_T20D15. Power (dynamic) is calculated from energy divided by execution time.

Figure 9 shows dynamic energy and dynamic power consumption of DDCache normalized to L2S. Dynamic energy

TABLE III
DYNAMIC ENERGY CONSUMPTION VALUES PER ACCESS FOR INDIVIDUAL STRUCTURES IN DDCACHE USING 45NM TECHNOLOGY (VALUES ARE IN FEMTO-JOULES (FJ))

L1$		Predictor	L2$		Router/Interconnect			
Tag	Data	Access	Tag	Data	BufRd	BufWr	Xbar	Arbiter
2688	16564	18593	58299	76621	760	1187	24177	402

consumption of DDCache ranges between 38% and 106% of the dynamic energy consumed by L2S, with a suite-wise average of 74%, whereas the average dynamic power consumed is 95% that of L2S (ranges from 81% to 130%). Due to the increase in execution speed when using DDCache, reduction in energy is higher than reduction in power. These energy and power numbers compare only on-chip storage resources and do not include energy consumed for off-chip communication and DRAM accesses.

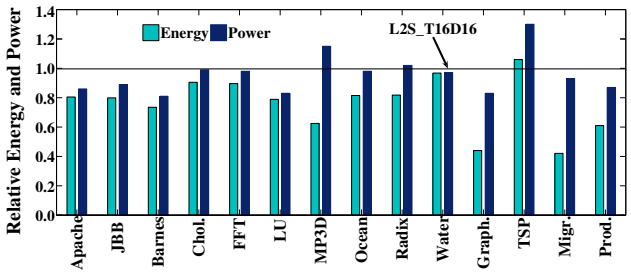

Fig. 9. Normalized dynamic energy and power consumption of DDCache with respect to L2S.

G. Storage Reduction/Overhead

Section III-A describes the extra fields and structure added in DDCache when compared to L2S. For the target system (Table I), at the L2 level, we reduce 1 data block (64B) and add 4 metadata blocks per set. This reorganization reduces 188 KB (-4.3%) storage bits compared to our baseline L2S.

At the L1 level, delegation requires an added 16-bit sharing bit-vector per L1 data cache line, which implies an extra 32 KB for all 16 cores. DDCache uses prediction and fine-grain sharing pattern optimizations as in ARMCO [16], which results in 70 KB of overhead for the 16 cores in the target configuration. On balance, the modifications reduced storage needs by a little over 1%.

H. Comparison with ARMCO and Victim Migration

We now contrast three different designs: ARMCO [16], Victim Migration (VM) [40], and DDCache (DDCache_T20D15). Compared to DDCache and ARMCO, VM uses a very different mechanism to reduce on-chip communication latency. By migrating private or replicating read-shared data to the local L2, remote accesses are reduced. However, when remote access is required, such as when servicing coherence misses, latency increases because all L1 misses have to check the local L2 first. Also, data migrated from the home results in extra indirection for access by other nodes. Finally, replication reduces on-chip remote access at the expense of reduced L2 capacity and thus may result in more expensive extra off-chip accesses.

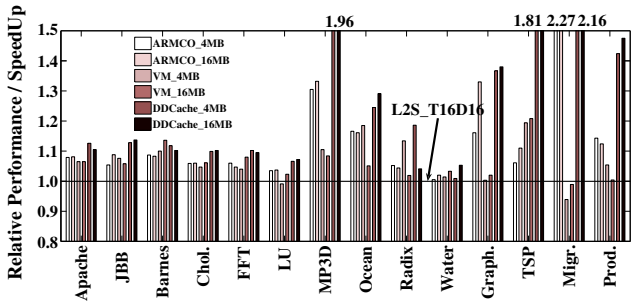

Fig. 10. Comparative performance improvement of ARMCO, Victim Migration (VM) and DDCache with respect to L2S. Each protocol is simulated with both 4MB and 16 MB L2.

Figure 10 shows the performance comparison. Two different L2 sizes (4MB and 16MB) are simulated. VM has 16 regular tag-data ways and 16 extra victim tags per set, L2S has 16 tag-data ways, and DDCache has 20 tag ways and 15 data ways. Results are shown as speedups relative to L2S (either the 4MB or 16MB configuration).

We first analyze VM's performance. Microbenchmark *Migratory* has a small L2 footprint and all the data are accessed in a read-modify-write pattern by each processor in sequence. Hence, the use of victim tags and local replicas results in a reduction in performance. Accesses that miss in the L1 are further delayed by checking the local L2 slice. Overall, we see a 6.1% performance degradation for *Migratory* with VM. Similarly, performance improvement is insignificant for *LU*, *FFT*, *Water*, *Cholesky*, and *Producer-Consumer*. In contrast, *TSP*, *Ocean*, and *Radix* are the best performing among the benchmarks. *Radix* and *Ocean* are L2 capacity hungry and so VM provides a performance boost. *TSP* gets benefits from migrating private data lines to the local L2 slice.

ARMCO achieves performance improvements from direct L1-to-L1 data transfer by predicting the closest potential sharer/owner of the requested data and using fine-grain sharing pattern optimizations. The simulated performance improvements are consistent with those presented in [16]. With a smaller L2 cache size in our target system, the speedup is also smaller.

DDCache outperforms both VM and ARMCO. With a 4MB L2, the average (geometric mean) speedup against the baseline is 1.08 and 1.09 for VM and ARMCO respectively. In contrast, DDCache achieves an average speedup of 1.24 (w/o microbenchmarks). For a 16MB L2, average speedups for VM, ARMCO, and DDCache are 1.07, 1.12, and 1.25 respectively.

I. Performance Scalability

So far, our results and analysis have focused on a 16-core CMP. We simulated L2S and DDCache for a 32-core CMP. Figure 11 compares the performance of DDCache on a 32-core CMP to that on a 16-core CMP. Speedup increases with the increase in the number of cores in the system. Most of the benchmarks' 32-core speedup relative to L2S is higher (1.3 versus 1.24 at 16 cores). With the increase in the number of cores, the number of hops to access the home node increases, resulting in greater savings with DDCache when home node access is eliminated. DDCache achieves this via localized and direct communication by delegation.

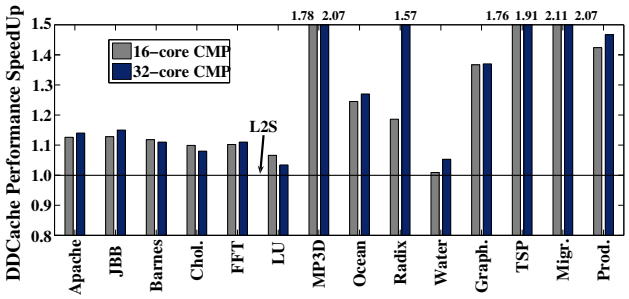

Fig. 11. Speedup of ARMCO, VM, and DDCache on a 16-core and 32-core CMP normalized to the corresponding L2S performance (with the same L2 cache size).

V. Related Work

The trend toward utilizing increasing transistor budgets for larger L2 caches and many cores increases the penalty for a miss at each processor core. Many non-uniform cache designs with adaptive protocols have recently been proposed in order to reduce L1 miss penalties [6], [8], [10], [11], [16], [20], [32], [33], [40], [41]. We focus here on those most directly related to DDCache.

Zhang and Asanovic's Victim Migration (VM) [40] proposal allows L2 cache lines in a globally shared cache to migrate among banks for improved locality of access. VM replicates the tag array (as opposed to DDCache's extra tag ways) at the home L2 in order to redirect requests for the cache line to the current owner. The replicated tags eliminate the need for a data replica at the L2 home when a cache line has been migrated, enabling more effective L2 capacity utilization. However, non-home L2 banks can still hold multiple copies in case data is accessed by multiple cores, resulting in potentially reduced L2 capacity.

CMP-NuRAPID [11] adapts to different sharing patterns by either replicating data at the L2 as in VM for read-shared data or performing in-situ communication in case of active read-write or false sharing. Data is also migrated to the L2 bank closest to the accessing cores. In-situ communication requires a write-through L1 cache, which can increase both bandwidth and energy demands on the L1-L2 interconnect. Moreover, as with VM, CMP-NuRAPID doubles L2 tag space and requires forward pointers (data group and frame number) and backward pointers (set index, tag, and processor id), resulting in higher storage costs.

Adaptive Selective Replication [6] enhances CMP-NuRAPID by controlling replication based on a cost/benefit analysis of increased misses versus reduced hit latency. Cooperative Caching [8] borrows concepts from software cooperative caching [4], [15], [38] to allow controlled sharing of essentially private caches, but requires a centralized coherence engine. Huh et al. [17] study controlled replication, migration, and cache sharing in a CMP NUCA cache. In the presence of migration, however, successive lookups across tags of all banks may be required.

Eisley et al. [14] reduce cache miss latency and on-chip bandwidth requirements by using a directory structure embedded in the network in order to get data directly from a sharer/owner that is on the way to the home node. Set-up/tear-down of the directory tree, however, can make overall latency variable due to potential deadlock recovery. Ros et al. proposed Direct Coherence [32], [33], which tries to avoid the home node indirection of directory protocols by storing the directory information with the current owner/modifier of the block and delegating coherence responsibility to this node. The home node has the role of storing the identity of the owner and is notified only if there is a change in ownership. Ownership is changed on every write miss or write upgrade. As a result of this policy, direct coherence may result in unnecessary redelegation in the case of active read-write sharing, which DDCache avoids via sharing pattern optimizations.

ARMCO [16] is a design that allows data to be sourced from any sharer (not necessarily the owner) via direct L1-L1 communication, with the goal of leveraging locality of access. Although ARMCO removes L2 directory/home access from the critical path, the directory must still be kept up to date, requiring global (across chip) communication. DDCache is implemented on top of ARMCO, leveraging its fine-grain sharing optimizations, while using delegation to avoid the traffic required to update the home.

Several coherence protocols that detect and optimize coherence actions for specific sharing patterns have been proposed in the past [9], [12], [35]. These protocols were leveraged in the development of ARMCO's sharing pattern optimizations. There has also been considerable work in the area of predicting coherence communication [19], [22], [26], [30]. Based on the design space outlined in [19], we employ a per-core address-based predictor of the current keeper or sharer.

VI. Conclusions

In this paper, we present DDCache, a cache coherence protocol that reduces on-chip communication overheads and provides larger effective cache capacity by eliminating duplication of data in on-chip cache space. Duplication is eliminated by decoupling L2 data and metadata. Communication overhead is reduced by delegating coherence enforcement to an L1 owner/modifier and away from the home L2 node. DDCache inherently supports direct L1-to-L1 communication via prediction and fine-grain sharing optimizations.

Our proposed DDCache protocol is able to reduce the average L1 miss penalty and reduce L2 misses resulting in an average (geometric mean) speedup of 1.3 (1.24 w/o microbenchmarks), with improvements ranging from 1% to 111%. These performance gains are coupled with a reduction in both on-chip and off-chip traffic (traffic generated is 65% and 74% of the on-chip and off-chip traffic generated by the base, respectively) as well as a reduction in on-chip memory and interconnect dynamic energy and power consumption (consuming 74% of the energy (95% of the power) of the base system). These benefits are achieved while using comparable total on-chip cache storage (actually a 1.32% reduction in the design we evaluate) when compared to a conventional L1-L2 hierarchy, with the caveat of some added complexity in the cache controllers.

Our design supports truly parallel workloads with actively shared data by optimizing communication to occur directly between sharing cores. Private data delegation also increases overall cache capacity. In future work, we will explore support for multiprogrammed workloads. While DDCache localizes

communication among sharing nodes, overflow/eviction from the L1s still requires accessing the home L2. Traffic to the home L2 is potentially non-local, resulting in interference with simultaneously executing threads. We will explore the benefits of some combination of DDCache and victim (or home) migration.

ACKNOWLEDGMENT

This work was supported in part by NSF grants 0702505, 0411127, 0615139, 0834451, 0509270, 0719790, 0747324, 0829915, and 0824075; and NIH grants 5 R21 GM079259-02 and 1 R21 HG004648-01.

REFERENCES

[1] M. E. Acacio, J. Gonzalez, J. M. Garcia, and J. Duato. Owner prediction for accelerating cache-to-cache transfer misses in a cc-NUMA architecture. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–12, Nov. 2002.

[2] A. Agarwala, R. Simoni, M. Horowitz, and J. Hennessy. An evaluation of directory schemes for cache coherence. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 280–289, May 1988.

[3] A. Alameldeen, M. Martin, C. Mauer, K. Moore, M. Xu, M. Hill, D. Wood, and D. Sorin. Simulating a $2m commercial server on a $2k pc. *IEEE Computer*, 36(2):50–57, Feb. 2003.

[4] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. *ACM Transactions on Computer Systems*, 14(1):41–79, Feb. 1996.

[5] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *ACM Sigmetrics Conf. Measurement and Modeling of Computer Systems*, pages 151–160, July 1998.

[6] B. Beckmann and M. Marty. ASR: Adaptive selective replication for CMP caches. In *39th Annual International Symposium on Microarchitecture*, pages 443–454, Dec. 2006.

[7] G. Buehrer, S. Parthasarathy, and Y. Chen. Adaptive parallel graph mining for CMP architectures. In *Proceedings of the Sixth International Conference on Data Mining*, pages 97–106, Dec. 2006.

[8] J. Chang and G. S. Sohi. Cooperative Caching for Chip Multiprocessors. In *Proceedings of the 33rd International Symposium on Computer Architecture*, pages 264–276, June 2006.

[9] L. Cheng, J. Carter, and D. Dai. An adaptive cache coherence protocol optimized for producer-consumer sharing. In *13th International Symposium on High Performance Computer Architecture*, pages 328–339, Feb. 2007.

[10] Z. Chishti, M. Powell, and T. Vijaykumar. Distance associativity for high-performance energy-efficient non-uniform cache architectures. In *Proceedings of the 36th International Symposium on Microarchitecture*, page 55, Dec. 2003.

[11] Z. Chishti, M. Powell, and T. Vijaykumar. Optimizing Replication, Communication, and Capacity Allocation in CMPs. In *Proceedings of the 32nd International Symposium on Computer Architecture*, pages 357–368, June 2005.

[12] A. Cox and R. Fowler. Adaptive cache coherency for detecting migratory shared data. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 98–108, May 1993.

[13] D. E. Culler and J. P. Singh. *Parallel Computer Architecture: a Hardware/Software Approach*. Morgan Kaufmann, 1999.

[14] N. Eisley, L. Peh, and L.Shang. In-network cache coherence. In *Proceedings of the 39th International Symposium on Microarchitecture*, pages 321–332, Dec. 2006.

[15] M. J. Feeley, W. E. Morgan, F. H. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing global memory management in a workstation cluster. In *15th ACM Symposium on Operating Systems Principles*, pages 201–212, Dec. 1995.

[16] H. Hossain, S. Dwarkadas, and M. C. Huang. Improving support for locality and fine-grain sharing in chip multiprocesors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 155–165, Oct. 2008.

[17] 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 International Conference on Supercomputing*, pages 31–40, June 2005.

[18] Intel News Release. Intel research advances "era of tera.". http://www.intel.com/pressroom/archive/releases/20070204comp.htm, Feb. 2007.

[19] S. Kaxiras and C. Young. Coherence communication prediction in shared-memory multiprocessors. In *Proceedings of the 6th International Symposium on High Performance Computer Architecture*, pages 156–167, Jan. 2000.

[20] C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 211–222, Oct. 2002.

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

[22] A. Lai and B. Falsafi. Memory sharing predictor: The key to a speculative coherent DSM. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 172–183, May 1999.

[23] J. Laudon and D. Lenoski. The sgi origin: A ccnuma highly scalable server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 241–251, June 1997.

[24] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the dash multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, May 1990.

[25] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Haogberg, F. Larsson, A. Moestedt, and B.Werne. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.

[26] M. Martin, P. Harper, D. Sorin, M. Hill, and D. Wood. Using destination-set prediction to improve the latency /bandwidth tradeoff in shared memory multiprocessors. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 206–217, June 2003.

[27] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *ACM SIGARCH Computer Architecture News*, 33(4):92–99, Sept. 2005.

[28] C. McNairy and R. Bhatia. Montecito: A dual-core, dual-thread itanium proessor. *IEEE Micro*, 25(2):10–20, Mar-Apr 2005.

[29] R. Merritt. IBM weaves multithreading into Power5. *EE Times*, 2003.

[30] S. Mukherjee and M. Hill. Using prediction to accelerate coherence protocols. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 179–190, June 1998.

[31] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing nuca organizations andwiring alternatives for large caches with cacti 6.0. In *Proceedings of the 40th International Symposium on Microarchitecture*, pages 3–14, Dec. 2007.

[32] A. Ros, M. E. Acacio, and J. M. Garcia. Direct coherence: Bringing together performance and scalability in shared-memory multiprocessors. In *Proceedings of the 14th International Conference on High Performance Computing*, pages 147–160, Dec. 2007.

[33] A. Ros, M. E. Acacio, and J. M. Garcia. DiCo-CMP: Efficient cache coherency in tiled CMP architectures. In *IEEE International Symposium on Parallel and Distributed Processing*, pages 1–11, Apr. 2008.

[34] Standard Performance Evaluation Corporation. Specjbb2005. http://www.spec.org/jbb2005/, 2005.

[35] P. Stenström, M. Brorsson, and L. Sandberg. An adaptive cache coherence protocol optimized for migratory sharing. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 109–118, May 1993.

[36] Sun News Release. Sun expands solaris/sparc cmt innovation leadership. http://www.sun.com/aboutsun/pr/2007-01/sunflash.20070118.3.xml, Jan. 2007.

[37] The Apache Software Foundation. Apache. http://www.apache.org/, 2008.

[38] A. Wolman, G. M. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. M. Levy. On the scale and performance of cooperative web proxy caching. In *17th ACM Symposium on Operating Systems Principles*, pages 16–31, Dec. 1999.

[39] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. Methodological considerations and characterization of the SPLASH-2 parallel application suite. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.

[40] M. Zhang and K. Asanovic. Victim Migration: Dynamically adapting between private and shared CMP caches. Technical report, MIT-CSAIL, Boston, MA, Oct. 2005.

[41] M. Zhang and K. Asanovic. Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 336–345, June 2005.