POPS: Coherence Protocol Optimization for both Private and Shared Data

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

***Abstract*—As the number of cores in a chip multiprocessor (CMP) increases, the need for larger on-chip caches also increases in order to avoid creating a bottleneck at the off-chip interconnect. Utilization of these CMPs include combinations of multithreading and multiprogramming, showing a range of sharing behavior, from frequent inter-thread communication to no communication. The goal of the CMP cache design is to maximize capacity for a given size while providing as low a latency as possible for the entire range of sharing behavior.**

In a typical CMP design, the last level cache (LLC) is shared across the cores and incurs a latency of access that is a function of distance on the chip. Sharing helps avoid the need for replicas at the LLC and allows access to the entire on-chip cache space by any core. However, the cost is the increased latency of communication based on where data is mapped on the chip. In this paper, we propose a cache coherence design we call POPS that provides localized data and metadata access for *both* shared data (in multithreaded workloads) *and* private data (predominant in multiprogrammed workloads). POPS achieves its goal by (1) decoupling data and metadata, allowing both to be delegated to local LLC slices for private data and between sharers for shared data, (2) freeing delegated data storage in the LLC for larger effective capacity, and (3) changing the delegation and/or coherence protocol action based on the observed sharing pattern.

Our analysis on an execution-driven full system simulator using multithreaded and multiprogrammed workloads shows that POPS performs 42% (28% without microbenchmarks) better for multithreaded workloads, 16% better for multiprogrammed workloads, and 8% better when one single-threaded application is the only running process, compared to the base non-uniform shared L2 protocol. POPS has the added benefits of reduced on-chip and off-chip traffic and reduced dynamic energy consumption.

***Keywords*-Cache Coherence; Chip Multiprocessors; Decoupled Cache; Delegable Cache; Bloom-Filter; Localized Access; POPS;**

I. Introduction

As advances in technology result in increased on-chip transistor density, designers are challenged to provide higher performance without significant changes to power budgets and pin counts. The extra transistors will likely be used both for more functionality (increasing core count) and more storage (larger caches). The larger number of cores places increasing pressure on on-chip cache resources to deliver low latency access to a large amount of data. Workloads on these future many-core processors will include combinations of single-threaded, multithreaded, and multiprogrammed workloads with a range of sharing and memory access behavior — from frequent inter-thread communication to essentially private data with no communication.

In order to improve scalability, the last level cache (LLC) is typically shared across the cores and incurs a latency of access that is a function of distance on the chip. Sharing helps avoid the need for replicas at the LLC when data is shared by more than one core and allows access to the entire on-chip cache space by any core. However, the cost is the increased latency of communication based on where data is mapped on the chip. In a conventional directory-based protocol for such a shared L2 cache (L2S), 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. Second, the home node retains a copy of data even when the copy is stale. Many server applications lose almost half of their potential performance (assuming all data could be accessed at the latency of the local cache) due to the increased latency of on-chip cache accesses [14].

Several coherence protocol optimizations to address the challenge of the non-uniform latency while retaining the benefits of a shared cache have been proposed in the literature [5], [7], [10], [15], [17], [32]. Each of these addresses aspects of optimizing data communication for specific workload scenarios. For example, DDCache [17] focuses on streamlining shared data communication and evaluates performance on multithreaded workloads. Victim migration [32] focuses on providing the latency of a localized/private LLC while still taking advantage of a shared LLC. Eviction from the cache levels closer to the processor results in a replica being created at the local L2 slice so that subsequent misses are able to retrieve data faster. Capacity pressure due to the replicas is mitigated through the use of victim tags at the home (pointers to the current data holder(s) that do not need associated data space).

Both DDCache and victim migration (VM) suffer from performance anomalies for specific workloads. Using access pattern specific optimizations as well as metadata delegation to L1 caches, DDCache optimizes traffic for actively shared data and performs well for multithreaded workloads that share data. However, for private data predominant in single-threaded or multiprogrammed workloads, the latency of communicating with the data home must be paid when the L1 cache capacity is exceeded, and the performance gain over L2S is small. Victim migration, on the other hand, uses local LLC slices to increase the range of data that can be served with low latency. While capacity pressure from this replication is eliminated by deleting the data at the home, sharing by more than one core still creates multiple replicas with its corresponding cache pressure. Hence, VM improves performance over L2S for multiprogrammed workloads, but shows very little performance gain for multithreaded workloads.

In this paper, we propose a cache coherence design we call POPS (coherence ***P***rotocol ***O***ptimization for ***P***rivate and ***S***hared data) that provides optimized coherence, localized data and metadata access for *both* shared data (in multithreaded work-

loads) *and* private data (predominant in multiprogrammed and single-threaded workloads), and larger effective LLC capacity. POPS achieves its goals by (1) decoupling data and metadata, allowing each to be delegated to either an L1 cache or a local LLC slice when evicted from the L1 (in the latter case, for private data), resulting in lower access latency, (2) freeing delegated data storage in the LLC, thereby ensuring a single copy in the LLC or one or more copies in the levels closer to the processor, resulting in larger effective cache capacity, and (3) changing the delegation and/or coherence protocol action based on the observed sharing pattern, supporting fine-grain sharing through L1-L1 direct accesses (no directory indirection) via prediction and access-pattern specific optimizations.

Our analysis on an execution-driven full system simulator using multithreaded and multiprogrammed workloads shows that POPS performs 42% (28% without microbenchmarks) better for multithreaded workloads, 16% better for multiprogrammed workloads, and 8% better when one single-threaded application is the only running process, compared to the base non-uniform shared L2 protocol. POPS has the added benefits of reduced on-chip and off-chip traffic and reduced dynamic energy consumption.

II. Design Overview

Figure 1 presents a high-level block diagram view of a tiled CMP architecture that we use as the base for our design. Each tile in the CMP contains a processor core, private L1 caches, and a slice of the globally-shared L2 cache (the last level cache), which is statically-mapped (line interleaved) by physical address in the base protocol. The tiles are interconnected by a 4x4 mesh network (for a 16-core system).

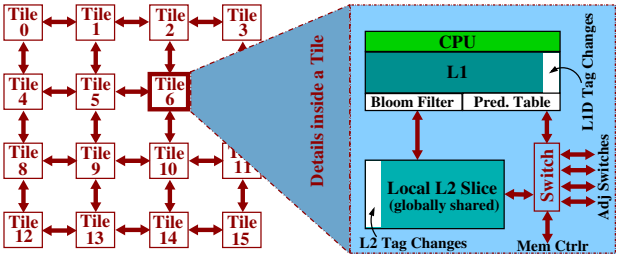

Figure 1. Block diagram of the underlying tiled CMP architecture depicting a processor with 16 cores where the right side highlights details inside a tile with additions for POPS identified by the white (unshaded) regions.

Changes to the base design in order to support the POPS protocol are shown in white (unshaded) in Figure 1. To support delegation, a sharer bit vector is added to the L1D cache line tags along with extra metadata ways in each L2 set. As a result, an L1D can be delegated to keep the coherence. At any time, only a single node (either the original L2 node, an L1D, or another L2 node) is in charge of keeping/maintaing coherence for a particular line. This node is referred to as the *coherence keeper* (or *keeper* in short). Also, L2 metadata and data are not tightly bound. 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.

The local L2 slice retains a copy of lines evicted from the local L1 under certain conditions for faster data availability in the presence of temporal locality. Specifically, in POPS only *delegated* lines that are evicted from the local L1 are migrated to the local L2 slice. To avoid the extra latency of a local L2 slice check on a miss in the L1, a bloom filter is added to represent delegated lines present in the local L2 slice — a miss indicates the absence of a local replica. On an L1 cache miss, the coherence logic works as usual by sending the request (*e.g.*, read or upgrade) to the coherence keeper. The only difference is that the keeper is not necessarily the home node; instead it can be any L1 or L2 (local or remote) slice in the system.

To optimize for fine-grain sharing, this baseline CMP is also augmented with architectural support (similar to those used in ARMCO [16]) to identify access patterns and predict coherence targets.

III. Protocol and Hardware Design

A. Extra Hardware in POPS

Figure 2 provides more details on the structure additions for POPS. We elaborate on the additions below.

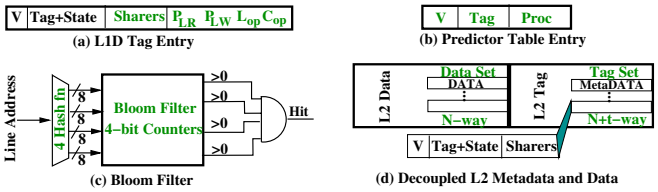

Figure 2. Hardware changes made for POPS. Changes and extra structures are represented using a lighter color.

1) Hardware for Decoupling and Delegation: Figure 2-(a) and (d) identify the changes at the L1 and L2 in order to support decoupling and delegation. Metadata and data in an L2 set are freed from their logical one-to-one binding and some extra metadata lines are added.[1] Metadata lines can point to any data line in the set or no data. The coherence management can be shifted to any L1 or L2 slice (delegation) and so a sharers list bit-vector in each L1D cache line (*Sharers* in Figure 2-(a)) is added to support delegation. This node (where the line is delegated) becomes the coherence keeper and is responsible for coherence ordering of the line. When the line is delegated, metadata points to no data in the L2 set and the sharers bit-vector points to the delegated location. These changes are similar to changes made in DDCache [17] to support delegation, with the addition of allowing delegation to any L2 slice as well as any L1.

2) Hardware for Controlled Relocation to non-Home L2 slice: To optimize access latency, a line evicted from L1 may be stored in the local L2 slice, essentially making a local replica. However, replication can reduce effective capacity and can thus improve on-chip cache access time at the potential expense of increasing off-chip accesses. We therefore follow a simple heuristic that only allows "replication" for the delegated lines evicted from L1. Recall that when a line is delegated, the directory gives up its data storage. Hence there is no actual duplication of data at the L2 level (utilizing the structures outlined in Figure 2-(d)).

When handling an L1 miss, we need to know if the data is in the node's local L2 slice. To avoid unnecessary local L2 access, there is one 4-bit counter-based, 1K-entry bloom filter (shown in Figure 2-(c)) added to the L1 miss path. The bloom filter tracks the evicted lines kept at the local L2 slice. The bloom filter table is indexed by indices generated from four hash functions on the cache line address, each indexing into a disjoint 256-entry region. Once a delegated line is evicted from the L1, the line address is added into the bloom filter and once that line is brought back to the L1 or evicted from

[1]The same relative area overhead as the baseline design can be retained by possibly scavenging the storage of one data line in each L2 set.

the L2, the line address is removed from the bloom filter. While both false postives and false negatives are possible in this design, in the common case, there are few false negatives (none in our experiments). The false positive rate is also relatively low, 3.14% on average across our workloads for the given configuration. The consequence of false positives is an extra local L2 tag access on the critical path and the consequence of false negatives is to go to the home node and get redirected back, at which point the bloom filter is updated.

3) Hardware for L1-L1 Communication: An address-based predictor table is added to the L1-miss path to find a potential close-by sharer or keeper of the requested line. On an L1 miss, if there is no replica at the local L2, this predictor table is consulted to find the potential closest data supplier (as in DDCache [17] and ARMCO [16]). Details of the predictor table entry are shown in Figure 2-(b).

4) Hardware for Sharing Pattern Optimization: Figure 2-(a) identifies the changes for sharing pattern optimization. As in DDCache [17] and ARMCO [16], we use $2log_2N+2$ bits (N being the number of tiles in the system) in each L1D cache line to track the access patterns and to decide appropriate actions including in-place accesses. The bits are P_{LR} (log_2N bits to identify the last reader), P_{LW} (log_2N bits to identify the last writer), C_{op} (1 bit flag to track whether multiple accesses have been made from the local tile without any intervening remote access), and L_{op} (1 bit flag to determine whether the last access to the cache line was read or write). On an L1 data cache hit, the fields P_{LR}, P_{LW}, C_{op}, and L_{op} are updated in order to maintain the necessary information to adapt to sharing patterns.

B. POPS Protocol Actions

Figure 3 shows the sequence of messages used to serve an L1 miss. Originating from the requester L1, a message can flow along any path between bloom filter, predictor table, local L2 slice, home L2, close-by sharer L1, and keeper L1. For any path it takes, the path label (consisting of numbers that might not be in sequence) represents logical ordering where a lower value indicates an earlier event. For example, an L1 miss satisfied (through prediction) by the keeper L1 will follow path $(1)-(2b)-(3a)-(9)$.

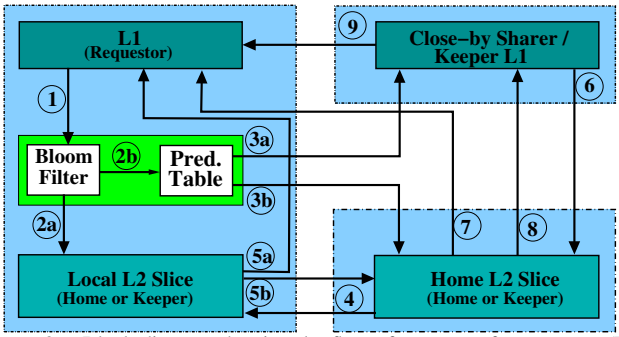

Figure 3. Block diagram showing the flow of messages for a request (L1 miss) among L1 (requestor), bloom filter, predictor table, local L2 slice, close-by L1 (keeper/sharer), and home L2. Shaded regions are used to identify separate tiles. The two shaded regions on the right can be parts of the same (or different) tile.

- **At the requester L1:** When there is an L1 cache miss and the local L2 slice is the home node then the request is sent to the local L2 slice. Otherwise, the bloom filter is consulted to determine whether the local L2 slice is the keeper of the requested line or not ((1) in Figure 3). The request is sent to the local L2 slice if the bloom filter indicates a hit ($(2a)$ in Figure 3). If the line does not exist in the local L2, or the bloom filter indicates a miss in the local L2, the predictor is consulted to find a nearby node that can potentially supply the data ($(2b)$ in Figure 3). If there is a predictor table hit, the request is sent to that node ($(3a)$ in Figure 3). This nearby node can be the keeper or just a sharer of the cache line. If no prediction is available, the request is sent to the home node (to be routed to the current keeper, if necessary) ($(3b)$ in Figure 3).
  If there is an eviction from the L1 cache, depending on the current state of the line, the line may be sent either to the non-home local L2 slice or to the home node. On eviction, if the L1 cache is the keeper of the line and there are no other L1 sharers in the system, the line (and keeper status) is delegated to the local L2 slice (on the assumption of temporal locality). If there are other sharers in the system or if this L1 is not the keeper, then no replica is made. For the former case, the line is "undelegated" and returned to the home node. As an optimization (not required for protocol correctness), sharers are informed about the change in keeper (no acknowledgements are necessary). If the L1 is not the keeper, the current keeper is informed of the eviction in order to keep the sharer bit vector up-to-date.
- **At the non-home L2 bank:** The non-home L2 slice can get a request to supply data in two ways. First, the local L1's bloom filter identifies the local L2 slice as the keeper ($(2a)$ in Figure 3). Second, requests from other L1s are forwarded to the non-home L2 bank from the home L2 ((4) in Figure 3). Requests from the local L1 are served by moving the line to the L1 in delegated state ($(5a)$ in Figure 3), after sending a message to the home L2 about the change in keeper. An exclusive request from another L1 is served by transferring keeper status to the requester and a read-only request is served by supplying a copy of data to the requester as well as returning keeper status to the home L2. In either case, if the L2 does not have the data, the request is forwarded to the home L2.
- **At the home L2 bank:** When there is a miss in the L1 cache, the bloom filter, or the local L2 slice, and the predictor table does not yield a predicted keeper, the request is sent to the home L2 ($(3b)$ in Figure 3). If a request is missed at the predicted node, the request is also forwarded 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. If the requested line is for private access then the line is also delegated.
  A delegated line will be surrendered back to the home L2 if the tag is evicted from the home L2. The keeper L1 will receive a request from the home to surrender and invalidate the line.
- **At the keeper L1:** 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) ($(3a)$ in Figure 3). Second, the request may be forwarded from the home ((8) in Figure 3).
  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 ((9) in Figure 3) and coherence information is updated in the background (as discussed later).

Coherence ordering: 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 on-going 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 acknowledgement 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 ((6) in Figure 3).
- **At a non-keeper and non-sharer L1:** Due to a misprediction, non-keeper and non-sharer L1s can get requests from other L1s. The requests are simply forwarded to home L2, which might serve the requests if the line is not delegated or forward the request to the keeper if the line is delegated.

C. Optimization for Private and Shared Data

In the optimization decision process, we classify data accesses into 4 categories: private, shared read-only, shared read-write, and migratory. The general guideline is to pick the 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 (and migratory) data:** Since private data will only be accessed by one thread, it should always be delegated to the accessing node. This avoids the need for upgrade requests and reduces 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 to the home with a hint bit indicating the line is probably private. When the request arrives at the home and the metadata shows that it is indeed not shared by other nodes, the line is treated as private and delegated to the requester. 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 (pointer to the current keeper). When delegated data is evicted from the L1, the line is replicated at the local L2 slice if there are no other sharers, as will be the case for private data.

  Although migratory data is shared among the cores, its behavior is similar to private data as at any time in the system only one core accesses the data. Hence we follow a similar optimization strategy for migratory data.
- **Shared data:** When data is shared, the appropriateness of delegation 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. Hence, 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. A read-shared line is not replicated at the local L2 on L1 eviction. Read-write shared lines can be replicated if at the time of L1 eviction, that is the only copy in the system.

D. Optimization for Sharing Patterns

When data is shared, unnecessary communication and indirection via the directory can be avoided by tailoring the coherence protocol to the access pattern. By using the hardware described in Section III-A4 we identify migratory, producer-consumer, false shared, read-only shared, or read-write sharing patterns and then apply pattern-specific optimizations as in [16].

E. Destination Prediction

If requests have to go through the home node to determine the current keeper, there will be little if any benefit from del-

egation. Hence, we use an address-based destination predictor to decide from where (keeper or a potentially closest regular sharer) we should fetch the data. When there is an L1 miss and the local L2 slice does not have the replica, the predictor table can help avoid indirection via the home.

The predictor table is a cache-like structure that keeps a valid bit, a tag, and a processor ID for the predicted destination node (Figure 2-(b)). 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 a 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.

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].

F. Replication and Replacement Policy

POPS has two goals: (1) to achieve larger effective cache capacity and (2) to achieve lower latency when data is in the cache. Lower latency can be achieve by allocating lines evicted from the L1 into the local L2 slice as in victim migration [32]. In order to reduce the interference from the displaced lines with cache lines that normally map to the local L2, POPS allocates only the delegated lines in the L1 that are not shared at the time of eviction, into the local L2 slice. The intuition behind this policy is to identify only the private data for local placement since maximum latency reduction is achieved for such data and avoid polluting the local L2 slice with shared data that shows less latency benefit. In order to further minimize the conflict between locally delegated and homed cache lines, a pseudo-LRU replacement policy is used across all lines, which favors cache lines with higher temporal locality.

IV. Performance Evaluation

A. Evaluation Framework

We use a Simics-based [22] full-system execution-driven simulator to evaluate the POPS protocol. We use the Solaris 10 operating system and SPARC architecture that is modeled faithfully in Simics. We use Ruby from the GEMS toolset [23], which is modified to encode all the protocols used in our analysis for cache memory simulation. For the cache, we model tag, data, transient buffers (MSHR), input and output queues, and coherence protocol. On top of the cache model, we also use a detailed model of the on-chip network (4x4 mesh), which models switches, IO buffers, and links between switches. We use GEMS and SLICC [23] to model the action of the protocol state machine as well as structures needed for POPS. Protocol actions that require accessing the bloom filter and predictor incur extra latency which delays the resulting network packets.

Our base design is a 16-core tiled CMP with a 4x4 mesh interconnect and an 8-core tiled CMP with a 2x4 mesh interconnect. Each tile contains separate private L1 instruction and data caches and a slice of the shared L2 as in the base system. The L1 cache and local L2 cache can communicate directly without going through the switch/mesh interconnect (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) with a MOESI-style directory-based protocol.

We compare POPS to DDCache [17], Victim Migration [32], and the base L2S. We also developed a version of Victim Migration with bloom filters to bypass local L2 access when there is no replica (which we call Modified Victim Migration). Table I provides the system configuration used in our simulation. The L2 is organized differently in different protocols. The base protocol, L2S, uses a 16-way set-associative shared L2 slice. POPS uses 16 metadata-data lines and 16 metadata-only lines in a set. DDCache uses a 20-way set-associative shared L2 slice with 16 metadata-data lines and 4 metadata-only lines. Victim Migration (VM) has a 16-way set-associative shared L2 slice along with a 16-way set-associative victim tag. The choice of metadata/victim tag ways added is a function of the potential for creating additional capacity. POPS and VM add 16 additional metadata tags (equal to the number of ways in the base L2S design). In the case of DDCache, the number of extra metadata ways is determined by the ratio of L1 capacity to L2 capacity. Empirical evidence suggests that DDCache's performance is best with the 4 extra metadata ways — additional ways are not well utilized. Modified Victim Migration adds a counting bloom filter (4 hash functions, each with 256 entries, 4-bit counters) to Victim Migration [32]. Additionally, both POPS and DDCache have a 512-entry 8-way associative predictor table on the L1 miss path. POPS also has a bloom filter as in modified victim migration.

Table I
System parameters

16-way/8-way CMP, Private L1, Shared L2	
Processor cores	16 in-order, single issue, 3GHz cores. Non-memory IPC=1, sequentially consistent (8 cores for 8-way CMP)
L1 (I and D) cache	64KB 2-way each, 64-byte blocks, 2-cycle
Predictor table	512 entry 8-way associative
Bloom filter	4 hash functions, 256-entry each, 4-bit counters
L2 cache	16MB, 16-way set associative, 16 banks (8 banks for 8-way CMP), 64-byte blocks, Sequential tag/data access, 14-cycle
Memory	4GB, 300-cycle latency
Interconnect	4x4 mesh (2x4 mesh for 8-way CMP), 4-cycle link latency, 128-bit link width, virtual cut-through routing

The interconnect network is modeled using GEMS [23]. The network link width is 16 bytes and so is the flit size. Messages of three different sizes (8 bytes, 16 bytes, and 72 bytes) are used for data and command communications. L2S and Victim Migration use 8-byte and 72-byte messages while DDCache and POPS use 8-byte, 16-byte, and 72-byte messages. 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 to be transmitted. Messages exchanged between L1s and L2s are treated as on-chip traffic and messages communicated between L2s and memory controller are treated as off-chip traffic. For power consumption modeling, we use Cacti 6.0 [26] 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.

B. Workloads

For our study, we use a wide range of programs that include single-threaded, multithreaded, and multiprogrammed workloads. Table II lists all the benchmarks used in all three types along with their input specifications. Single-threaded workloads are taken from SPEC CPU2006. Multiprogrammed workloads are the 8-way mixes of single-threaded workloads from SPEC CPU2006and SPEC CPU2000. Multithreaded workloads are from the commercial, scientific, mining, and branch and bound domains, or from microbenchmarks. In order to demonstrate efficiencies for specific access patterns, we have developed microbenchmarks with *producer-consumer* and *migratory* access patterns. To demonstrate the effectiveness of local data replication, we have developed *PrvRW* benchmark that reads and writes a specified amount of privately allocated memory (larger than L1 size but smaller than L2 bank size) repeatedly. As commercial workloads, we use the *Apache* web server with the surge [4] request generator and *SPECjbb2005*. Alameldeen *et al.* described these commercial workloads for simulation [2]. For scientific benchmarks, we have a large set of applications and kernels from the SPLASH2/SPLASH suites [31], which include *Barnes*, *Cholesky*, *FFT*, *LU*, *MP3D*, *Ocean*, *Radix*, and *Water*. Our multithreaded workload suite also includes a *graph mining* application [6] and a branch-and-bound based implementation of the traveling salesman problem (*TSP*).

All the single-threaded and multiprogrammed workloads use an 8-way CMP and the multithreaded workloads use a 16-way CMP as specified in Table I. They are compiled with optimization flags enabled before creating the checkpoints. As specified in Table II, most multithreaded workloads are simulated to completion using default input sets except in a few cases where small inputs were used to keep the simulation time manageable. Other multithreaded workloads are transaction-based and are simulated for a specific number of transactions. Single-threaded and multiprogrammed workloads are fast-forwarded for 40 billion instructions and then simulated for 100 million instructions for each of the programs. Faster programs are allowed to continue execution till the slowest one completes 100 million instructions. Reference inputs are used for all of these SPEC CPU programs for both single-threaded versions and multiprogrammed workload mixes.

Future server workloads are likely to run on CMPs with many cores, while multi-programmed desktop workloads are likely to run on CMPs with fewer cores that free die area for a larger on-chip cache. We therefore use an 8-core CMP for multiprogrammed and single-threaded workload simulation. In multiprogrammed mode, any set of programs can run concurrently and so we *randomly* chose a set of programs to form a mix. All possible combinations of 8 programs could be chosen. However, simulation time would be excessive. Our choice of mixes show a range of behavior with varying numbers of high and low miss rate applications in the mix, each with a different fraction of replica utilization.

C. POPS Performance Improvement

As a performance metric, we use (1) *execution time* in terms of processor cycles required to do the same amount of work for multithreaded workloads and (2) *throughput* in terms of instructions per cycle (IPC) for whole systems for multiprogrammed workloads. For single-threaded workloads, we also use *throughput* in terms of instructions per cycle executed on the core that the thread uses.

Figures 4, 5, and 6 show the performance improvements of POPS (normalized to L2S) for multithreaded, multiprogrammed, and single-threaded workloads respectively. On average, POPS improves performance by 42% (by 28% when 3 microbenchmarks are not included) for multithreaded, by 16% for multiprogrammed, and by 8% for single-threaded workloads.

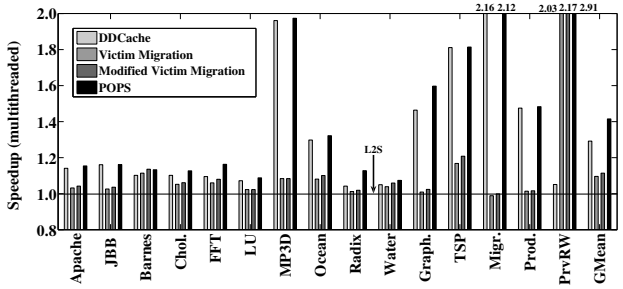

Figure 4. Performance improvement of POPS (normalized to L2S) along with DDCache, Victim Migration, and Modified Victim Migration for multithreaded workloads.

There are several contributing factors for POPS's speedup including (1) faster data availability by serving the request from the local L2, close-by L1 sharer, or keeper, (2) faster coherence via direct L1-L1 communication using delegation, (3) larger effective cache capacity by decoupling L2 metadata and data in a set and by adding extra metadata, and (4) sharing access pattern specific optimizations. The speedups of DDCache, VM, and Modified VM are shown along with POPS's speedup to justify the values of these contributing factors.

The *PrvRW* microbenchmark (in Figure 4) is designed to show the potential of faster data availability from controlled local L2 replication and achieves the maximum speedup of 2.9X. *PrvRW* is a multithreaded benchmark mimicking multiprogrammed behavior. Each thread allocates 128KB of memory in its stack and reads/writes that memory repeatedly (512 times). POPS delegates all these lines to the corresponding core as they are identified as private lines. Since the L1 cache size is 64KB, each iteration will result in data spilling to the local L2, which is large enough to hold them. Except for cold misses, accesses will be served by the local L2 slice for each thread and this will be much faster than going to the home L2 node as in L2S. Moreover, POPS has some added benefits from faster coherence using prediction and sharing pattern optimizations as in DDCache. VM should have performed close to POPS for *PrvRW* but does not because (1) delegated lines preferentially evict regular lines resulting in higher miss rates for data that maps to the local L2, (2) shared lines that are evicted due to capacity conflicts from multiple L1s can result in multiple replicas (at several L2 slices); (3) the local L2 slice is always on the critical path even when lines are not delegated, and (4) the L1 is in the communication path between the local L2 slice and the home L2 slice when the line is delegated (example, when supplying data to other L1s). In *Modified Victim Migration*, we address problem (3) with the bloom filter used in POPS, and see noticeable improvements. However, this does not address problems (1), (2), and (4). These problems still affect OS and shared library code and data. They play a significant role in the case of *PrvRW* (*e.g.*, 13% L1 misses from supervisor code itself in *PrvRW*). VM creates replicas for all these shared lines (1.2X

Table II
WORKLOADS AND INPUT SPECIFICATION. SINGLE-THREAD WORKLOADS COMPLEMENTED WITH L2 MPKI IN L2S AND REPLICA UTIL. (%) IN POPS.

Multithreaded	
Name	Input specifications
Apache	80K Tx fast forward, 2K Tx warmup, and 3K Tx for data collection
JBB2005	350K Tx fast forward, 3K warmup, and 3K for data collection
Barnes	8K particles; run-to-completion
Cholesky	lshp.0; run-to-completion
FFT	64K points; run-to-completion
LU	512x512 matrix, 16x16 block; run-to-completion
MP3D	40K molecules; 15 parallel steps; warmup 3 steps
Ocean	258x258 ocean
Radix	550K 20-bit integers, radix 1024
Water	512 molecules; run-to-completion
GraphMine	340 chemical compounds, 24 different atoms, 66 atom types, and 4 types of bonds; 200M instr.; warmup 300 nodes exploration
TSP	18 city map; run-to-completion
Migratory	512 exclusive access cache lines
ProdCon	2K shared cache lines and 8K private cache lines
PrvRW	128KB memory private read-write access/thread

Multiprogrammed (CPU 2006, ref input)	
Mix1	bzip2, gcc, mcf, milc, namd, gobmk, dealII, soplex
Mix2	povray, gcc, mcf, milc, hmmer, gobmk, dealII, soplex
Mix3	bzip2, sjeng, mcf, milc, namd, h264ref, dealII, soplex
Mix4	lbm, sjeng, mcf, milc, omnetpp, h264ref, dealII, soplex
Mix5	lbm, astar, mcf, milc, omnetpp, gcc, sphinx, soplex
Mix6	lbm, astar, Xalan, milc, omnetpp, h264ref, sphinx, soplex
Mix7	bzip2, astar, Xalan, milc, gcc, h264ref, hmmer, soplex
Mix8	bzip2, astar, mcf, milc, gcc, h264ref, specrand, dealII
Mix9	(CPU 2000) gzip, vpr, mcf, parser, eon, bzip2, twolf, mcf

Single-thread
SPEC CPU 2006 (L2 MPKI) (Replica Util%)
astar (0.37) (86)
bzip2 (0.58) (65)
dealII (0.33) (91)
gcc (0.20) (97)
gobmk (0.38) (92)
lbm (22.7) (37)
mcf (8.33) (16)
milc (15.0) (16)
namd (0.10) (94)
omnetpp (6.8) (65)
povray (0.01) (92)
sjeng (0.3) (70)
soplex (13.3) (34)
sphinx (10.4) (75)
Xalan (1.15) (67)

w.r.t. POPS) resulting in reduced effective L2 capacity and for *PrvRW* we see 19X more L2 evictions in VM w.r.t. POPS. Moreover, as VM does not maintain copies at the home for shared lines (but POPS does as shared lines are not delegated), supplying data to the subsequent sharers incurs extra latency. POPS attains a speedup of 2.9x for *PrvRW*. Although *PrvRW* amplifies the consequences of the differences in policies, these differences also affect, to varying degrees, other workloads, whether multithreaded or multiprogrammed. (Figure 7 later shows the effect on L2 misses.)

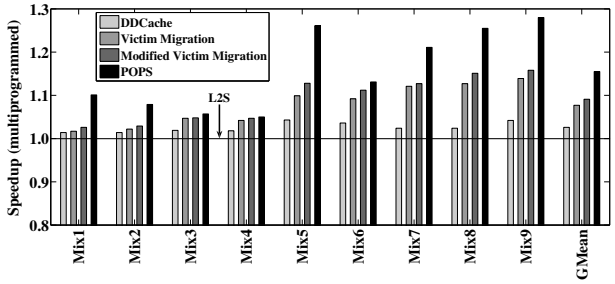

Figure 5. Performance improvement of POPS (normalized to L2S) along with DDCache, Victim Migration, and Modified Victim Migration for multiprogrammed workloads.

POPS also improves system performance via faster data availability by predicting a close-by keeper/sharer. Delegation avoids the need to communicate metadata to the home L1. POPS therefore improves performance significantly for multithreaded workloads. *MP3D*, *TSP*, *GraphMine*, and *Producer-Consumer* are a few that get most of the benefits from this delegation and direct L1-L1 communications through prediction. *Migratory*, *TSP*, and *MP3D* are applications that gain most from the migratory sharing pattern optimization where the requester is provided write permission proactively upon issuing read-shared requests. A significant improvement in *TSP* comes from a false-sharing optimization where cache lines are pinned down at the dominating writers' L1. *Ocean*, *Barnes*, and *Produce-consumer* benefit from the producer-consumer sharing pattern optimization. By adding extra metadata lines and decoupling metadata and data in each L2 set, POPS eliminates data copies at the home when the line is delegated and provides a larger effective L2 capacity. *Apache*, *JBB*, *Ocean*, and *Radix* are the multithreaded workloads that benefit most from this larger effective capacity.

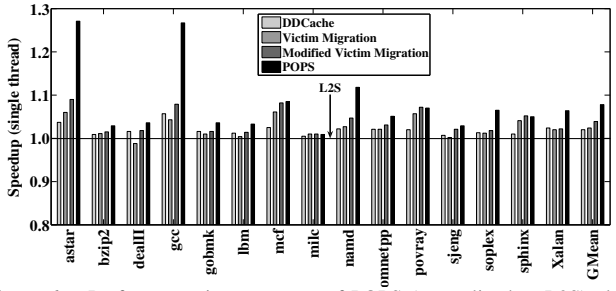

Figure 6. Performance improvement of POPS (normalized to L2S) along with DDCache, Victim Migration, and Modified Victim Migration for single-threaded workloads.

POPS improves the performance of multiprogrammed workloads via a larger effective capacity, faster data supply through delegation to the local L2, and reduced coherence communication (by delegation). Figure 5 reflects the performance improvement for multiprogrammed workloads. Since POPS delegates data in a controlled manner, the L2 is not overwhelmed by delegated lines. Additionally, the replacement policy at the L2 does not distinguish between home and delegated L2 lines. As in Modified VM, POPS improves single-threaded workload performance (Figure 6) due to local replication. Additionally, POPS reduces coherence communication for shared lines (from OS and shared library code and their data) as in DDCache.

In summary, POPS performs robustly across all types of workloads. In subsequent sections, we analyze different aspects of POPS such as L2 miss reduction, on-chip and off-chip traffic reduction, dynamic energy reduction, L2 size sensitivity, replica utilization, and storage overhead.

D. Effect on L2 Misses

Figure 7 shows L2 misses for the different protocols for multithreaded and multiprogrammed workloads, normalized to L2S. The L2 misses include all the misses in the system including those generated by OS and shared library code execution while executing the workloads. POPS reduces 20% of the L2 misses of L2S for multithreaded and 4% of the L2 misses of L2S for multiprogrammed workloads. DDCache exhibits behavior similar to POPS. However, VM and modified VM increases L2 misses on average.

The L2 miss rate is a function of cache capacity provided the

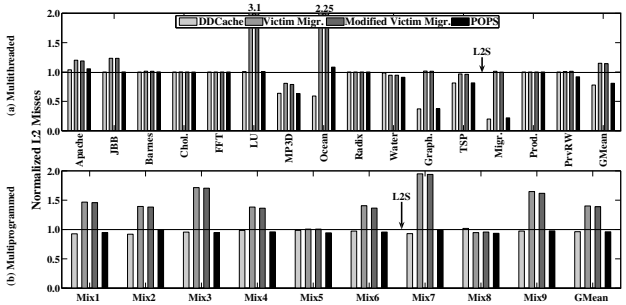

Figure 7. Comparative L2 misses (normalized to L2S) of DDCache, Victim Migration, Modified Victim Migration, and POPS for multithreaded and multiprogrammed workloads.

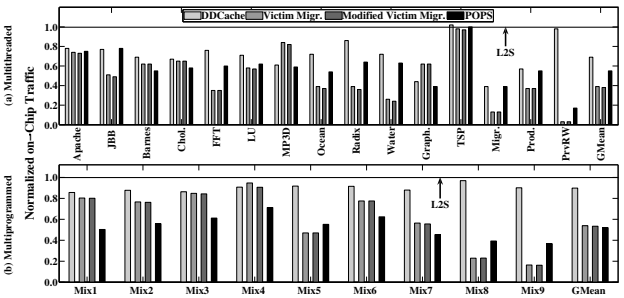

Figure 8. Comparative on-chip traffic generated (normalized to L2S) by DDCache, Victim Migration, Modified Victim Migration, and POPS for multithreaded and multiprogrammed workloads.

workloads and the replacement policies are the same, although 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. POPS has the advantage of extra effective cache capacity as POPS adds 16 metadata tags per L2 set, which can be used to link the data lines that are delegated to some L1s without storing the data at the L2, and so we see reduced L2 misses in POPS. DDCache also employs a similar technique to increase cache capacity by adding 4 extra metadata tags per L2 set. POPS has slightly higher L2 misses w.r.t. DDCache due to some replacement interference. Although there is at most one copy in the L2, moving the line to the local L2 interferes with the replacement policy for data for which the local L2 is the home. Both VM and modified VM use uncontrolled replication of evicted L1 data that is shared, which creates significant capacity pressure at the L2; we see that L2 misses increase, on average, by 15% for multithreaded and 40% for multiprogrammed workloads in the VM protocols.

The trend for L2 misses is that POPS exhibits slightly higher L2 misses than DDCache, but much lower L2 misses than L2S, VM, and modified VM. *PrvRW* is an exception to this trend. This is due to the fact that POPS is 2.8X faster than DDCache for *PrvRW*. The extra time under DDCache implies that more OS time-dependent code is executed, for which DDCache incurs extra misses. For VM and modified VM, *LU* and *Ocean* have very high L2 misses because of the fact that VM creates many replicas but only a few of them are used by the local core (see Section IV-G). Since VM preferentially evicts regular lines without any L1 sharer over replicas, this results in more misses.

E. Effect on on-Chip and off-Chip Traffic

On-chip and off-chip traffic are measured in terms of flit-hops, which is calculated by adding the number of hops traversed by all the flits in a control or data message. Traffic communicating between L1s and L2s are classified as on-chip and traffic communicating between L2s and memory are classified as off-chip traffic. Off-chip traffic is proportional to the number of L2 misses and L2 writebacks. Communication between L1 and local L2 slice is not counted toward network traffic since it does not traverse the network switch. Figure 8 shows on-chip network traffic for the different protocols for multithreaded and multiprogrammed workloads, normalized to L2S.

Since data messages contribute 5X-9X traffic compared to control messages, reduced data communication will directly translate into reduced overall traffic. POPS reduces both data and coherence communication for both private and shared cache lines by (1) delegating the coherence to an L1, (2) accessing shared data from close-by cores using prediction, (3) accessing private data from the local L2 slice when delegated, and (4) supporting access pattern-optimized communications. Due to these mechanisms, POPS reduces on-chip traffic, on average, by 45% for multithreaded workloads and by 48% for multiprogrammed workloads. DDCache reduces traffic for shared cache lines only. VM reduces on-chip traffic mainly by reducing data communication, since delegated data is transferred between local L1 and L2 without traversing the network switch.

Figure 9 shows off-chip network traffic normalized to L2S for the different protocols for multithreaded and multiprogrammed workloads. The off-chip traffic generation largely follows the L2 miss patterns. POPS reduces off-chip traffic, on average, by 21% for multithreaded workloads and by 6% for multiprogrammed workloads due to reduced L2 misses and reduced memory writebacks from L2. DDCache also reduces off-chip traffic but for VM and modified VM, off-chip traffic increases significantly due to increased L2 misses and increased evictions incurred due to the lower effective cache capacity.

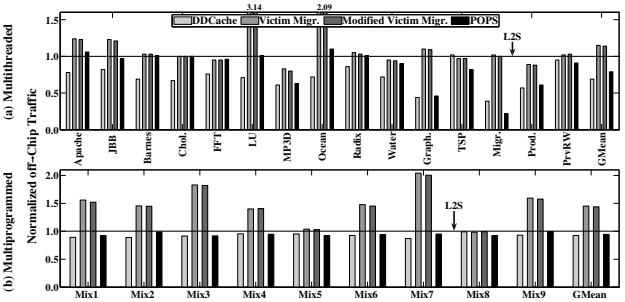

Figure 9. Comparative off-chip traffic generated (normalized to L2S) by DDCache, Victim Migration, Modified Victim Migration, and POPS for multithreaded and multiprogrammed workloads.

F. Effect on Dynamic Energy

Table III lists the energy consumption per access derived from Cacti 6.0 [26]. These numbers, along with collected access statistics, are used to derive dynamic energy numbers for each of the protocols.

Table III
DYNAMIC ENERGY CONSUMPTION VALUES PER ACCESS FOR INDIVIDUAL STRUCTURES IN POPS AND OTHER COMPARING PROTOCOLS USING 45NM TECHNOLOGY (VALUES ARE IN FEMTO-JOULES (FJ))

L1$		Predictor	Bloom fltr	L2$		Router/Interconnect			
Tag	Data	Access	Access	Tag	Data	BufRd	BufWr	Xbar	Arbiter
2688	16564	18593	9640	58299	76621	760	1187	24177	402

Figure 10 shows dynamic energy consumption normalized to

L2S for the different protocols for multithreaded and multiprogrammed workloads. These energy numbers only account for on-chip storage resources and do not include energy consumed for off-chip communication and DRAM accesses. Network energy consumption can be saved by reducing traffic generation (especially reducing data communication). Cache (L1 and L2) energy consumption can be reduced by reducing L2 accesses, which can be replaced by L1 accesses for shared data.

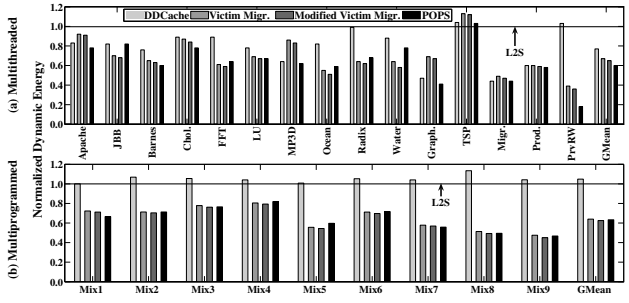

Figure 10. Comparative dynamic energy consumption (normalized to L2S) of DDCache, Victim Migration, Modified Victim Migration, and POPS for multithreaded and multiprogrammed workloads.

Dynamic energy consumed by POPS ranges between 18% and 103% of the dynamic energy consumed by L2S for multithreaded workloads and between 47% and 82% for multiprogrammed workloads with a suite-wise average of 60% for multithreaded workloads and 63% for multiprogrammed workloads. Since POPS reduces data and metadata communication by delegation, prediction, and localization of private data, and reduces coherence communication by delegation and access pattern optimization, we see significant reductions in energy consumption. DDCache also saves dynamic energy for multithreaded workloads but increases dynamic energy for multiprogrammed workloads. The VM protocols good energy reductions for both multithreaded and multiprogrammed workloads. Keep in mind, however, that their energy increase and performance loss comes from off-chip accesses, the energy for which is not included in Figure 10.

G. Replica Utilization

POPS uses a controlled mechanism to decide which evicted L1 lines to delegate to a local L2 slice. We have measured, over the execution, how many lines are delegated to the local L2, how many of those delegated lines are used by the local core, and how many requests from other cores are serviced by the local L2. We see that, on average, 57.4% of the delegated lines are used by the local core (100% would mean all the lines are used by the local core), while 8.5% of the delegated lines serve requests from other cores in POPS for multithreaded workloads; for multiprogrammed workloads, the numbers are 51.7% and 0.04% respectively.

Comparing POPS with VM in terms of replica creation and utilization, we see that VM creates almost 10X more replicas compared to POPS although their utilization percentages are similar to POPS. In VM, for multithreaded workloads, the local core uses 55.8% replicas and other cores are served by 10.6% of the replicas; for multiprogrammed workloads, the numbers are 59% and 0.4% respectively. Among multithreaded workloads that require L2 cache capacity, *LU* and *Ocean* have the least replica utilization, which results in more L2 misses.

H. Sensitivity Analysis of L2 Size

We analyze the sensitivity of protocol behavior to L2 cache sizes by varying the L2 cache size between 4 MB and 32 MB. Figure 11 shows the performance of the protocols normalized to L2S with the corresponding changed L2 size for multithreaded and multiprogrammed workloads. On average, POPS speedup over L2S ranges from 39% to 43% for multithreaded workloads and from 10% to 16% for multiprogrammed workloads. The general trend is that with the increase of L2 size, POPS performs better. This is due to the fact that POPS can keep more delegated lines that have been evicted from the L1 at the local L2 slice to reduce on-chip L2 access time. For some applications, a main benefit is the larger effective L2 cache size; this benefit reduces as the total L2 size increases (*e.g.*, *Radix*).

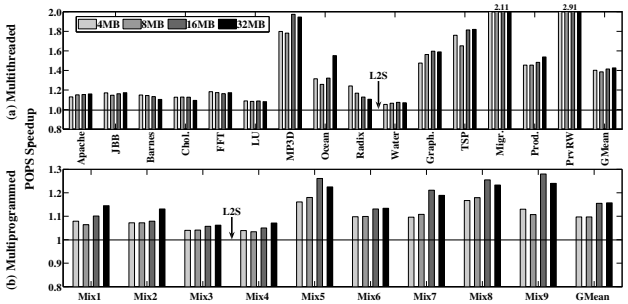

Figure 11. POPS's sensitivity to L2 size for multithreaded and multiprogrammed workloads. For each configuration, POPS performance is normalized to L2S with the correspondingly L2 size.

I. Storage Overhead

In Section III-A, we described the extra structures and fields added to implement POPS compared to L2S. We consider only the storage bits required for on-chip storage. POPS adds 16 metadata tags to each L2 set. It is worth noting that these extra metadata lines are smaller in size than regular metadata lines as they do not require a full sharers vector (N-bit) — instead, $log_2 N$ bits are sufficient to identify the node to which the line is delegated. POPS also uses a 1K 4-bit counter-based bloom filter, a 512-entry 8-way associative predictor table per node, and L1D tags augmented with 10 bits for sharing pattern detection as well as N bits for the sharers vector. Taking all extra storage into account, POPS requires 3.72% extra storage compared to L2S. In comparison, DDCache requires 1.22% extra storage, VM requires 5.24% extra storage, and Modified VM requires 5.28% extra storage.

V. Related Work

A large number of non-uniform cache designs with adaptive protocols have recently been proposed in order to reduce L1 miss penalties and reduce off-chip traffic [5], [7], [9], [10], [16], [17], [20], [27], [28], [32], [33]. We focus here on those most directly related to POPS.

ARMCO [16] 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 [17] improves on ARMCO to achieve larger effective cache capacity via decoupling of data and metadata and delegation of metadata ownership to an L1 cache. DDCache improves performance significantly

for multithreaded workloads but does not improve single-threaded and multiprogrammed workloads. Like DDCache, Direct Coherence [27], [28] also 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.

Victim Migration (VM [32]) tries to provide faster data availability by creating a replica at the local L2 slice. VM uses victim tags at the home L2 slice in order to eliminate data copies at the home, thereby mitigating some of the capacity pressure created by the replicas. Multiple replicas (upto $num_{cores} - 1$) may however be created for shared data. In contrast, POPS effectively increases LLC capacity by maintaining at most one data copy at either a local or home L2 (and can even have *no* data copy at the L2 if there are active sharers). Although VM has the potential to provide a performance boost, several shortcomings must be overcome. These include VM's need to access the local L2 slice to check for replicas, as well as the cache pressure due to the extra replicas.

Reactive-NUCA (RNUCA) [15] optimizes block placement for different access types through cooperation with the operating system. RNUCA places private data at the local L2 slice, and instruction and shared read-only data in a cluster formed by neighboring L2 slices; however, it leaves shared modified data placement unchanged, defaulting to the base L2S mapping. RNUCA utilizes rotational interleaving to lookup the nearest neighbor where the shared read-only data might reside. The RNUCA proposal operates at a page granularity to relocate private data to the local L2 bank. In comparison, POPS classifies data at a finer (line level) granularity, resulting in reduced misprediction. RNUCA requires changes in the OS, page table, and TLB while POPS is transparent to everything other than the coherence hardware. In addition, POPS also provides fine-grain sharing support (for both read-only and read-write shared data) via direct L1-L1 transfer of data.

CMP-NuRAPID [10] adapts to different sharing patterns by either replicating data at the L2 for read-shared data or performing in-situ communication for read-write or false shared data. CMP-NuRAPID also migrates data to L2 banks closer to the accessing cores. CMP-NuRAPID doubles L2 tag space and requires forward pointers (data group and frame number) and backward pointers (set index, tag, and processor id), which might have higher storage costs. In-situ communication requires a write-through L1 cache, which can increase both bandwidth and energy demands on the L1-L2 interconnect.

Adaptive Selective Replication (ASR [5]) enhances CMP-NuRAPID by controlling replication of read-shared data based on a cost/benefit analysis of increased misses versus reduced hit latency. ASR also creates LLC capacity pressure similar to VM due to the replication. ASR is optimized only for read-shared data. Private and read-write data default to the base L2S mapping.

Cooperative Caching [7] borrows concepts from software cooperative caching [3], [13], [30] to allow controlled sharing of essentially private caches, but requires a centralized coherence engine. Huh *et al.* [18] 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.* [12] 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.

Several proposals [1], [16], [24] use prediction to avoid indirection through a home/keeper in order to get to data faster. Some proposals [19], [21], [25] predict coherence actions and optimize coherence communication accordingly. Several coherence protocols that detect and optimize coherence actions for specific sharing patterns have been proposed in the past [8], [11], [29]. These protocols were leveraged for sharing pattern optimizations in ARMCO [16].

VI. Conclusions

We have developed the POPS cache design and protocol optimization that localizes data and coherence for *both* private *and* shared data through delegation and controlled local migration. POPS also provides a larger effective last level cache (LLC) capacity through decoupling, supports fine-grain sharing through L1-L1 direct accesses (no directory indirection) via prediction, and supports sharing and access pattern specific optimizations. The benefits of POPS include improvement in performance and reduction of energy and traffic. To the best of our knowledge, no other cache design supports such optimizations across all types of data and workloads.

Results on multiprogrammed and single-threaded workloads demonstrate that POPS is able to provide low latency access for multiprogrammed and single-threaded workloads, effectively treating the local L2 slice of a core as a large victim buffer. Results on multithreaded workloads show that POPS is also able to effectively localize communication for shared data between the sharers. POPS also provide a larger effective cache capacity by allowing the L1 and L2 caches to be non-inclusive, while using extra L2 cache tags to ensure coherence. Performance simulation on multithreaded and multiprogrammed workloads shows that POPS performs 42% better for multithreaded workloads, 16% better for multiprogrammed workloads, and 8% better when one single-threaded application is the only running process, compared to the base non-uniform shared L2 protocol. By lowering both on-chip and off-chip traffic, and reducing overall energy consumption, POPS presents a more complexity-effective design alternative to conventional shared L2 caches in a tiled architecture.

Acknowledgment

This work was supported in part by NSF grants CCF-0702505, CNS-0615139, CNS-0834451, CNS-0509270, CCF-1016902, 0747324, 0829915, and 0901701, and by NSFC grant 61028004.

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 *Proc. Int'l Conf. on Supercomputing*, pages 1–12, Nov. 2002.

[2] 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.

[3] 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.

[4] 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.

[5] B. Beckmann and M. Marty. ASR: Adaptive selective replication for CMP caches. In *Proc. Int'l Symp. on Microarchitecture*, pages 443–454, Dec. 2006.

[6] G. Buehrer, S. Parthasarathy, and Y. Chen. Adaptive parallel graph mining for CMP architectures. In *Proc. Int'l Conference on Data Mining*, pages 97–106, Dec. 2006.

[7] J. Chang and G. S. Sohi. Cooperative Caching for Chip Multiprocessors. In *Proc. Int'l Symp. on Computer Architecture*, pages 264–276, June 2006.

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

[9] Z. Chishti, M. Powell, and T. Vijaykumar. Distance associativity for high-performance energy-efficient non-uniform cache architectures. In *Proc. Int'l Symp. on Microarchitecture*, page 55, Dec. 2003.

[10] Z. Chishti, M. Powell, and T. Vijaykumar. Optimizing Replication, Communication, and Capacity Allocation in CMPs. In *Proc. Int'l Symp. on Computer Architecture*, pages 357–368, June 2005.

[11] A. Cox and R. Fowler. Adaptive cache coherency for detecting migratory shared data. In *Proc. Int'l Symp. on Computer Architecture*, pages 98–108, May 1993.

[12] N. Eisley, L. Peh, and L.Shang. In-network cache coherence. In *Proc. Int'l Symp. on Microarchitecture*, pages 321–332, Dec. 2006.

[13] M. J. Feeley et al. Implementing global memory management in a workstation cluster. In *15th ACM Symp. on Operating Systems Principles*, pages 201–212, Dec. 1995.

[14] N. Hardavellas et al. Database Servers on Chip Multiprocessors: Limitations and Opportunities. In *Proc. Conf. on Innovative Data Systems Research*, Jan. 2007.

[15] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamak. Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches. In *Proc. Int'l Symp. on Computer Architecture*, pages 3–14, June 2009.

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

[17] H. Hossain, S. Dwarkadas, and M. C. Huang. Ddcache: Decoupled and delegable cache data and metadata. In *Proc. Parallel Architectures and Compilation Techniques*, pages 227–236, Oct. 2009.

[18] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler. A NUCA substrate for flexible CMP cache sharing. In *Proc. Int'l Conference on Supercomputing*, pages 31–40, June 2005.

[19] S. Kaxiras and C. Young. Coherence communication prediction in shared-memory multiprocessors. In *Proc. 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 *Proc. Architectural Support for Programming Languages and Operating Systems*, pages 211–222, Oct. 2002.

[21] A. Lai and B. Falsafi. Memory sharing predictor: The key to a speculative coherent DSM. In *Proc. Int'l Symp. on Computer Architecture*, pages 172–183, May 1999.

[22] P. Magnusson et al. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.

[23] M. Martin et al. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *ACM SIGARCH Computer Architecture News*, 33(4):92–99, Sept. 2005.

[24] 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 *Proc. Int'l Symp. on Computer Architecture*, pages 206–217, June 2003.

[25] S. Mukherjee and M. Hill. Using prediction to accelerate coherence protocols. In *Proc. Int'l Symp. on Computer Architecture*, pages 179–190, June 1998.

[26] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing nuca organizations andwiring alternatives for large caches with cacti 6.0. In *Proc. Int'l Symp. on Microarchitecture*, pages 3–14, Dec. 2007.

[27] A. Ros, M. E. Acacio, and J. M. Garcia. Direct coherence: Bringing together performance and scalability in shared-memory multiprocessors. In *Proc. Int'l Conf. on High Performance Computing*, pages 147–160, Dec. 2007.

[28] A. Ros, M. E. Acacio, and J. M. Garcia. DiCo-CMP: Efficient cache coherency in tiled CMP architectures. In *Int'l Symp. on Parallel and Distributed Processing*, pages 1–11, Apr. 2008.

[29] P. Stenström, M. Brorsson, and L. Sandberg. An adaptive cache coherence protocol optimized for migratory sharing. In *Proc. Int'l Symp. on Computer Architecture*, pages 109–118, May 1993.

[30] 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 *Proc. Symp. on Operating Systems Principles*, pages 16–31, Dec. 1999.

[31] 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 *Proc. Int'l Symp. on Microarchitecture*, pages 24–36, June 1995.

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

[33] M. Zhang and K. Asanovic. Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors. In *Proc. Int'l Symp. on Microarchitecture*, pages 336–345, June 2005.