Improving Support for Locality and Fine-Grain Sharing in Chip Multiprocessors*

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

ABSTRACT

Both commercial and scientific workloads benefit from concurrency and exhibit data sharing across threads/processes. The resulting sharing patterns are often fine-grain, with the modified cache lines still residing in the writer's primary cache when accessed. Chip multiprocessors present an opportunity to optimize for fine-grain sharing using direct access to remote processor components through low-latency on-chip interconnects. In this paper, we present Adaptive Replication, Migration, and producer-Consumer Optimization (ARMCO), a coherence protocol that, to the best of our knowledge, is the first to exploit direct access to the L1 caches of remote processors (rather than via coherence mechanisms) in order to support fine-grain sharing.

Our goal is to provide support for tightly coupled sharing by recognizing and adapting to common sharing patterns such as migratory, producer-consumer, multiple-reader, and multiple read-write. The protocol places data close to where it is most needed and leverages direct access when following conventional coherence actions proves wasteful. Via targeted optimizations for each of these access patterns, our proposed protocol is able to reduce the average access latency and increase the effective cache capacity at the L1 level with on-chip storage overhead as low as 0.38%. Full-system simulations of 16-processor CMPs show an average (geometric mean) speedup of 1.13 (ranging from 1.04 to 2.26) for 12 commercial, scientific, and mining workloads, with an average of 1.18 if we include 2 microbenchmarks. ARMCO also reduces the on-chip bandwidth requirements and dynamic energy (power) consumption by an average of 33.3% and 31.2% (20.2%) respectively. By evaluating optimizations at both the L1 and the L2 level, we demonstrate that when considering performance, optimization at the L1 level is more effective at supporting fine-grain sharing than that at the L2 level.

Categories and Subject Descriptors: B.3.2 [Memory Structures]: Design Styles—Cache memories; Shared memory C.1.2 [Processor Architectures]: Parallel processors

General Terms: Design, Performance

Keywords: Cache Coherence, Chip Multiprocessors, Fine-Grain Sharing, L1-to-L1 Direct Access, ARMCO

1. INTRODUCTION

CMOS scaling trends allow increasing numbers of transistors on a chip. In order to take advantage of the trend while staying within power budgets, processor designers are increasingly turning toward multi-core architectures — often chip multiprocessors (CMPs) of simultaneous multithreaded (SMT) cores [21, 26, 27]. While these initial multi-core efforts provide a limited number of cores and contexts, future processors would allow hundreds of simultaneously executing threads [31, 32]. If this computing power is to be applied to conventional workloads, previously sequential applications will need to be rewritten as fine-grain parallel code. To support such fine-grain code, it will be increasingly imperative to devote chip real estate to mechanisms that facilitate data communication and synchronization.

Several recent proposals [4, 6, 8, 37] have examined enhancements to a non-uniform level-2 (L2) cache architecture both for improved locality and sharing. Victim replication [37] proposes a shared L2 cache (distributed among the tiles of a CMP) as the base design but replicates lines evicted from the level-1 (L1) to the local L2 bank. Private L2 tags and shared data is another option used in CMP-NuRAPID [8]. Their design replicates data based on application access patterns, i.e., repeated accesses to the same cache line by a processor result in a replica in the closest cache bank. They also use in-situ communication for read-write shared data by pinning down the cache line at one location when this access pattern is detected. Adaptive selective replication [4] improves on the above protocol by controlling replication based on a cost/benefit analysis of increased misses versus reduced hit latency. Cooperative caching [6] starts with private L2 banks as the base design and attempts to increase the effective capacity by cooperatively keeping cache lines in other cores' L2s. However, all of these efforts require communication through the L1-L2 hierarchy in the presence of fine-grain communication.

Prior research [7, 13, 18, 19, 29, 33] has also demonstrated the benefits of protocols that can detect and adapt to an application's sharing patterns for specific cache lines. Chip multiprocessors present an additional unique opportunity for direct access to remote processor components through low-latency on-chip interconnects. In this paper, we explore direct access to the L1 caches of remote processors (rather than via coherence mechanisms). Similar to

*This work was supported in part by NSF grants CCF-0702505, CNS-0411127, CNS-0615139, CNS-0719790, CCF-0747324, and CNS-0509270; NIH grants 5 R21 GM079259-02 and 1 R21 HG004648-01; and an IBM Faculty Partnership Award.

PACT'08, October 25–29, 2008, Toronto, Ontario, Canada.

most of the L2-level proposals, we use a non-uniform-shared L2 (L2S) as our base design point in order to maximize capacity. Our goal is to provide support for tightly coupled sharing by recognizing and adapting to common sharing patterns such as migratory, producer-consumer, multiple reader, and multiple read-write, while at the same time addressing locality and capacity issues.

We present Adaptive Replication, Migration, and producer-Consumer Optimization (ARMCO), a protocol that adaptively optimizes data communication for migratory, producer-consumer, multiple-readers, multiple-writers, and false-shared data via hardware mechanisms. ARMCO uses a predictor table at the L1 level to predict the closest L1 containing the requested cache line, thereby reducing expensive L2 data accesses by getting the data directly from the predicted L1. Last reader and last writer ID tags per L1D cache line help determine the access pattern of the cache line. This information is used by the controller in determining the action that will minimize overall communication in the critical path of the application. One new state, migratory (MG), is added to the base MSI/MESI protocol to switch between migrate-on-read and replicate-on-read. The use of distributed logic and low storage overhead (0.38% of on-chip storage bits for our design) allows ARMCO to scale as the number of cores is increased.

In summary, the contributions of ARMCO are:

- high accuracy in predicting adjacent L1 for missed shared data with low storage overhead (0.38% of on-chip storage bits).

- lower latency, reduced bandwidth requirements, and reduced on-chip energy (and power) consumption in the memory hierarchy due to improved locality of access.

- adaptive detection of migratory, producer-consumer, multiple readers, and multiple writers data.

- adaptive switching between migrate-on-read and replicate-on-read for migratory and non-migratory data.

- in-place writes for multiple writer/false-write-shared data.

Our results on a 16-core CMP with 64KB L1 split-cache and 16MB L2 cache (in 16 different banks) show performance speedup over L2S ranging from 1.04 to 2.26. For the 12 commercial [12, 16], scientific [35], mining [5], and branch-and-bound benchmarks, average (geometric mean) speedup of ARMCO is 1.13 (1.18 including the 2 micro benchmarks) over L2S. ARMCO is able to reduce interconnect network packets by 33.3% on average, dynamic energy consumption in the on-chip memory hierarchy by 31.2% (and dynamic power consumption by 20.2%), and shows better scalability than the base L2S. We also compare ARMCO to one example adaptive protocol at the L2 level — victim replication [37]. Our results demonstrate that when considering performance, optimization at the L1 level shows better promise in supporting fine-grain sharing than that at the L2 level.

2. DESIGN OVERVIEW

Caches are very effective at exploiting memory access locality. However, latency and capacity tradeoffs, in addition to active sharing among processors or cores, limit their effectiveness. Figure 1 presents a high-level view of the multicore architecture we base our design on. Our baseline platform is a CMP with distributed nodes interconnected by a general-purpose network. Each node contains a processor core, a private L1 (both I and D) cache, and a slice of the globally-shared L2 cache. The L1 caches are dual-ported in order to reduce interference with the processor-L1 path for both L2S and ARMCO. Coherence at the L1 level is maintained using an invalidation-based protocol. Directory entries are maintained at the corresponding L2 bank. The baseline depicted (and representative of our implementation) uses a switched mesh interconnect, although other interconnects are also possible. Caches communicate with the memory controller using a hierarchy of switches.

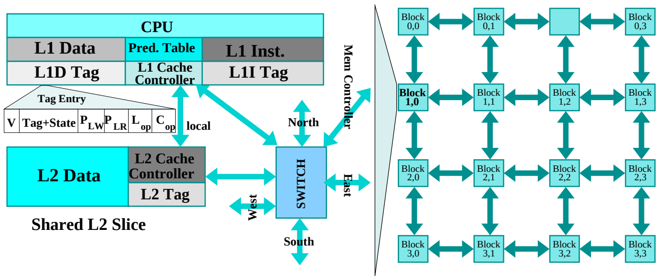

Figure 1. Schematic of the underlying CMP architecture depicting a processor with 16 cores.

As can be seen from the layout, the L1 caches of neighboring processors are sometimes closer and have lower access latency than a potentially remote slice of the L2 cache to which a particular cache line maps. We propose to take advantage of the proximity and direct access to L1 caches in order to provide low-latency fine-grain communication among processes. Direct L1-to-L1 communication uses the same interconnect as the L1-to-L2 communication, thereby avoiding any additional penalties or overheads associated with separate links and avoiding interference with the processor-to-L1 path.

ARMCO leverages direct access to the remote L1s to facilitate low-latency fine-grain communications among simultaneously executing processes. Several access patterns are recognized, predicted, and optimized for:

- Producer-consumer: This type of access pattern usually involves a single producer with one or more consumers. Traditional directory-based protocols require communication via the directory in order to bring a copy of the cache line into the local L1 cache. Subsequent invalidation by the producer also requires a trip through the directory. ARMCO avoids the expensive 3-hop access through the directory by directly accessing the remote L1 and avoiding making a copy.

- Migratory: Data elements such as reduction variables are often accessed with a unique pattern. The data is read and then modified in quick succession. In the common invalidation-based protocols, this often means that the accessing processor initiates two separate requests (resulting in accessing the remotely modified copy twice) to obtain first read and then write permission. ARMCO recognizes the migratory pattern in a manner similar to that in [13, 33], thereby avoiding the multiple remote accesses.

- Multiple read/write: This type of data is possible in applications where the logical organization of the data structures and the natural parallelization strategy are in conflict, resulting in temporary *false sharing*. In a normal coherence protocol, the cache line bounces between L1 caches, each time requiring a trip to the L2/directory in order to maintain coherence. ARMCO avoids this *ping-pong* by reading and writing the data in place.

- Multiple reader: For data that is read without modification by multiple processors, ARMCO defaults to the underlying coherence behavior of replication but with faster cache-line availability from close-by L1s via location prediction.

In order to identify the sharing patterns and locate copies of the cache lines in remote processors, ARMCO utilizes prediction mechanisms at the local L1 along with extensions to the cache line tag to capture its access behavior. To minimize intrusion, the added logic is off the L1 cache's critical cache-hit path and only affects how a miss is handled. In the following section, we describe the details of the on-chip memory hierarchy design that facilitates low-latency fine-grain communication among simultaneously executing threads/processes.

3. PROTOCOL AND ARCHITECTURAL SUPPORT

3.1 ARMCO Predictor and Tag Structures

In order to predict data location and type of access pattern, ARMCO uses the following structures and states along with traditional cache structures to track the accesses from different processors to guide the decision of data migration or replication:

- *Access History Tags*: $2 \times log_2(P)$-bits are used to track the last reader (P_{LR}) and the last writer (P_{LW}) of the corresponding cache line, with P being the number of processors in the CMP. A single bit (L_{op}) is used to indicate whether the last access is a read or a write. Finally, whenever the local processor accesses the cache line consecutively without an intervening access from another processor, we set a C_{op} bit. This bit is preserved in the predictor table (see below) when the line is evicted and is used in order to decide whether to perform in-place access or request a full cache line from a neighbor.
- *Migratory State*: Inspired by the techniques proposed by [13, 33], we add one additional state, called Migratory state (MG), to the base MSI coherence protocol for L1 caches (for a total of four stable states) to keep track of whether the cache line is in migratory state. We use the access history tags to adapt in and out of migratory state.
- *Predictor Table*: Each processor has a predictor table to identify a potential remote L1 with a cached copy of the line in case of a miss at the local L1. Each predictor entry has a valid bit, cache-line tag bits, $log_2(P)$-bits for remote L1 id, and the same C_{op} bit as in the access history tag in the L1 cache.

3.2 ARMCO Protocol Actions

ARMCO state transition diagram: Table 1 shows the state transitions at the L1 level for the ARMCO protocol. The subsequent subsections describe the details for choosing different actions on different situations.

Local L1 cache hit: On a local L1 data cache look-up by the processor, the cache tag and the predictor table are looked up in parallel. A read or write hit in the L1 cache (whether by the local or a remote processor) results in the last reader or writer field (P_{LR} or P_{LW}) being updated accordingly.

Request	State	Pred. Table	Cache Message	InPlace Rd/Wr	Migration	Next State
LD(ST)	I	Miss	Req to Dir/L2	-	-	S(M)
LD(ST)	I	Hit	L1-L1 LD(ST)	N	N	S(M)
LD(ST)	I	Hit	L1-L1 LD(ST)	Y	N	I
LD(ST)	I	Hit	L1-L1 LD(ST)	N	Y	MG
LD	S	-	-	-	-	S
ST	S	-	Req to Dir/L2	-	N(Y)	M(MG)
LD or ST	M/MG	-	-	-	-	M/MG
L1-L1 LD(ST)	I	-	Fwd to Dir/L2	-	-	I
L1-L1 LD	S	-	Data	-	-	S
L1-L1 ST	S	-	Fwd to Dir/L2	-	-	S
L1-L1 LD	M/MG	-	Data	N(Y)	N	S(M)
L1-L1 LD	M/MG	-	Data	N	Y	I
L1-L1 ST	M/MG	-	Data	N	-	I
L1-L1 ST	M/MG	-	InPlaceWr	Y	N	M
INV	S/M/MG	-	-	-	-	I
Downgrade	M/MG	-	-	-	-	S

Table 1. State transition table at the L1 level for ARMCO. '-' indicates invalid or don't care. InPlace Rd/Wr and Migration can never be Y at the same time. LD(ST) are load (store) requests at the local L1 cache and L1-L1 LD(ST) are L1-to-L1 requests from the predicting L1 to the predicted L1 due to a hit in the predictor table on a cache miss. Data (response) might be full cache line if InPlace is N or required bytes (atmost 8B) if InPlace is Y.

Local L1 cache miss; predictor table miss: If no match is found either in the tag or the predictor table or the requested address is mapped to the local L2 cache bank, the request is sent to the associated L2 cache bank.

Local L1 cache miss; predictor table hit: An L1 cache miss (with the address mapped to a non-local L2 cache bank) with a hit in the predictor table results in a request being sent to the predicted remote L1 cache. An incorrect prediction will result in the request being forwarded to the proper L2 bank. Actions on a correct location prediction are a function of the access pattern.

3.3 Recognizing and Handling Access Patterns

The last reader/writer (P_{LR}, P_{LW}), last access (L_{op}) and multiple access (C_{op}) fields along with the additional stable state per cache line are used for access pattern identification. We first focus on cache lines that are in modified state in some L1 cache. To facilitate access pattern tracking, the directory forwards a read/write request from another processor to the current owner. The owner, with access pattern information, will handle the request in the most appropriate way and notify the directory for proper bookkeeping — when it can. When it cannot service the request, such as when the ownership is being or has been transferred to another processor or back to the directory, the request is forwarded to the directory on behalf of the requester as a regular request. No negative acknowledgment is used in communications.

Migratory data: When a remote access (say from P_2) arrives at an L1 (say at P_1) for a cache line in modified state (M), P_{LR} and P_{LW} are looked up. A writer that finds a matching P_{LR} (=P_2) indicates the start of migratory behavior. A reader that finds a matching P_{LW} with owner(=P_1) in cache state *MG* indicates the continuation of migratory behavior. We supply the cache line and the tracking tags (P_{LR}, P_{LW}, and L_{op}) to the requester (P_2) and invalidate the line from the original owner (P_1's cache). This allows us to quickly supply the data and exclusive permission (in *MG* state) to the next requester (say P_3) upon only a read request. In the steady state, migration takes only one transaction between successive owners.

In some cases, when the requester P_2 reads more than once before writing to the data, the cache controller of the current owner (P_1) will transfer the cache line to P_2 upon the second read request and therefore will not detect the migratory access pattern. Here, we use the help of the directory. When P_2 sends an upgrade request for exclusive access, if there is only one other sharer to invalidate, the directory piggybacks the ID of that sharer. If this ID matches with P_{LW} at P_2, then the cache line starts migratory behavior and the state is set to MG rather than M.

Since the migratory state is "sticky", if the data is no longer accessed in a migratory pattern, we need to revert the line back to normal (shared). If a migratory cache line is requested by another processor before the current owner had a chance to write to it, then the line loses the migratory property and is downgraded to a shared line or invalidated depending on whether the remote request is a read or a write. In the migratory state, P_{LW} is used to determine whether the current owner actually modified the cache line. In our design, the directory does not make the distinction between migratory and modified cache lines. When a migratory line is evicted and written back, the migratory state is lost and has to be re-learned the next time around. Of course, another option would be to add a stable state in the directory to avoid the relearning.

Multiple Read-Write: When a cache line is being read by some processors and written to by others in an intermingled manner (whether due to true communication or false sharing), remote access is unavoidable. Sometimes when the accesses from different processors are finely meshed, bouncing the cache line back and forth only wastes time and energy. We try to pin down the data when called for. In general, if the the last reader (P_{LR}) and writer (P_{LW}) keep switching from one processor to another, we keep the cache line in the current node and service consumers' requests via in-place read and accept in-place writes from other producers. When one particular remote node generates back-to-back requests indicating a stronger locality, then the cache line will be transferred there. For example, upon a read request from a remote processor (P_2), if P_{LR} or P_{LW} is P_2, the cache line is replicated at P_2. Two back-to-back writes from P_2 will move the ownership to P_2 (M state), while a read followed by a write will also switch the cache line into migratory mode (MG state). If none of the above conditions are met, the request is serviced in-place, at the current L1.

With this policy, the first miss is in general serviced via in-place read or write at a remote node. This would result in two remote accesses when a processor makes a series of accesses. We attempt to anticipate such series of accesses and bring in the cache line upon the first access. This is done with the help of the C_{op} bit in the cache and the predictor table. If a processor makes consecutive accesses to a cache line, the C_{op} bit of the line will be set in the cache and copied to the predictor table upon the eviction of the line. The next time we follow the prediction table to fetch the same line, if C_{op} is set, we will avoid an in-place access and issue a line-fetch request instead.

Multiple reader: With direct access to remote L1 caches, access latency may be improved even for read operations. Specifically, to service a cache miss, we can fetch the cache line from a nearby L1 cache instead of from the home L2 cache bank. This can cut down the latency of the cache miss as the directory may be further away and accessing a slower and physically larger L2 partition can take more cycles than accessing the smaller, faster L1 cache. The process is illustrated in Figure 2-(a).

Note that if the line is in dirty state, we will generally satisfy a remote request via an in-place access as discussed earlier. When the same remote node generates two back-to-back read requests with no intervening read or write from any other processor, the entire line will be supplied and the current processor will downgrade the line to shared state. Table 2 summarizes the conditions for migration and/or replication in the presence of back-to-back accesses by the same remote processor.

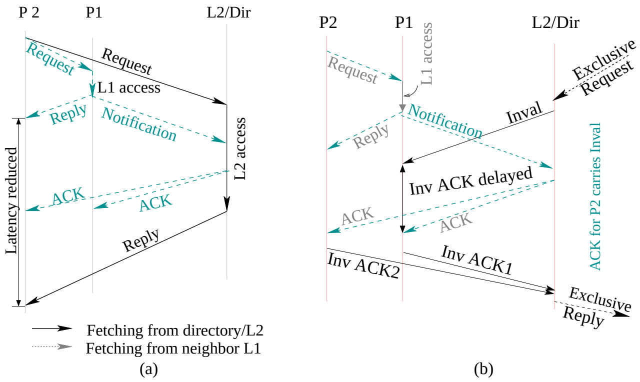

Figure 2. Illustration of messages and activities of fetching data from nearby neighbor, in contrast to fetching from the directory.

On the other hand, if the cache line is in shared state in the current processor, a remote read request would replicate the line without any hysteresis (i.e., without checking P_{LR}). Note that an L1 cache that only has a shared copy of a line can not service a remote write request to that line and treats the request as a destination misprediction.

Access	Note	Reaction
R_iR_i/ W_iR_i	Read/Write followed by read from same processor	Replicate line to requester; downgrade local copy to shared
R_iW_i	Read followed by write: migratory pattern	Migrate line in MG state; invalidate local copy
W_iW_i	Producer i updates more frequently	Transfer line in exclusive (M) state to i; invalidate local copy

Table 2. Summary of possible options with multiple back-to-back accesses by the same remote processor to a dirty line where first access is an in-place access. R_i and W_i stand for read and write access from the remote processor i, respectively.

3.4 Protocol Implementation Issues

Whenever an L1 cache (of P_1) responds to a remote request (from P_2), it is temporarily serving the role of the directory. Clearly, any change to the ownership and sharer list needs to be reflected in the directory. This is done by the L1 cache controller of P_1 sending a *notification* to update the directory (sometimes with the write-back data). Note that when an in-place read is performed, the directory generally does not need to be updated, unless the processor core issues memory operations in a speculative, out-of-order fashion and relies on invalidations to maintain a restrictive consistency model [36].

When a notification to the directory is sent as a result of an L1-to-L1 data transfer, the cache line would enter a transient state in both P_1 and P_2's caches – for different reasons – until the directory acknowledges the notification to both processors.

When P_1 supplies a cache line to P_2, before the notification arrives at the directory, it is possible for an independent exclusive access request to the same cache line from P_3 to arrive at the directory and result in invalidations sent from the directory. To ensure the directory only replies to P_3 after it has also invalidated P_2's copy, P_1 will not respond to the invalidation request until the directory has processed and acknowledged the notification P_1 sent (Figure 2-(b)). On the other hand, for P_2, the cache line is also set to a transient state before P_2 receives the directory's acknowledgment. While in this transient state, P_2 will delay supplying the cache line to another L1 cache. This is done in order to achieve ordering with simplicity.

3.5 Predicting Destinations

Clearly, the benefit of direct data access from a remote L1 will be greatly limited if we need indirection from the directory for each access. Thus, predicting the location of an L1 cache having the needed cache line becomes a necessary component. We use a set-associative predictor table per node to make a best-effort attempt to track the whereabouts of cache lines. For instance, when an L1 controller transfers the ownership of a line to another L1 cache, we remember the destination node as the new location for that line. Conversely, when an L1 controller receives a cache line in M or MG state, the entry for that line is invalidated in the prediction table. Given a directory-based protocol, we have limited knowledge about data migration happening elsewhere in the system. Nevertheless, when the L2 directory controller sends out any request or acknowledgment to an L1 cache, it can always piggyback the location of the current exclusive owner or the closest neighbor having a shared copy. This includes:

- When the directory sends a data reply to an L1 controller in shared mode;
- When the directory sends out invalidation messages;
- When the directory downgrades an L1's cache line; and
- When the directory sends an acknowledgment of a write-back, or the acknowledgment of the eviction notification of a shared line (in systems that do that).

The predictor table is also updated by utilizing the messages sent between L1 controllers, including:

- Direct L1-to-L1 requests; the predicted processor PP_{id} field in the predictor table entry is set to the requester if: (1) the requester is served by giving up ownership, (2) the request was for exclusive access, or (3) if the requester was served by a shared copy and the requester is closer than the current PP_{id} if there is one.
- Predictor table entry invalidation requests; when a cache line is invalidated or evicted, a predictor table entry invalidation request is sent to P_{LR} and/or P_{LW} if those are not the local processor. Seeing this invalidation request, the L1 controller invalidates the corresponding predictor table entry if PP_{id} matches the requester.

Of course, the predictor can not always be correct and when an L1 cache controller sends a request to a node which no longer has the data, the receiver forwards the request to the directory. If the protocol supports silent drop of a shared line, then this forwarded request also implies the absence of the cache line, which the directory can utilize. Note that we do not need to proactively notify the requester of the misprediction, as when the directory replies, updated information about which neighbor has a copy will be piggybacked. Finally, if the predicted node is far away, the cost of misprediction is high, especially when the directory is close-by. Therefore, it is wise to limit such predictions to close-by neighbors. In this paper, we only consider nodes within 2 hops and do not predict for a request whose home node is the local L2 slice.

Storage overhead: As discussed in Section 3.1, each processor needs a predictor table as well as extra bits in the L1 cache lines. This hardware storage cost is very low. In a 16-processor CMP (our baseline), each L1 cache line needs 10 extra bits. Each predictor table entry takes 25 bits, which can be further reduced by storing only partial cache tags. The number of entries in the predictor table need not be large. We experimented with a range of predictor table sizes and associativity and found that a 1024-entry 8-way associative table provides reasonable performance while meeting access latency constraints. In our baseline system, this configuration brings the total storage overhead of ARMCO to 70 KB, or about 0.38% of the total on-chip cache storage. The storage overhead is compared with other previous related proposals [4, 8, 13, 19, 33, 37] in Section 5.9.

4. RELATED WORK

Most previous multi-core cache designs have assumed either a shared L1 data cache (e.g., SMTs) or L1 caches that are private (and local) to each individual cluster/core (e.g., CMPs) with coherence maintained across the L1s at the L2 level. Cache architectures have been proposed that use non-uniform access latency to reduce the access time of the L2 cache [9, 20] for single-threaded workloads. More recently, there has been a focus on chip multiprocessor L2 cache designs [4, 6, 8, 37].

Zhang and Asanovic [37] assume an L2 cache that is distributed among the tiles of a CMP and propose victim replication (VR) at the L2 level in order to reduce the L1 miss penalty. The entire L2 is shared among all processors, with replication of the victims from the primary cache in each local L2 slice. In effect, coherence is maintained at both L1 and L2 levels. They do not study the effects on different sharing patterns. In addition, they present only the effects on memory latency and not the effects on overall performance. By consuming multiple L2 cache blocks for a single data line, VR creates more pressure on the on-chip memory hierarchy. CMP-NuRAPID [8] identifies the need to address both L1 miss latency and read-write sharing. Repeated accesses to the same cache line by a processor result in a replica in the closest cache bank. Reads and writes by multiple processors to the same cache line result in "in-situ" communication (an optimization requiring a write-through L1). Once the "in-situ" access pattern is recognized, the location of the cache line is fixed, resulting in non-ideal latencies for a processor that might be the dominant accessor in case of poor placement. Moreover, CMP-NuRAPID also doubles the L2 tag space required, hindering scalability. Adaptive selective replication [4] improves on the above protocol by controlling replication based on a cost/benefit analysis of increased misses versus reduced hit latency. Cooperative caching [6] borrows concepts from software cooperative caching [2, 15, 34] to effectively increase the capacity of essentially private caches through controlled cooperation but it suffers a scalability bottleneck due to the need for a centralized coherence engine. Eisley et al. [14] describe an in-network coherence protocol that leverages a network-embedded directory to get data directly from a sharer/owner if found on the way to the home, thereby reducing access latency. Set-up/tear-down of the tree, however, can make overall latency highly variable due to potential deadlock recovery.

The above designs work at the L2 level. While capacity and access latency issues addressed at the L2 level are an important problem and better addressed at the L2 level, they are orthogonal and complementary to the design of the L1 cache: the tight coupling of the L1 with the processor allows for optimizations to improve fine-grain sharing and synchronization that involve active read-write sharing. This paper extends our earlier design that exploits direct access to the L1 cache [17] via a protocol that does not rely on broadcast or on additional interconnects or ports to the L1 cache.

There has also been considerable research in adaptive coherence protocols that choose actions (such as invalidation versus update) based on the sharing patterns observed (e.g., adaptive migratory sharing [13, 33] or producer-consumer patterns [7]). Our migratory extensions borrow ideas from the work of Cox and Fowler [13].

Techniques have also been proposed to predict the coherence state of a cache block in order to hide the latency of the coherence mechanism in shared-memory multiprocessor systems [18,19,22, 24, 29]. Mukherjee and Hill [29] used an extension of Yeh and Patt's two-level PAp branch predictor to predict the source and type of the next coherence message by using address-based prediction. Martin et. al. have proposed destination set prediction, a variant of multicast snooping, which tries to optimize bandwidth/latency with respect to broadcast snooping and directory protocols by being in the middle of these two extreme cases.

Lai and Falsafi [22] used a pattern-based predictor to predict the next coherence messages. Kaxiras and Goodman [18] proposed instruction-based prediction to reduce the higher overhead of address-based prediction. Kaxiras and Young [19] explored the design space of prediction mechanisms in SMP machines for predicting coherence messages. All these approaches aim to hide the long latencies of fetching data from a remote cache/memory. Such predictive mechanisms could be combined with our adaptive protocol for improved performance. While instruction-based prediction was proposed to reduce the higher storage overhead of address-based predictors, our address-based predictor already shows low overhead with good performance.

5. PERFORMANCE EVALUATION

5.1 Evaluation Framework

To evaluate ARMCO, we use a Simics-based [23] full-system execution-driven simulator, which models the SPARC architecture. We use Ruby from the GEMS toolset [25], modified to encode ARMCO's requirements, for cache memory simulation. We simulate a 16-way chip multiprocessor (CMP) with private split L1 instruction and data caches and a 16-way banked shared L2 as the base system for our evaluation. The baseline coherence protocol is a non-uniform-shared L2 (L2S) MESI-style directory-based protocol[1].

Each processor has one L2 bank very close to it. A 4x4 mesh interconnect is used to connect the 16 L1 controllers, the 16 L2 controllers for the 16 L2 banks, and one memory controller. Each interconnect switch is connected to the four adjacent switches in the mesh in addition to the local L1 controller, local L2 controller, and the memory controller. L1 and the corresponding L2 bank are directly connected (schematic diagram shown in Figure 1). We use Cacti 6.0 [30] to derive the access times and energy/power parameters for the predictor table, different levels of caches, and interconnects. We employ virtual cut-through switching for transferring cache messages through the interconnect. We use GEMS's [25] network model for interconnect and switch contention modeling, using the parameters in Table 3. We encode all stable and transient states and all required messages for a detailed network model simulation of ARMCO and L2S using SLICC [25].

16-way CMP, Private L1, Shared L2	
Processor cores	16 3.0GHz in-order, single issue, non-memory IPC=1
L1 (I and D) cache	64KB 2-way each, 64-byte blocks, 2-cycle
Predictor table	1K entry 8-way associative
L2 cache	16MB, 16-way unified, 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 (sensitivity analysis for 2, 4,and 6 cycles link latencies), virtual cut-through routing

Table 3. Processor, cache/memory, and interconnection parameters

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 [3] request generator and SPECjbb2005. Alameldeen et al. [1] described these commercial workloads for simulation. As scientific benchmarks, we have a large set of applications and kernels from the SPLASH2/SPLASH suites [35], which includes Barnes, Cholesky, FFT, LU, MP3D, Ocean, Radix, and Water. Our benchmark suite also includes a graph mining application [5] and a branch-and-bound based implementation of the non-polynomial (NP) traveling salesman problem (TSP). All these applications are thread-based except Apache, which is process-based. Table 4 lists the problem sizes, access patterns, and L1 miss rates for L2S at 16 processors.

5.2 Estimating Expected Performance Speedup

ARMCO improves performance via several optimizations that reduce the number of network packets and hops, and thereby the overall latency. Direct L1 accesses occur for: (1) replication from shared state, (2) replication from modified state, (3) migration from modified (M) state, (4) migration from migratory (MG) state, and (5) in-place-read/write. For each of these scenarios, the ARMCO cost (latency) is

$$c = 2 * d_{\mathcal{R},\mathcal{P}}(L_{link} + L_{sw}) + 4 * L_{link} + 2 * L_{sw} + L_{L1}$$

where $d_{\mathcal{R},\mathcal{P}}$ is the distance between the requester ($\mathcal{R}$) and the predicted processor/owner ($\mathcal{P}$), and L_{link}, L_{sw}, and L_{L1} are the latencies for the link, switch, and the L1 cache, respectively.

The total benefit can be calculated by the formula

$$B = \sum_i n_i * b_i - n_{mp} * P_{mp}$$

[1] Previous proposals [4, 6, 8, 37] have used both shared and private caches for comparison purposes. In [8], the performance improvement options for non-uniform-shared, private, and ideal caches (capacity advantage of shared cache and latency advantage of private cache) are explored with respect to a conventional uniform-shared cache. Their results show that non-uniform-shared and private caches are close to each other with respect to performance and better than a uniform-shared cache. We therefore use a *non-uniform-shared* cache as the baseline for our comparison

Benchmark	Simulated problem size	Major data access pattern	L1 miss rate (L2S)
Apache	80000 requests fastforward, 2000 warmup, and 3000 for data collection	read-shared,read-write-shared	11.2%
SPECjbb2005	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	single producer-consumer	6.9%
Radix	550K 20-bit integers, radix 1024	read-shared, producer-consumer	3.2%
Water	512 molecules; run-to-completion	read-shared, migratory	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%
Producer-consumer	2K shared cache lines and 8K private cache lines	singel producer-multiple consumer	7.1%

Table 4. Problem size, data access patterns, and base miss rates at 16 processors for the benchmarks evaluated.

where n_i is number of L1-to-L1 transfers of type i, b_i is the benefit of one L1-to-L1 transfer of type i, and n_{mp} and P_{mp} are the number and penalty of mispredictions, respectively. b_i is calculated by subtracting c from the cost of one transaction in L2S for type i. For example, b for type 1 is

$$b_1 = 2*(d_{\mathcal{R},\mathcal{H}} - d_{\mathcal{R},\mathcal{P}})(L_{link} + L_{sw}) + L_{L2} - L_{L1}$$

where $d_{\mathcal{R},\mathcal{H}}$ is the distance between the requester ($\mathcal{R}$) and the home L2 bank ($\mathcal{H}$) and L_{L2} is the access latency of an L2 bank. Other b_i are calculated in a similar manner and will be larger due to indirection via the home for the L2S protocol. For example, type 2 will add $2*d_{\mathcal{H},\mathcal{P}}(L_{link} + L_{sw}) + 4*L_{link} + 2*L_{sw} + L_{L1}$ to the benefit b_2 in addition to b_1. The total cost can be calculated by

$$C = \sum_i n_i * c$$

The benefit calculation is conservative in that it does not take contention in the network into account. The expected improvement (speedup) in the memory component of the application is thus $Speedup_{memory} = \frac{C+B}{C}$. Applying Amdahl's law, we get a rough estimate of the expected speedup in total execution time for the application.

5.3 Performance Comparison

Performance: Figure 3 shows the performance of ARMCO normalized to L2S. We use execution time in terms of processor cycles required to do the same amount of work as our performance metric for all the benchmarks. ARMCO outperforms L2S for all the benchmarks, with speedup ranging from 1.04 to 2.26 having average (geometric mean) speedup of 1.13 (1.18 with microbenchmarks). For clarity, the normalized performance is plotted with the Y axis starting at 0.8. Figure 4 shows the timing break-down of the execution time (once again normalized to L2S). Timings were collected by instrumenting the workloads (except for Apache and JBB2005, where we do not separate the synchronization component). Note that as expected, the "non-synchronization computation" portion of time is the same for both ARMCO and L2S (except for GraphMine, where the use of work-queue style parallelization makes both warm-up point and exact computation performed non-deterministic).

Cache access distribution: Figure 5 shows the distribution of processor data cache accesses among local L1 hits, remote in-place read/write, remote hits resulting in replication from shared and modified state, remote hits resulting in migration from modified and migratory state, and L2 accesses (including L2 misses). The distribution is normalized to the total number of accesses made under L2S. The remote hits are from adjacent (with distance at most 2 hops) processors' L1 caches rather than L2.

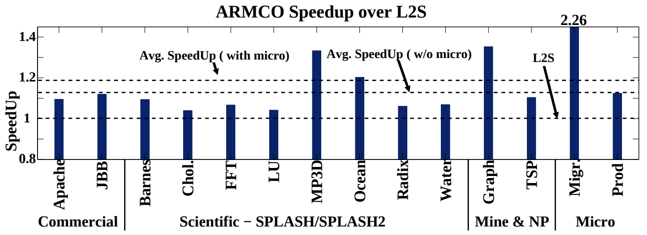

Figure 3. Performance speedup of ARMCO with respect to L2S (16 threads).

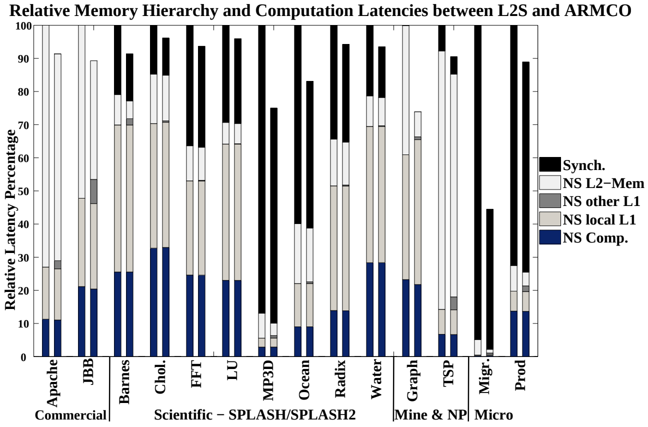

Figure 4. Timing breakdown among computation, different levels of the memory hierarchy, and synchronization, normalized to L2S (16 threads). For commercial workloads, we do not distinguish between synchronization and non-synchronization time. 'Synch' is all synchronization time spent in both computation and the memory hierarchy. 'NS' refers to non-synchronization. 'other L1' refers to time spent in direct L1-to-L1 transfers.

In the breakdown, we see the reflection of the characteristics of the benchmarks described in Table 4. ARMCO also reduces

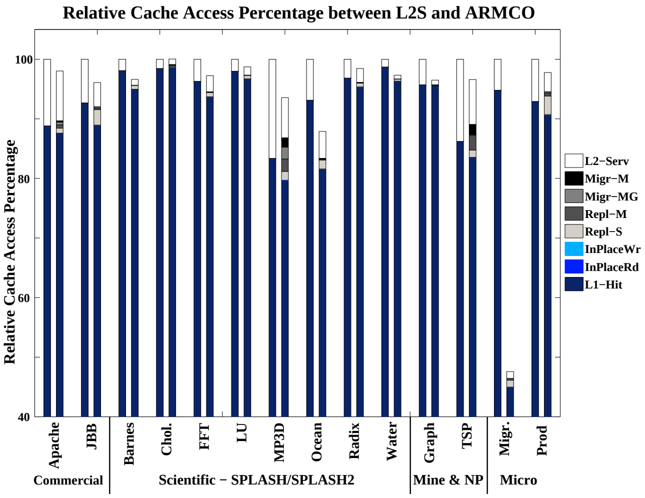

Figure 5. Normalized cache access distribution (16 threads). InPlaceRd and InPlaceWr occur at the predicted L1 cache without cache line allocation at the requester L1. Rep-S and Rep-M refer to remote L1 hits resulting in replication via L1-to-L1 transfers where the cache line is in shared and modified state respectively. Migr-M and Migr-MG refer to remote L1 hits resulting in migration via L1-to-L1 transfers where the remote cache line is in modified and migratory state respectively.

the number of cache accesses. This is a result of several factors, including reduced contention and wait time (resulting in reduced synchronization time) in locks and barriers, as well as reduced amount of operating system code (interrupts, scheduler, daemon processes, etc.) execution and data access as the benchmark completes in a shorter amount of time. Figure 6 shows the reduction in L2 accesses for ARMCO due to direct access to remote L1s.

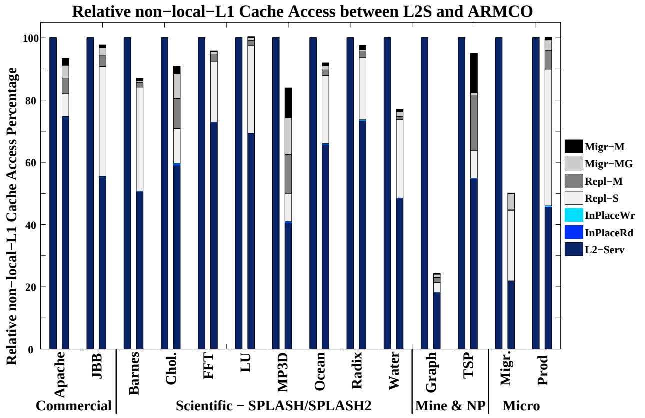

Figure 6. Normalized non-local-L1 cache access distributions (16 threads). Legends are described in the caption of Figure 5.

Using collected statistics on the number and type of direct L1-to-L1 transfers as well as distance among requester, predicted L1, and L2 bank, we used the models described in Section 5.2 to qualitatively verify our results. For example, for the migratory microbenchmark, the estimated speedup is 2.63 (by using L1-to-L1 transfer numbers from simulation), which correlates well with our experimental speedup results. The large reduction in synchronization overhead as a result of reducing contention and wait time, as well as the migratory optimizations, contribute to the high performance gain. For the Producer-consumer microbenchmark, we see a 12.4% performance improvement for ARMCO over L2S due to correctly predicting the ID of the producer, resulting in lower latency accesses by the consumers.

From [4], 44% of Apache's accesses are to read-shared cache lines and another 44% are to read-write-shared cache lines. Similarly, 42% of SpecJBB2005's accesses are to read-shared cache lines. Our JBB2005 version uses a concurrent thread-safe backend. Hence a substantial portion of the accesses are in read-write-shared mode. We see the reflection of this characteristic in Figure 6 where more than 25% of the L1 misses are satisfied through L1-to-L1 accesses, of which the larger fraction is from modified/migratory state for Apache, and 45% of the L1 misses are satisfied through L1-to-L1 transfers from shared state for SpecJBB2005, resulting in 9.5% improvement for Apache and 12% improvement for JBB.

In all cases, the data used for synchronization exhibits a migratory pattern, which is correctly recognized and handled by ARMCO. GraphMine also has a significant number of accesses exhibiting a migratory pattern, which is correctly recognized and optimized, resulting in 35.3% improvement. The synchronization-like behavior of GraphMine results in a substantial migratory pattern. In the case of TSP, there is little data sharing. However, there are a few falsely shared but heavily accessed cache lines and some data accessed in a migratory fashion. Via in-place read/write for the former and the migratory optimization for the latter, ARMCO is able to achieve a 12.4% performance improvement. MP3D has a lot of migratory, read-write-shared, as well as read-shared accesses due to high synchronization. ARMCO optimizes most of those accesses, which results in the highest improvement of 33.3% among the SPLASH applications.

Ocean has a substantial number of data accesses demonstrating the producer-consumer pattern among processors who have adjacent data in the matrix, resulting in a performance improvement of 20.3% (arising from a 35% reduction in L2 accesses due to direct L1-to-L1 transfers). Barnes also has similar producer-consumer as well as read-shared access patterns, showing a performance improvement of 9.4%.

Both MP3D and Ocean have high L1 miss rates, which provide greater opportunity for optimization. Cholesky, FFT, LU, Radix, Water are some of the applications with low L1 miss rates, resulting in lower performance improvement (4%-7%) wrt other applications.

5.4 Interconnection Network Bandwidth Requirements

The network link width is 16B. For L2S, we have 8B and 72B packet sizes and for ARMCO, we have 8B, 16B (at most 8B of data transferred along with L1-to-L1 requests), and 72B packet sizes. We use a configuration in which 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. Figure 7 shows the number of network packet-hops for ARMCO normalized with respect to L2S for the configuration in Table 3. The network's packet-hop numbers are accumulated by adding the number of hops traversed by each packet. We see a large reduction in packet-hops across all benchmarks when using ARMCO: 33.3% on average. Data (usually 8X of control info) transfer among cache controllers is the major contributor to the packet-hop count. Since

ARMCO attempts to move data using L1-to-L1 transfers from adjacent cores whenever possible, the total packet-hop count is significantly reduced.

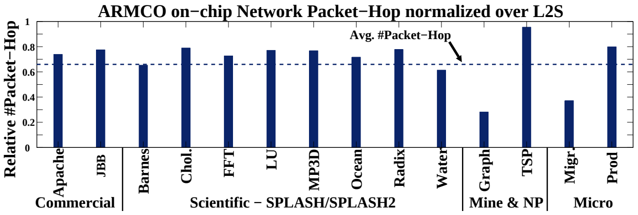

Figure 7. Normalized number of interconnection network packet-hops with respect to L2S (16 threads).

5.5 Dynamic Energy and Power Requirements

For power modeling, we use Cacti 6.0 [30] 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. Table 5 lists the energy values per access derived from Cacti 6.0. These numbers, along with collected access statistics, are used to derive dynamic energy numbers for ARMCO.

Figure 8 shows ARMCO's dynamic energy and dynamic power consumption normalized to L2S. ARMCO's reduction in dynamic energy and power is 31.2% on average (ranging between 6% and 72%) and 20.2% on average (ranging between -4% and 71%), respectively, with respect to L2S. While the extra state in ARMCO (predictor table and extra tag bits in the L1) adds to dynamic energy, this is sufficiently compensated for by a reduction in the number of packet-hops and by the substitution of L2 bank accesses with L1 accesses.

While we did not model leakage (static) power for the full system or dynamic power for the cpu logic, based on prior studies such as [11, 28], we estimate the dynamic energy in the on-chip memory hierarchy to be roughly 30% of overall chip energy consumption. This percentage is likely to go up if recent leakage reduction techniques (such as high-k dielectric from Intel [10]) are factored in.

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

Table 5. Dynamic energy consumption values per access for individual structures in ARMCO using 45nm technology (values are in femto-joules (fJ))

Predictor table accuracy and sensitivity: Since the performance of ARMCO is heavily dependent on correct prediction, we estimate the sensitivity of our results to the size and characteristics of the predictor table. We analyze the sensitivity of the results to predictor table size (2K, 1K, 512, and 256 entries) and associativity (16 and 8). The following list shows a few configurations of the predictor table and overall performance speedup (geometric mean) over L2S for all the applications excluding (including) the microbenchmarks:

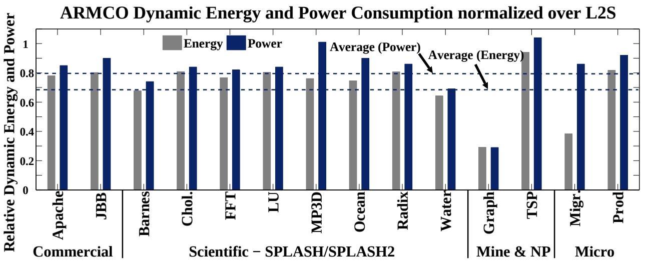

Figure 8. Normalized dynamic energy and power consumption of ARMCO with respect to L2S (16 threads).

	16-way	8-way
2K-entry	1.13 (1.19)	1.13 (1.18)
1K-entry	1.12 (1.18)	1.13 (1.18)
512-entry	1.10 (1.16)	1.12 (1.17)
256-entry	1.11 (1.17)	1.11 (1.16)
Ideal Predictor ARMCO		1.15 (1.23)
Ideal Pr. and Prefetch ARMCO		1.25 (1.40)

This suggests that a small table is sufficient to identify most of the sharing patterns. If the table is large with high associativity, it may hold old prediction information resulting in an increase in mispredictions and the penalty for misprediction might supercede the benefit from extra prediction. The table also has the average performance numbers for ARMCO with ideal predictor and prefetch which are described in next paragraph. Due to the inherent characteristics of directory-based protocols, there is limited opportunity for keeping the predictor table updated with accurate sharing information since we use existing messages to piggyback this information. Table 6 shows the percentage of L1 misses for which a prediction is made, and the percentage of those that are correctly predicted for each benchmark. Prediction accuracy ranges from 67.3% to 92.8% and is 74% on average. The percentage of L1 cache misses for which a prediction is made ranges from 21.4% to 74.5%, which is 50.5% on average. Most of these misses that are not predicted would be to non-shared data that do not require prediction.

	Apache	jbb	Barnes	Chol.	FFT	LU	MP3D	Ocean	Radix	Water	Graph	TSP	Mig.	Prod.	Avg
Coverage	21.4	53.5	62.4	41.5	37.2	47.8	64.8	41.0	37.1	55.6	38.0	62.9	74.5	69.2	50.5
Accuracy	92.8	81.4	70.9	78.6	68.2	70.4	67.3	73.5	68.6	68.2	76.8	69.2	71.5	78.9	74.0

Table 6. Percentage of L1 misses predicted (among L1 misses) and prediction accuracy

Comparison with ideal prediction: We evaluate ARMCO with an ideal predictor where correct prediction is made at request time. This can, however, be a miss when the request reaches the predicted L1 due to the activities during the request traversal time. We also evaluate ARMCO with an approximate prefetching effect: if the requested data line is in stable state in the system, a zero-latency fetch is performed (when in transient state, the request is enqueued at the predicted L1's incoming request queue with zero latency). Figure 9 shows the normalized performance of these systems with respect to L2S. We see a substantial performance improvement possible for many benchmarks by increasing the prediction accuracy. If we can, somehow, prefetch the required data from the predicted L1 then we will have even higher performance improvement. The reduced performance for GraphMine and TSP using ideal prediction with respect to regular ARMCO is the effect of the non-deterministic behavior of these two applications as the amount of work done varies based on changes in execution path.

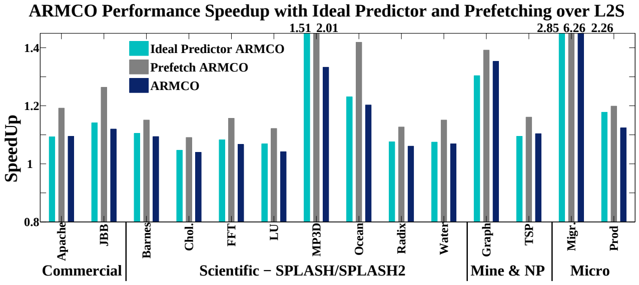

Figure 9. Normalized Performance of ARMCO with ideal prediction, ARMCO with ideal prediction and prefetch, and ARMCO with regular predictor over L2S (16 threads). Prefetching effect is incorporated by zero-latency fetching at request time from predicted L1 if the requested data line is at stable state in the system

5.6 Scalability of the System

We evaluate ARMCO and L2S on an 8-core, 16-core, and 32-core CMP system. Table 7 shows the average (geometric mean) speedup of ARMCO over L2S. ARMCO's performance over L2S increases with the increase of number of cores as the improved locality of access results in a bigger performance gain. ARMCO has better parallel efficiency for both 8→16 and 16→32, which suggests that predicting and accessing data directly from its current location is important to the scalability of future multi-core systems.

Workloads	Average Speedup		
	8-core	16-core	32-core
W/O Micro	1.10	1.13	1.15
W/ Micro	1.17	1.18	1.22

Table 7. Average (geometric mean) speedup of ARMCO over L2S for 8-core, 16-core, and 32-core CMP systems.

5.7 Sensitivity Analysis

We have analyzed the sensitivity of both ARMCO and L2S to the interconnection hop latencies and L2 latencies. Interconnect hop latencies are in the range of 3-5 cycles [30] and routers are usually 2-3 stage pipelined. We use a 2-cycle crossbar latency and vary the hop latency among 2, 4, and 6 cycles. We vary the L2 latency from 14 (Table 3) to 20 cycles. Table 8 shows the summary results. ARMCO improves its performance speedup over L2S with the increase of L2 latencies and interconnect hop latencies as suggested by the formulae derived in Section 5.2.

5.8 Comparing Performance with Victim Replication

While ARMCO optimizes memory-hierarchy performance for fine-grain sharing at the L1 level, the related proposals [4,6,8,20,37] do so at the L2 level. Figure 10 shows the relative performance of Victim Replication (VR) [37] (one of the above proposals) and ARMCO with respect to L2S for the same configuration as in Figure 3. We chose VR since it uses a similar baseline design (directory-based protocol and type of interconnect) to that used in our system. ARMCO outperforms VR for all the workloads. Our results for VR corroborate qualitatively with those in [4] for the workloads common to both studies — Apache, JBB, Barnes, and Ocean. Comparing relative magnitudes of improvement over VR in [4, 6, 8, 20], ARMCO compares favorably, indicating that optimizations at the L1 level help reduce overall latency and energy consumption.

Workloads	Hop Latency(cycle)			L2 Latency(cycle)	
	2	4	6	14	20
Commercial	1.10	1.11	1.13	1.11	1.13
Scientific	1.08	1.11	1.10	1.11	1.11
Mining & NP	1.10	1.22	1.18	1.22	1.22
Microbench	1.54	1.59	1.61	1.59	1.65
Average	1.14	1.18	1.18	1.18	1.20

Table 8. Influence of L2 cache and interconnection latencies. Data are ARMCO performance speedup over L2S categorized according to types of benchmarks.

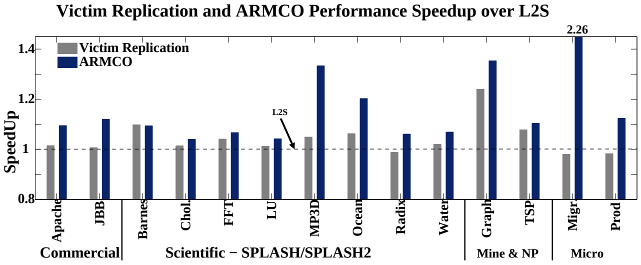

Figure 10. Normalized Performance with respect to L2S (16 threads) for VR and ARMCO.

5.9 Comparing Storage Overhead with Related Proposals

Section 3.1 describes the extra fields added to ARMCO, which contributes to some extra storage. For the target system (Table 3), we have 10 bits per L1 data cache line, which implies an overhead of 1.25 KB per L1 cache. The predictor table uses a 19-bit tag, 4-bit PP_{id}, 1-bit C_{op}, and a 1-bit Valid field per entry and 1024 entries per L1, which requires 3.13 KB per L1 cache. The total storage overhead for the target system is thus 70 KB, which is 0.38% of the on-chip cache-hierarchy. Table 9 compares this overhead qualitatively with other proposals at the L2 level.

6. CONCLUSIONS

The memory hierarchy design in a CMP must provide low-latency data communication for fine-grain sharing in order to truly harness the multi-core revolution. In this paper, we present a design that leverages direct L1-to-L1 access in order to facilitate low-latency fine-grain communication. Our protocol uses Adaptive Replication, Migration, and producer-Consumer Optimization(ARMCO) via hardware mechanisms that recognize and optimize for migratory, producer-consumer, multiple-reader, and multiple-writer sharing patterns. Our proposed protocol is able to reduce the average L1 miss penalty, resulting in performance speedup ranging from 1.04 to 2.26 (1.18 on average), and energy savings from 6% to 72% (31.2% on average). Both energy (and power, 20.2% on average) savings and performance speedup are a direct result of the reduction in the number of packet-hops (33.3% on average) due to the ability to access nearby cache banks, and secondarily due to accessing L1 rather than L2 caches.

The area cost as a fraction of the total on-chip cache storage is small — 0.38% in our design. Remote L1 cache location predic-

Proposals	Overhead Measuring Info	Overhead
ARMCO	3.13 KB (1K-entry) predictor table per L1-Data cache and 10-bit per L1D cache block	70 KB (0.38%)
ASR [4]	1 bit per L1 cache block, 2-bit per L2 cache block,8-bit per entry for 128K-entry NLHBs, and 16-bit per entry for 8K-entry VTBs	211 KB (1.14%)
Victim Replication [37]	1-bit per L2 cache block	32KB (0.17%)
CMP-NuRapid [8]	doubling each coreś tag capacity	1024 KB (5.54%)
Cooperative Caching [6]	cache tag duplication, singlet/reuse bits in cache, and spilling buffers at CCE	866 KB (4.69%)
Migratory Detection [13]	5-bit per L2 cache block	160 KB (0.87%)
Migratory Optimization [33]	5-bit per L2 cache block	160 KB (0.87%)
Producer-Consumer Opti. [7]	40 KB for delegate cache and RAC per 1MB L2	640 KB (3.46%)
Instr-based Prediction [18]	13-byte per entry in 128-entry predictor table per node	26 KB (0.14%)
Cosmos - Addr-based Prediction [29]	variable size for MHT and PHT	Variable (up to 21.9%)

Table 9. Storage overhead for different optimization proposals related to ARMCO for a 16 MB L2 CMPs system.

tion is a key enabler of the pattern-specific optimizations. While our results have been collected on a base protocol that is directory-based, ARMCO is possibly more effective on a broadcast-based protocol whether on a point-to-point or shared network, since the local predictor table is likely to be more accurate. Future work includes analysis using different base protocols and the development of a protocol that combines the benefits of ARMCO for fine-grain sharing and controlled victim replication to address the latency of mostly private data.

7. REFERENCES

[1] 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.

[2] 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.

[3] 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.

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

[5] 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.

[6] J. Chang and G. S. Sohi. Cooperative Caching for Chip Multiprocessors. In *Proceedings of ISCA-33*, pages 264–276, June 2006.

[7] 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 (HPCA)*, Feb. 2007.

[8] Z. Chishti, M. Powell, and T. Vijaykumar. Optimizing Replication, Communication, and Capacity Allocation in CMPs. In *Proceedings of ISCA-32*, pages 357–368, June 2005.

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

[10] Intel Corporation. Introducing the 45nm Next-Generation Intel Core Microarchitecture. http://www.intel.com/technology/architecture-silicon/intel64/45nm-core2_whitepaper.pdf.

[11] Intel Corporation. Power delivery for high-performance microprocessors. http://www.intel.com/technology/itj/2005/volume09issue04/art02_powerdelivery/p03_powerdelivery.htm, 2005.

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

[13] A.L. Cox and R.J. 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.

[14] N. Eisley, L. Peh, and L.Shang. In-network cache coherence. In *the 39th International Symposium on Microarchitecture (MICRO-39)*, pages 321–332, December 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 (SOSP)*, Dec. 1995.

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

[17] R. Garg, A. El-Moursy, S. Dwarkadas, D. Albonesi, J. Rivers, and V. Srinivasan. Cache Design Options for a Clustered Multithreaded Architecture. Technical Report TR 866, Department of Computer Science, University of Rochester, Aug. 2005.

[18] S. Kaxiras and J. Goodman. Improving cc-numa performance using instruction-based prediction. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, pages 161–170, Jan. 1999.

[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] 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.

[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 *the Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 206–217, June 2003.

[25] 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, Sep. 2005.

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

[27] R. Merritt. Ibm weaves multithreading into power5. *EE Times*, 2003.

[28] M. Monchiero, R. Canal, and A. GonzÃąlez. Power/performance/thermal design-space exploration for multicore architectures. *IEEE Transactions on Parallel and Distributed Systems*, 19(5):666–681, May 2008.

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

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

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

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

[33] 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*, May 1993.

[34] 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 (SOSP)*, pages 16–31, Dec. 1999.

[35] 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*, June 1995.

[36] K. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2):28–40, Apr. 1996.

[37] 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.