Predicting Coherence Communication by Tracking Synchronization Points at Run Time

Socrates Demetriades[†] and Sangyeun Cho[‡†]
[†]Computer Science Department, University of Pittsburgh
[‡]Memory Division, Samsung Electronics Co.
{socrates,cho}@cs.pitt.edu

Abstract

Predicting target processors that a coherence request must be delivered to can improve the miss handling latency in shared memory systems. In directory coherence protocols, directly communicating with the predicted processors avoids costly indirection to the directory. In snooping protocols, prediction relaxes the high bandwidth requirements by replacing broadcast with multicast. In this work, we propose a new run-time coherence target prediction scheme that exploits the inherent correlation between synchronization points in a program and coherence communication. Our workload-driven analysis shows that by exposing synchronization points to hardware and tracking them at run time, we can simply and effectively track stable and repetitive communication patterns. Based on this observation, we build a predictor that can improve the miss latency of a directory protocol by 13%. Compared with existing address- and instruction-based prediction techniques, our predictor achieves comparable performance using substantially smaller power and storage overheads.

1. Introduction

Inter-thread communication in shared memory systems is realized by allowing different threads to access a common memory space. This model simplifies the concept of communication; however, it creates important scaling challenges mainly due to the cache coherence problem [32]. Traditionally, shared memory architectures employ either a directory- or a snooping-based protocol to keep the per-processor caches coherent. Directories maintain a full sharing state for each cache line and therefore can precisely direct each miss to its destinations. The indirection to the directory adds, however, considerable extra latency to cache misses that are serviced by other caches. Snooping protocols avoid the latency and storage overheads of a directory by resorting to broadcasting messages on each miss; however, they place significant bandwidth demands on the interconnect even for a moderate number of processors.

A common approach to improving coherence communication is to predict the processors that a coherence request must be delivered to. Accurate prediction would reduce the latency of a cache miss by avoiding indirection to the directory, or reduce the high bandwidth demands of broadcasting by using multicasting in snooping protocols. Such predictions can be made by programmers (e.g., [1]), compilers [29, 47], or transparently by the hardware [2, 3, 8, 11, 28, 30, 31, 33, 36, 39]. Given that compiler techniques are limited to static optimization [29] and that the shared memory model should be kept transparent while offering high performance [39], a preferred communication predictor would dynamically learn and adapt to an application's sharing behavior and communication patterns.

Much prior work explored coherence target prediction using address- and instruction-based approaches [2, 3, 8, 27, 28, 30, 36, 39]. Address-based coherence prediction was first proposed by Mukherjee and Hill [39], who showed that coherence events are correlated with the referenced address of a request. To exploit the correlation, they associate pattern history tables with memory addresses, train them by monitoring coherence activity, and probe them on each request to obtain prediction. Alternatively, instruction-based prediction, as proposed by Kaxiras and Goodman [28], correlates coherence events with the history of load and store instructions. This allows a more concise representation of history information since the number of static loads and stores is significantly smaller than that of accessed memory blocks.

The basic design of address- and instruction-based predictor has been extended further to mainly relax the large space requirements of those approaches [8, 30, 36, 40]. However, the extensions still require relatively large and frequently accessed prediction tables. Furthermore, to attain high accuracy, they often keep long sharing pattern history per entry or rely on multi-level prediction mechanisms. Designs that exploit the spatial locality of coherence requests, such as the ones based on macroblock indexing [36], have shown improvements for both space efficiency and prediction accuracy, indicating that predicting sharing patterns at very fine granularities is not necessarily optimal. Nevertheless, the window for capturing such opportunities is still tight to hardware-level observation, limiting the scope in which communication localities can be expressed and exploited.

In this work, we propose *Synchronization Point based Prediction* (SP-prediction), a novel run-time technique to predict coherence request targets. SP-prediction builds on the intuition that *inter-thread communication caused by coherence transactions is tightly related with the synchronization points* in parallel execution. The main idea of SP-prediction is to dynamically track communication behavior across synchronization points and uncover important communication patterns. Discovered communication patterns are then associated with each synchronization point in the instruction stream and used to predict the communication of requests that follow each synchronization point.

SP-prediction is different than existing hardware techniques because it exploits inherent application characteristics to predict communication patterns. In contrast to address- and instruction-based approaches, it associates communication patterns with *variable-length, application-defined execution intervals*. It also employs a simple history structure to recall past communication patterns when the program execution repeats previously seen synchronization points. These two properties allow a very low implementation cost and hardware resource usage, yet delivering relatively high performance. In summary, this work makes the following contributions:

• We examine the communication behavior as observed between synchronization points for various multithreaded applications (Section 3). Our characterization reveals prominent prediction opportunities by identifying (1) strong communication locality during periods between consecutive synchronization points and (2) predictable communication patterns across repeating instances of such periods.

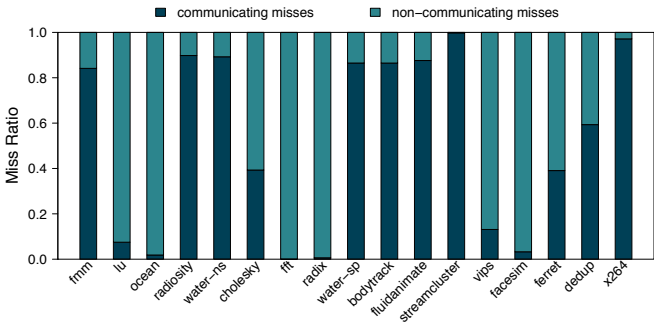

Figure 1: ***Ratio of communicating misses.*** **(Note: Details on the evaluation environment are given in later sections.)**

• We propose SP-prediction, a run-time technique to accurately predict the destination of each coherence request using a small amount of hardware resources. SP-prediction captures synchronization points at run time and monitors the communication activity between them. By doing so, it extracts a simple communication signature and uses it to predict the set of processors that are likely to satisfy coherence requests of the program interval, as well as requests that will occur in future dynamic instances of the same interval (Section 4).

• We fully evaluate SP-prediction over a directory-based coherence protocol on an elaborate chip multiprocessor (CMP) model (Section 5). Our results show that SP-prediction can accurately predict up to 75% of the misses that must communicate with other caches, without adding excessive bandwidth demands to the baseline directory protocol (below 10% of what broadcasting would add). Correct predictions translate into sizable reduction in miss latency (13% on average) and execution time (7% on average) compared to the baseline directory protocol. Compared to existing address- and instruction-based predictors, our approach achieves comparable performance, albeit at significantly lower cost.

2. Background and Motivation

Communicating misses. Coherence communication occurs on every memory request that must contact at least one other processor in order to be satisfied. Those requests, also called *communicating misses*[1], are read/write misses or write upgrades (upgrade misses) on cache blocks that have valid copies residing in non-local caches. Prior studies have shown that many applications incur a large fraction of such communicating misses [5, 36]. This fraction depends primarily on application characteristics like working set size, data sharing, and data reuse distance, as well as on cache parameters. Figure 1 shows results for the workloads studied in this work. On average, communicating misses account for 62%, with considerable variation among different applications. In general, applications with a high rate of communicating misses benefit from coherence target prediction.

Coherence communication prediction. Predicting the communication requirements of a coherence request involves guessing a set of processors *sufficient* to satisfy a given miss. A prediction scheme may exploit the communication behavior of recent misses to predict the next one, assuming that misses exhibit temporal communication locality. For instance, a prior study has shown that the two most recent destinations grab a cumulative 65% chance of sourcing the data of the next miss [25]. Communication locality is better captured, however, if misses are tracked based on the address they refer to, or the corresponding static instructions, thus motivating the address- and instruction-based prediction approaches.

Address-based prediction builds on the expectation that misses to the same address (cache block) will have to communicate with the processor that wrote on the same address previously, or the set of processors that read from the same address recently. Tracking misses in such fine granularity, however, adds significant area requirements. To reduce the overhead, a practical address-based predictor is implemented with limited capacity (i.e., as a cache), or/and indexed by blocks of larger granularity, e.g., a macroblock or page. As for the case of macroblock indexing, it has been shown to in fact improve both accuracy and space efficiency, since misses on adjacent addresses are likely to have identical communication behavior [36]. Similar in concept and motivation, instruction-based prediction resorts to the expectation that misses generated by the same static instructions will have related coherence activity. This compacts further the tracked information since the number of static load and store instructions is much smaller than the number of data addresses accessed.

The above prediction approaches are typically implemented as hardware mechanisms that consume a considerable amount of resources and are unaware of any application-level characteristics. However, the way parallel applications are coded and structured embodies intuition to create high-level understanding of how communication activity occurs and changes through time. This work examines the idea of exploiting such opportunity through the synchronization points that exist in applications.

Synchronization points. The shared memory model eliminates the explicit software management of data exchange between processors. Nevertheless, race conditions between concurrent threads require the explicit enforcement of synchronization points, through software mechanisms, to ensure that operations on shared memory locations are consistent. As a result, they naturally indicate points when certain data private to a processor will become visible—and possibly be communicated—to other processors. In what follows, we give a motivating example that shows how synchronization points partition the execution of an application into intervals, capture the existing communication locality in the application and, expose the repeatability of those partitioned intervals throughout the execution.

Figure 2 plots how a processing core communicates with other cores on a simulated 16-core CMP over (a) the whole execution (b) different execution intervals, and (c) dynamic instances of a single interval. By zooming into a granularity defined by synchronization points (plot (b)), it becomes clearer that the spatial behavior of the communication is strongly related to the specific intervals chosen. The sharp changes in communication behavior at the interval boundaries suggests that synchronization points are likely to indicate directly when behavior changes, and potentially hint a predictor to adapt faster to such changing behavior. In addition, the small set of processors that are contacted during each interval suggest that tracking the behavior on individual addresses or instructions within the interval may not necessarily result in more accurate prediction. Lastly, predictable communication patterns that may appear across the dynamic instances of the same interval (plot (c)) create a new scope of temporal predictability and a key opportunity to exploit the repeatability of the communication behavior.

To illustrate how such variations in communication behavior are manifested through shared memory programming practices, we list a simple example code in the following. Shared data (`ME` and `LE`)

[1] "Coherence request", "coherence miss", and "cache-to-cache miss" are also commonly used names.

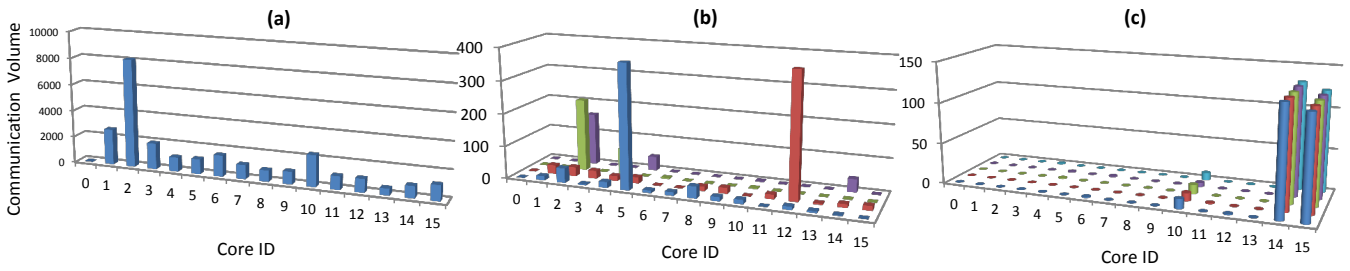

Figure 2: ***Communication Distribution of Core 0 in*** *bodytrack:* **(a) As seen during the whole execution. (b) As seen during the execution of four consecutive synchronization-defined sub-intervals. (c) As seen across five different dynamic instances of the same sync-defined interval.**

are exchanged between parents, children and siblings in a tree-like structure, which has its nodes arranged across multiple processors in a balanced way. During interval A, processors act as leafs and communicate data from processors where their parents and parents' sibling nodes reside. However, during interval B, processors act as inner nodes, hence the communication direction switches towards the set of processors that hold their children. This shift can be successfully detected and exposed by the synchronization point separating the two intervals.

Example Program Code

```
for nodes in this processor:
...
barrier();                              // interval A begins
node is a leaf:
  p = node.parent.LE[];
  for some node.parent.sibling:
    ps = node.parent.sibling.LE[];
...
barrier();                              // interval A ends
...                                     // interval B begins
node is a parent:
  for each node.child:
    node.LE[] = translate(node.child.ME[]);
...
barrier();                              // interval B ends
```

3. Communication Characterization

The communication behavior of a core over a certain interval can be characterized by the target cores with which it communicates (called *communication set*) and the *distribution of the communication volume* across that set. We have already shown examples of such distributions in Figure 2. In this section, we first introduce simple notions about synchronization point based intervals, and then we characterize the communication behavior of those intervals for various workloads.

3.1. Synchronization based Epochs

Synchronization primitives are implemented by various software libraries, often with different terminology and semantics, e.g., POSIX threads, OpenMP. Nonetheless, their range and use are similar in concept in most programming environments. We assume a POSIX thread library in this work; however, our methodology is applicable to other implementations.

A synchronization point (*sync-point*) is an execution point in which a software synchronization routine is invoked. Each sync-point has a type, e.g, `barrier`, `join`, `wakeup`, `broadcast`, `lock`, and `unlock`, and a static and dynamic ID. The static ID identifies each sync-point statically in the program code and corresponds to its calling location (e.g., program counter) or the lock address in the case of a lock sync-point. At run time, the dynamic ID uniquely identifies the multiple dynamic appearances of sync-points that have the same static ID. The

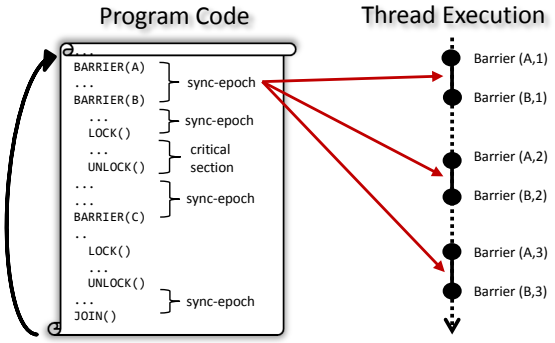

Figure 3: Static and dynamic sync-points and sync-epochs.

dynamic ID of a sync-point can be expressed with the corresponding static sync-point ID and how many times it has been executed so far.

Next, we define *synchronization epoch (sync-epoch)* as the execution interval enclosed by two consecutive sync-points. Based on this simple definition, on each sync-point, a new sync-epoch starts and the previous sync-epoch ends. A sync-epoch is described by the type, static ID, and dynamic ID of the beginning sync-point. Using our terminology, a critical section could be simply a sync-epoch that begins with a lock and ends with an unlock. A static sync-epoch that is exercised multiple times during execution creates *dynamic instances* of itself. Figure 3 depicts different sync-epochs and the notion of static and dynamic ID.

3.2. Simulation Environment

For the characterization study in this section, we employ a 16-core CMP model based on Simics full-system simulator [35]. The target system incorporates 2-issue in-order SPARC cores with 1MB private L2 cache, and a MESIF coherence protocol [23]. To track inter-core communication, we collected L2 miss traces that contain the miss data address, type, PC, and the target set of cores that must communicates with. The traces also contain all sync-points along with their type and static/dynamic IDs. Traces do not capture the effects of timing and are used only for characterization purposes. A full evaluation of our prediction scheme uses a detailed execution-driven performance model and is described in Section 5.

We study benchmarks from the splash2 and parsec suites [7, 48]. Table 1 lists key statistics related to sync-epochs for each studied benchmark. We use all available processor cores by spawning 16 concurrent threads in all experiments. For stable and repeatable measurements, we prevent thread migration by binding each thread to the first touched core. This was done except for dedup, ferret, and x264, because they create more threads than the available CPUs and rely on the OS for scheduling. Section 5.5 describes how our scheme can handle thread migration.

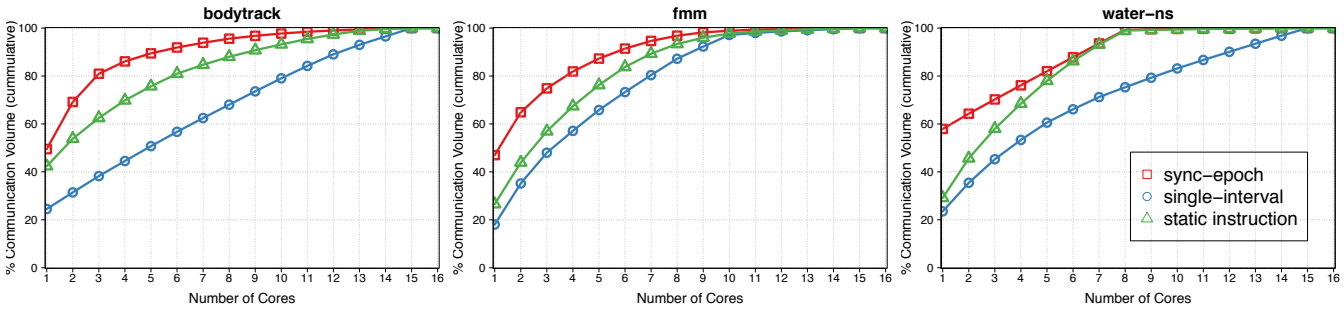

Figure 4: ***Average communication locality of*** *bodytrack, water-ns and fmm:* **Each curve shows the average cumulative communication distribution as seen in different granularities. Higher communication coverage for a given number of cores translates to better communication locality.**

BENCHMARK	# STATIC CRIT. SECT.	# STATIC SYNC-EPOCHS	PROGRAM INPUT	# TOTAL DYN. SYNC-EPOCHS
fmm	30	20	16K (particles)	2,789
lu	7	5	521 (matrix)	185
ocean	28	20	258 (grid)	2,685
radiosity	34	12	room	17,637
water-ns	20	8	512 (mol.)	1,224
cholesky	28	27	tk15.O	1,998
fft	8	8	256K (points)	22
radix	8	4	4M (keys)	35
water-sp	17	1	512 (mol.)	83
bodytrack	16	20	simsmall	456
fluidanimate	11	20	simsmall	8,991
streamcluster	1	24	simsmall	11,454
vips	14	8	simsmall	419
facesim	2	3	simsmall	3,826
ferret	4	6	simsmall	25
dedup	3	4	simsmall	508
x264	2	3	simsmall	56

Table 1: Sync-epoch statistics of benchmarks (per core average).

3.3. Communication Locality

The distribution of the communication volume characterizes the spatial behavior of the communication during an interval and illustrates whether it is "localized" to a certain set of targets. Examples of such localization are clearly observable in the communication distributions of Figure 2. For instance, core 0 during the first sync-epoch in example (b) communicates mostly with a single "hot" target, core 5, while nine other targets are contacted sporadically.

The communication locality is expressed by measuring the amount of communication volume that is *covered* by a certain number of cores. Using the previous example, core 5 covers more than 90% of the communication volume. Generally, if each individual miss communicates with C targets on average and the overall volume of the interval appears to be fully covered by C cores, then the interval has a perfect locality. When comparing different intervals with a similar C value, we can simply say that better locality exists as the communication is concentrated to fewer destinations.

A question that arises is *how good* is this locality relative to various granularities. For example, based on Figure 2(a), one could say that a certain level of locality also exists at the whole execution granularity since core 2 is "hotter" than the rest. To answer this question, Figure 4 shows the communication locality in applications, as captured by three different granularities: The sync-epoch granularity, the whole interval (as in Figure 2(a)), and the one that is based on static instruction indexing. Curves display average cumulative distributions over the whole execution and each point in the curve directly measures the average volume covered by a certain number of cores.

As the comparison shows, sync-epochs can capture the communication locality considerably better than a direct observation over the whole execution, suggesting that localities in communication's spatial behavior are closely related to sync-epochs. Moreover, sync-epochs often show better locality even to instruction-based granularity. This implies that communication activity could possibly be tracked as effectively as in traditional methods using sync-epochs—which is a much coarser-grain granularity. The results indicate that, overall, sync-epochs are attractive for extracting and exploiting repeatable communication behavior.

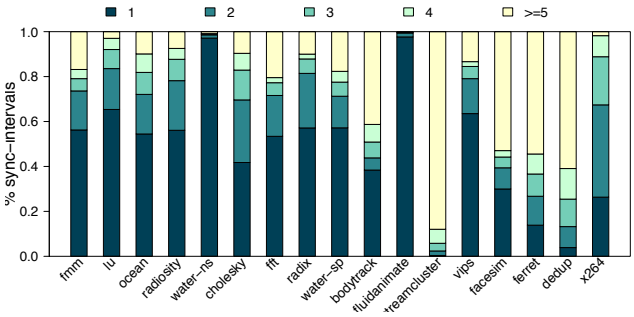

Figure 5: ***Distribution of intervals based on their hot communication set size:*** **More than 78% of intervals have a hot communication set size of smaller than or equal to 4.**

To create a representative signature of the communication behavior over each execution interval, we derive a *hot communication set* for each sync-epoch. A core is considered hot if it draws more than a certain amount of the total communication activity in the interval. Hence, the hot set could be formed based on a threshold over the communication distribution of the interval. The *size* of the set represents the amount of the interval's hot targets. Figure 5 shows, for each application, the distribution of sync-epochs based on the size of their hot communication set. The results consider a threshold of 10%, meaning that a core is considered hot if it is contacted by at least 10% of the total communication activity of the interval. In contrast to Figure 4 where only the average number of the hot communication set size is clear, the latter figure shows how this size varies among the sync-epochs of the applications. Note that to further measure how close the hot set size is to the optimal locality, one should also consider the average communication set size per miss.

3.4. Dynamic Instances of Sync-Epochs

Sync-points are executed repeatedly and create a sequence of dynamic instances for each sync-epoch. As these instances exercise the same or similar code and operate on the same (or related) data structures, it is likely that they present behavioral similarities between them [14]. Such similarities or variations may also be reflected on the communication's behavior, depending on how the shared data are accessed in each instance, their sharing patterns, the level of deter-

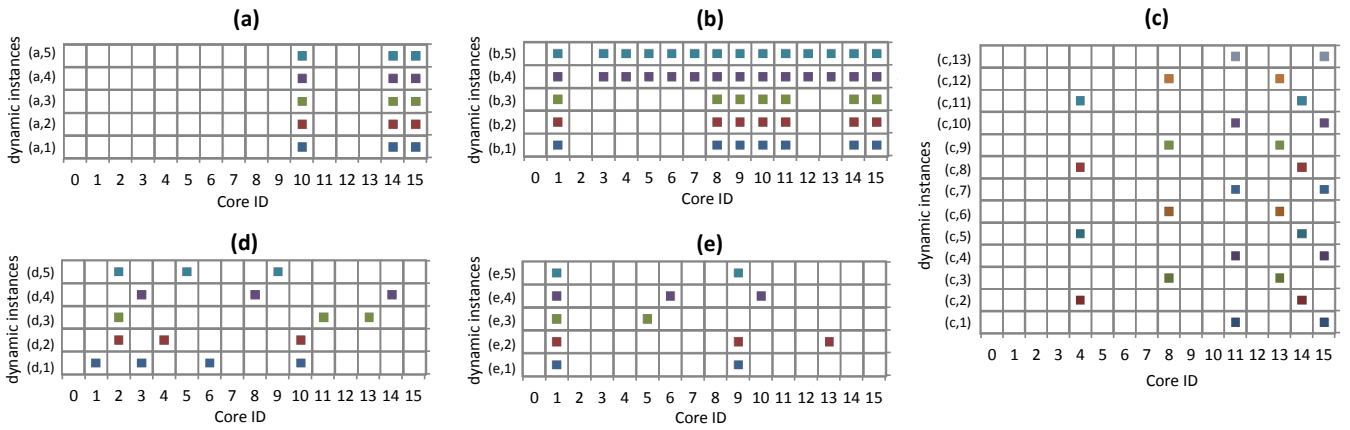

Figure 6: ***Example hot communication set patterns across dynamic instances of a sync-epoch:*** **(a) Stable pattern. (b) Change from one stable pattern to another. (c) Repetitive pattern with stride 3. (d) Random pattern (critical section). (e) Combination of stable and random hot destinations.**

minism, and possible machine artifacts, e.g., local cache capacity, false sharing effects.

Here we present our general observations on how communication activities appear in the dynamic sync-epoch instances in the examined applications. Our findings are derived from extracting the hot communication set of every dynamic instance of a sync-epoch, and characterizing how it changes from instance to instance.

Hot communication set patterns. Hot communication sets change across the dynamic instances of a sync-epoch following predictable or random pattern. We categorize the patterns into: *Stable*, *repetitive*, *random*, or some *combination* of these. Figure 6 illustrates example patterns by representing each hot communication set as a bit vector.

Stable hot communication sets occur when the majority of the data consumed each time are provided by a single core. This case is common in applications with stable producer-consumer sharing aligned to sync-epoch granularity. Hot communication sets that follow repetitive patterns are commonly found in fairly structured parallel algorithms that exercise a different but finite number of data paths on different sync-epoch iterations. For similar reasons, communication sets may also demonstrate spatial-stride or next neighbor patterns. In contrast, random patterns are usually caused by accesses on migratory and widely shared data that are produced/consumed in a non-deterministic order. Those occur when threads repeatedly compete before they are granted the privilege to produce data that will be shared (e.g., accesses within critical sections), or when the data sharing sequences are dynamically determined by the parallel algorithm (e.g., decisions made within critical sections). Patterns that appear to combine various patterns are usually an artifact caused by the granularity in which we track the communication (e.g., a long sync-epoch may span across multiple functions and data structures, each having different sharing patterns).

"Noisy" sync-epoch instances. Oftentimes, some dynamic instances of a sync-epoch appear to have very low communication activity relative to other instances. This is usually caused by a control statement, which forces specific instances to flow through different execution paths that exercise code with relatively few accesses to shared data. Such instances may not give a representative sample when forming a hot communication set due to statistical bias; therefore, we treat them as noise and exclude them from the dynamic pattern.

4. Sync-Epoch based Target Prediction

The existence of communication locality at the sync-epoch granularity implies that misses within the sync-epoch are likely to communicate with processors in the hot communication set. Thus, the hot set, if known, could be a *relatively small* and *sufficient* target predictor for the majority of misses within the interval. Based on this observation and, on evidences that many hot communication sets are predictable, we propose **SP-prediction**, a run-time scheme that exploits the temporal predictability within and across sync-epochs to predict the communication destination of misses.

SP-prediction is different from other prior approaches that exploit the temporal sharing patterns of misses in two fundamental ways. First, it makes use of the communication locality over application-specific execution intervals to predict for each miss in the interval, with no reliance on the temporal communication locality between consecutive misses. This is a significant advantage when communication locality is only seen among a broader temporal and spatial set of misses. Second, it can recall communication patterns from the past at a sync-epoch granularity and not for specific address or instruction. This may allow the predictor to adapt quickly to old and forgotten patterns without complex mechanisms and long history information.

4.1. Basic Idea of Run-Time Prediction

SP-predictor exploits sync-epochs' communication locality to predict the destinations of a miss. Each program thread is seen as a sequence of sync-epochs, many of which are exercised multiple times during program execution. Obtaining a predictor of the communication behavior in a sync-epoch involves retrieving history information from previously executed instances of the same sync-epoch, as well as tracking the coherence communication of the currently executed interval. Each private L2 cache controller would hold the obtained predictor and accelerate miss-incurred communication by invoking a prediction action in the standard coherence protocol on each miss.

Synchronization primitives are exposed to the hardware so that it can identify the sync-epochs and sense their beginning and end. This requires simple annotations in the related software library (or program code) and corresponding support in the hardware. The hardware design cost entails the addition of a new instruction that retrieves the PC or lock address of the sync-point and forwards it to the coherence controller. The insertion of the instruction in the code is trivial and could be done by the library developer or automatically by a compiler. We consider that such support is feasible in today's hardware and software, and similar implementations exist (e.g., [10, 45]).

EVENT	ACTION
Sync-point captured (*sync-epoch begins*)	- Store sync-epoch's tag and type into SP-table. - Reset all communication counters.
Data response on RD/WR-miss	If the response comes from a remote node's cache: - Increment communication-counters[responder].
Invalidation Ack responses	- Increment communication-counters[responders]
Sync-point captured (*sync-epoch ends*)	- Extract hot communication set from counters - Store the hot set as a signature to the SP-table

Table 2: Building communication signatures.

EVENT	ACTION
Sync-point captured	Retrieve d signature(s) from SP-table Obtain predictor: - If d=0 => extract current hot set (after warmup) - If d=1 => last hot set - If d=2 => last stable hot set - If d>=2 => test for pattern (if supported) - If sync-point is a lock => last d processors holding the lock Forward predictor to the L2 controller
RD/WR-miss	- Invoke a prediction action using the obtained predictor.
Confidence alert	- Extract new hot communication set - Replace predictor with new hot set

Table 3: Obtaining prediction.

4.2. Building Communication Signatures

Each processor monitors its communication activities by tracking responses to misses that have invoked the coherence protocol. A set of *communication counters* record the overall communication towards each destination. Responses for read misses include the data provider's ID and increment the communication counter that corresponds to the source processor. Responses for write and upgrade misses include a bit vector capturing the invalidated processors and increment the communication counters that correspond to the invalidated set. The communication counters are reset at the beginning of each sync-epoch. Effectively, as the execution progresses within the sync-epoch, the counters would reflect the processor's communication spatial behavior up to the current execution point within the sync-epoch. At the end of the sync-epoch, the hot communication set is extracted from the counters and stored as a communication signature (bit vector) in a history table called *SP-table*.

When the sync-epoch is a critical section, the communication signature encodes only the ID of the processor that releases the lock. This allows other critical sections that are protected by the same lock to retrieve and use this information as their possible communication target. Note that for noisy instances (Section 3.4), no communication signature is stored. Table 2 summarizes how the communication signatures are constructed during the execution.

4.3. SP-Table

SP-table is an associative table where each entry records a single, per processor, static sync-epoch. Entries are indexed/tagged with the static ID of the sync-epoch and the processor ID. For locks, entries are tagged with the lock variable and are shared by all processors. This allows all critical sections protected by the same lock (in the same or different threads) to share the same communication history.

Each SP-table entry keeps a sequence of communication signatures. This sequence has a bounded size d, the *history depth*. Whenever a sync-point is encountered, SP-table is probed to store the signature of the ending sync-epoch and retrieve the signature(s) of the next sync-epoch. Updates involve shifting out the oldest signature and shifting in the newest. For critical sections, updates occur just after the lock is acquired. This ensures atomic updates in the shared entries and avoids lookups of the table when a processor spins on a lock.

4.4. Obtaining Predictions

When a new instance of a previously seen epoch is detected, the associated communication signature(s) are retrieved from SP-table to generate a destination predictor for the misses that will occur in the new instance. The obtained predictor for the sync-epoch will be forwarded to the processor's L2 cache controller and will trigger an action to the coherence protocol on each miss. The state of the predictor would be simply the previous communication signature or some combination of previous signatures. A summary of how the predictor is formed is given in Table 3. More specifically:

No history available ($d = 0$)**.** If the sync-epoch is met for the first time (or if no history table exists), then history information is not available. In this case, the predictor uses a hot communication set that is extracted from the communication counters while the sync-epoch runs, after allowing some warm-up time, e.g., 30 misses. This would essentially form a predictor that predicts requests based on the activity recorded in the early stages of the interval.

Last hot communication set ($d = 1$)**.** If only one history signature is available so far (or if the table has history depth of one), then the predictor uses the last—and only available—communication signature stored in the corresponding predictor entry.

Last stable hot communication set ($d = 2$)**.** The intersection between communication bit vectors (bit-wise AND) returns the set of destinations that remain stable across the instances. Our predictor combines only the two most recent bit vectors, since this successfully catches stable destinations across consecutive instances, as well as adapts faster to changing stable patterns such as the one shown in Figure 6(b).

Pattern-based hot communication set ($d \geq 2$)**.** A longer history of signatures available to a sync-epoch could capture further hot communication set patterns such as the repetitive pattern shown in Figure 6(c). Specifically, to capture such repeatable patterns, history depth should be at least as large as the repetition distance (or stride) of the pattern, e.g., $d \geq 3$ for the same example. Hardware could detect a repetitive pattern by comparing a new bit vector with all the stored bit vectors, saving the depth s of the one that matches, and correctly predicting the next bit vectors using the one at depth $s-1$. Our current predictor is tuned to detect only repetitive patterns of stride-2, as it uses a history depth of no more than two.

Lock sync-point. If the captured sync-point is a `lock`, then the retrieved signatures will indicate the sequence of processors holding the lock last. A union of the available d signatures will therefore form a prediction set that includes the last d processors that have held the lock. The predictor may be further extended to return a union that also includes the bit vector of the preceding sync-epoch, as coarse critical sections are likely to benefit from it.

In order to detect and recover from pathological cases where the predicted communication set does not provide correct prediction, we employ a mechanism that sense low prediction confidence and adapts to a new hot communication set. A recovery step is usually needed in coarse sync-epochs, where communication's spatial behavior could oscillate within a sync-epoch instance. In our current design, the confidence mechanism is a simple 4-bit saturating counter that increments on correct predictions and decrements otherwise. On each new interval, the counter starts with a high confidence towards the predicted communication signature (counter is fully set) and triggers a recovery step if the confidence level drops below a threshold (counter is zero). To recover, we reconstruct the predictor by extracting the hot communication set of the currently running interval, as it appears up to the current point. The hot set is extracted based on the information recorded in the communication counters that dynamically track the

communication activity of the interval.

4.5. Integration to the Coherence Protocol

SP-prediction requires additional functionality in the coherence protocol. However, it does not interfere with the base protocol and operates on top of it. We briefly describe how our protocol arbitrates prediction actions, verifies results, and recovers from mispredictions.[2] As a baseline protocol, we use a directory-based MESIF coherence protocol, an extended version of MESI that effectively supports cache-to-cache transfers of clean data [23]. Note that the prediction engine can be integrated into any directory-based protocol, or any snoop-based protocols that can recover from mispredictions [8, 36].

• Requesting node: When an L2 miss for a memory line occurs, a prediction request is generated. The request is sent to the node(s) predicted to have the valid copy of the line and includes a bit identifying it as predicted. The request is also sent to the directory along with a bit vector identifying the predicted nodes.

• Directory: The directory node will receive the bit vector of predicted nodes for every miss and detect whether the targeted set was sufficient or not. Upon detecting a misprediction, it will satisfy the request as it would normally do, resulting in a miss latency similar to the baseline protocol. If the request was for upgrade or write miss with multiple sharers, the directory will invalidate the sharers that were not predicted (if any), and reply to the requesting node, indicating whether the predicted set was sufficient or not and which sharers were correctly predicted.

• Predicted node: When a predicted request for a memory line arrives at the cache controller, the line is searched in the L2 cache. If the line is in Exclusive, Modified, or Forwarding state [23], then a copy of the line is immediately forwarded to the requesting processor. Also, an update message is sent to the directory indicating the new sharing state of the cache line. If the line must be invalidated (i.e., due to request for exclusive ownership), an Ack message is sent back to the requesting processor after invalidation. Otherwise, the cache replies with a Nack message.

• The requesting node will receive responses from the predicted nodes, and also from the directory in case the request was for exclusive ownership (write or upgrade miss). Upon receiving data, the controller will perform line replacement as usual and, if the request was a read, the miss will be completed. If the request was for exclusive ownership, then it will be completed only after the response from the directory and the necessary invalidation Acks from the correctly predicted sharers have arrived (if any). Given that the directory is always aware of the prediction result and can proceed as normal on mispredictions, it is unnecessary for the requesting node to reissue requests.

4.6. Discussion on SP-Table Implementation

SP-table can be implemented either in system software or hardware. In the former case, the table is statically allocated at boot time by the OS and kept at a certain memory location. Every sync-point will invoke a trap to the OS, which will handle all necessary operations on SP-table and return a predictor for the upcoming sync-epoch. In a hardware embodiment, a slice of SP-table can be integrated with the L2 cache controller on each processor and hold the information specific to that processor. Table entries that are shared by all processors (for lock sync-points) could be either located at a centralized location on chip, or distributed across the slices in an address-interleaved fashion. All implementations assume that the sync-point's PC, lock address and the processor ID can be extracted at the processor, and the necessary information can be piggybacked and transferred between the hardware and software components involved.

[2]More details on how the protocol handles race conditions and conflicts can be found in similar extensions [2, 3, 8, 43].

Parameter	*Value*
Proc. model	in-order
Issue width	2
L2 Cache (private)	
Line size	64 B
Size/Assoc.	1 MB, 8-way
Tag latency	2 cycles
Data latency	6 cycles
Repl. policy	LRU

Parameter	*Value*
L1 I/D Cache	
Line size	64 B
Size/Assoc.	16 KB, 1-way
Load-to-Use lat.	2 cycles
Network-on-Chip	
Topology	4×4 2D mesh
Router	2-stage pipeline
Main mem. lat.	150 cycles

Table 4: Simulated machine architecture configuration.

SP-table has fairly low space requirements. Each slice requires as many entries as the number of static sync-points in an application, which is generally small ($\leq 30 + 2\sim3$ entries as shared portion). Each entry may hold more than one signatures, depending on the history depth (we allow no more than two in our evaluation). The length of the signature (in bits) is equal to the number of processors (e.g., 16-bits for a 16-core CMP). Each entry also has a 32- or 64-bit tag (PC) depending on the machine's architecture and an additional bit indicating whether the entry is shared, i.e., a lock. Although each SP-table slice is considered to work as fully-associative, a smaller set-associativity array is also possible without much cost from set conflicts. A 2 KB aggregate SP-table is adequate to hold all necessary information for even the most demanding applications (including 32-bit tags). As we will discuss later in Section 5, this size is significantly smaller compared to address- or instruction-based tables.

The location and management of SP-table is an implementation choice that has no significant performance implications, since it is small and accessed relatively infrequently (only on sync-points). A hardware implementation would generally be more appropriate if sync-epochs are short, e.g., the application has very fine-grain locking. In general, the SP-table design should be dictated by both the design goals and the target application domain.

5. Evaluation

5.1. Methodology

To evaluate the performance of the proposed predictor, we extend the system described in Section 3.2 with detailed timing models for cache hierarchy and interconnect. The target system is a 16-core tiled CMP with a 4×4 2D mesh network-on-chip (NoC), similar to models used in recent studies and commercial developments [12, 46]. Each tile incorporates a processor core that has two levels of private caches, coherence logic, and a NoC router. Coherence is maintained through a distributed directory-based MESIF coherence protocol with some extensions as described in Section 4.5. The NoC operates at the processor core frequency and is a wormhole-switched network with deterministic X-Y routing and Ack/Nack flow control. Table 4 summarizes our architecture configuration parameters.

For the SP-table, we consider a distributed hardware implementation and each entry can hold no more than two signatures ($d = 2$). The SP-table is accessed only on sync-points and the access latency is rarely in the critical path. Updates on communication counters complete in a single cycle, and we account four cycles for extracting a hot communication set. We present the performance of the SP-predictor with respect to the baseline directory protocol and a

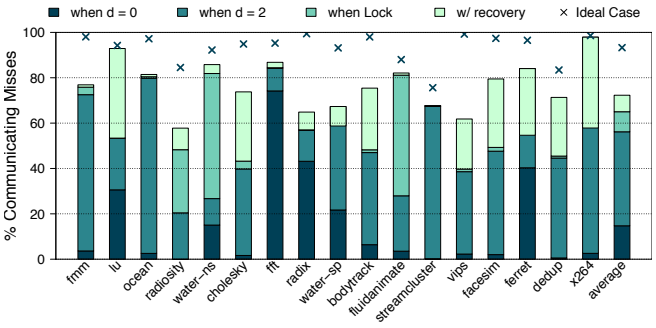

Figure 7: ***SP-prediction accuracy:*** **Percentage of communicating misses that avoid indirection to the directory.**

broadcast protocol. Results consider both serial and parallel sections, although the predictor is effective only during parallel sections. To fairly evaluate a broadcast snoop-based protocol, we assume a totally ordered interconnect with the same configuration as the one with directory. At the end, we compare our prediction approach against a simple locality-based predictor and state-of-the-art address- and instruction-based destination set predictors [36].

5.2. Prediction Effectiveness

Prediction is correct when the predicted set is sufficient to satisfy a communicating miss, i.e., a superset of the sharing information in the directory. The *size* of the predicted set—which is the size of the hot communication set in our case—creates a trade-off between prediction accuracy and bandwidth waste. The fewer the cores included in the predicted set, the less the probability to communicate with the correct cores(s) for each request. On the other hand, the more cores in the predicted set, the more redundant messages will be sent, and hence the more bandwidth will be added on the interconnect. In our evaluated scheme, the size of the hot communication set depends on the communication locality of each sync-epoch as explained in Section 3.3, and adapts to the changing communication patterns as described in Section 4.4.

Figure 7 shows the percentage of communicating requests predicted correctly. On average, the SP-predictor correctly predicts and eliminates indirection to the directory for 77% of all communicating requests, with 98% (x264) and 59% (radiosity) as the best and the worst case, respectively. The crosses indicate the accuracy that the SP-predictor could obtain ideally, if the hot communication set for each sync-epoch was known a priori. The gap between the actual and the ideal accuracy comes from the lack of predictability in some sync-epoch instances and the sensitivity level of the recovery mechanism. This gap may be bridged somewhat if off-line profiling offers initial prediction information and the sensitivity level is adjusted dynamically.

The percentage breakdown indicates the prediction accuracy when different information was available to the SP-predictor. The bottom stack accounts for correct predictions made when no information from past sync-epoch instances was available. Such situations appear in applications where major sync-epochs are not replayed (fft, radix and ferret). In those cases, the predictor relies mostly on most recent within-interval communication activity to predict miss targets. The next two stacks correspond to misses correctly predicted based on signatures from past sync-epochs, indicating separately those occurring within critical sections. Applications with highly repeatable sync-epochs such as ocean and streamcluster can take advantage of the pattern-based prediction policy. Similarly, applications with fine-locking such as water-ns and fluidanimate gain with highly accurate

BENCHMARK	AVG. ACTUAL TARGETS PER REQ.	AVG. PREDICTED TARGETS PER REQ.	RATIO OF PREDICTED TO ACTUAL
fmm	1.19	3.11	2.61
lu	1.01	2.46	2.46
ocean	1.08	3.15	2.94
radiosity	1.11	4.12	3.71
water-ns	1.41	2.53	1.80
cholesky	1.04	1.89	1.83
fft	1.01	2.37	2.36
radix	1.00	2.75	2.75
water-sp	1.58	2.75	1.75
bodytrack	1.13	2.8	2.49
fluidanimate	1.14	2.05	1.79
streamcluster	1.14	1.95	1.72
vips	1.01	2.06	2.05
facesim	1.04	2.56	2.47
ferret	1.01	1.14	1.13
dedup	1.10	2.34	2.15
x264	1.01	1.93	1.93

Table 5: Average actual and predicted set size.

predictions due to the ability of our predictor to retrieve the random sequence in which threads execute the critical sections. On average, those sync-epoch history-based predictions account for up to 40% in prediction accuracy. Sync-epochs with unpredictable intervals will eventually adapt their predictors based on the recovery mechanism and correctly predict an additional 9% of requests on average.

Messages will be wasted if the predicted target set for a miss is incorrect, or larger than the minimum sufficient target set. Table 5 summarizes the differences between the minimum and the predicted average target set size. The minimum sufficient set size is generally close to 1 since read requests—which are the majority—must always contact only a single destination.[3] By comparing separately the reads and writes, we found that, on average, the predicted set includes 1.4 and 0.5 more targets per request respectively. More insight on how the prediction affects the bandwidth demands is given by more detail results presented later in this section.

The way the hot communication set is extracted (Section 3.3) strongly affects the trade-off between latency and bandwidth. The current policy leads to some bias towards higher bandwidth when the locality is poor, since there are no strict bounds on the maximum size of the set. In general, the policy can be tuned depending on the design goals and requirements. For example, in a case where bandwidth demands must be bounded to avoid exceeding a power envelope, one could tune the policy to extract a hot set that does not exceed a certain size.

5.3. Performance Results

Impact on miss latency. Correct predictions will satisfy misses without paying the cost of indirection to the directory, thereby reducing the average cache miss latency. Incorrect predictions are detected by the directory, which will then satisfy the miss without noticeably degrading the latency of the indirected miss. Figure 8 shows the average miss latency achieved by the SP-predictor and the baseline protocols. Average latency is calculated by treating each miss individually, and results are normalized to the directory protocol. The results show that on average, SP-prediction reduces miss latency by 13% relative to the directory protocol and attains up to 75% of what the broadcast snooping protocol can achieve. Under the (true) assumption that the NoC does not get severely congested, the broadcast scheme approximates the ideal case in terms of miss latency.

The predictor predicts correctly and reduces the latency for both read and write requests. A correctly predicted "read" has slightly

[3]The reported numbers assume a cache-to-cache transfer request for clean data to have a sufficient set size of 1, which is not necessarily true in a MESIF protocol [23].

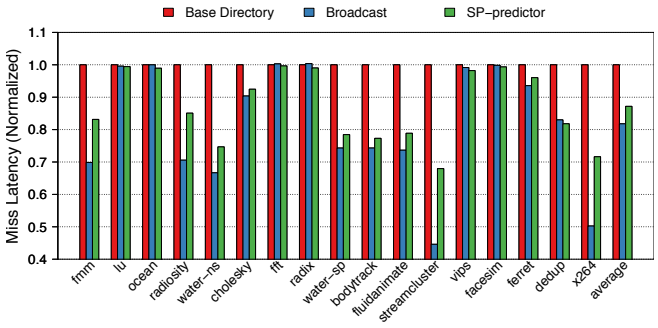

Figure 8: Average miss latency. (Note: Y-axis starts at 0.4.)

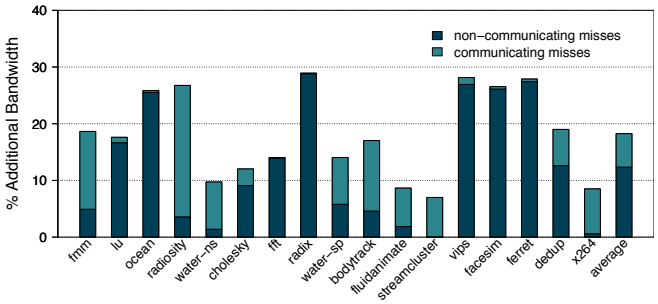

Figure 9: Additional bandwidth demands of SP-prediction relative to the base directory protocol.

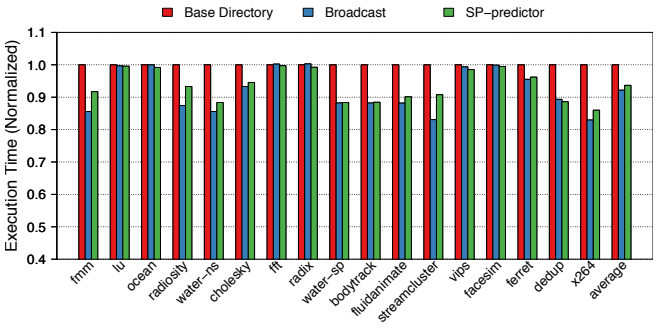

Figure 10: Execution time. (Note: Y-axis starts at 0.4.)

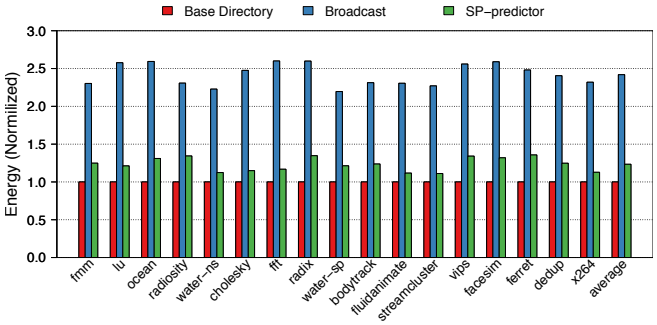

Figure 11: Energy consumed on NoC and cache lookups.

higher impact compared to a correctly predicted "write", as writes may have multiple targets to reach and wait for acknowledgments. Also, the prediction accuracy slightly declines as the number of the targets increases. Nevertheless, write requests with multiple targets are generally a small fraction of the overall misses, and their impact on the overall reductions in latency is limited.

Marginal improvements in some applications (e.g., lu, radix) are due to the limited fraction of communicating misses (recall Figure 1). The smaller this fraction is, the fewer the opportunities for latency reduction. Moreover, the high miss latency of non-communicating misses (i.e., off-chip misses) will, in the end, overshadow the improvements coming from accelerating on-chip, communicating misses. A quick look at how this fraction varies across the applications directly explains why the miss latency reduction is limited for each application. Note that this also limits the effectiveness of the broadcasting scheme. It is generally possible for a larger cache size to elevate the fraction of communicating misses for memory bound applications, and hence increase the impact of the predictor to the miss latency reduction. Sensitivity analysis of cache parameters and workload input sizes (not reported in this work) have shown expected observations and trends.

Impact on bandwidth requirements. To measure the impact of target prediction on bandwidth, we track the number of bytes transmitted on the NoC due to L2 cache misses. These include request messages to predicted cores, request and update messages to the directory, and control and data responses. Figure 9 shows the additional average bandwidth requirements of a coherence request, relative to those of the baseline directory protocol. The results show that SP-prediction increases the bandwidth requirements by 18% compared to the baseline. The snooping protocol would have the highest bandwidth demands since messages are broadcast to all targets on each miss, whereas the directory protocol essentially approximates the ideal case possible. Overall, SP-prediction keeps its additional bandwidth requirements below 10% of what the broadcasting protocol would additionally demand to the baseline directory protocol (the actual bars for broadcasting are not shown due to the very large difference).

Much of the additional bandwidth comes from the (always unfortunate) attempts to predict non-communicating misses. This portion is shown by the bottom stack and accounts for 70% of the overhead. Applications with a large fraction of non-communicating misses will therefore increase the bandwidth demands with no positive return in latency. Prior work has shown that most of such attempts can be detected and avoided by simple snoop filtering [38]. For example, a simple low cost TLB-based snoop filter can detect ∼75% of them [17]. Thus, the use of orthogonal techniques can substantially reduce the associated bandwidth overheads without compromising the latency improvements.

Impact on execution time. Figure 10 depicts the overall improvements in execution time as a result of reducing miss latency. SP-prediction improves the execution time by 7% on average, with x264 seeing the best improvement (14%). Depending on the interconnect design and control parameters, an excessive traffic could congest the network and affect the performance negatively. In our simulated system, congestion levels remain low for both, the prediction-augmented directory protocol and base broadcast protocol. Marginal negative impact was observed for broadcasting only in applications with very small fraction of communicating misses.

Impact on energy. We estimate the energy impact of SP-prediction using an intuitive analytical model that considers the dynamic energy consumed on the interconnect and L2 cache snoops. For the network, we assume that the energy consumed is proportional to the amount of data transferred [4]. We also assume that the energy consumed in a router is four times that consumed in the link. For cache snoops, a single cache tag lookup energy is estimated using CACTI [21], assuming a 32nm technology. Figure 11 presents the normalized

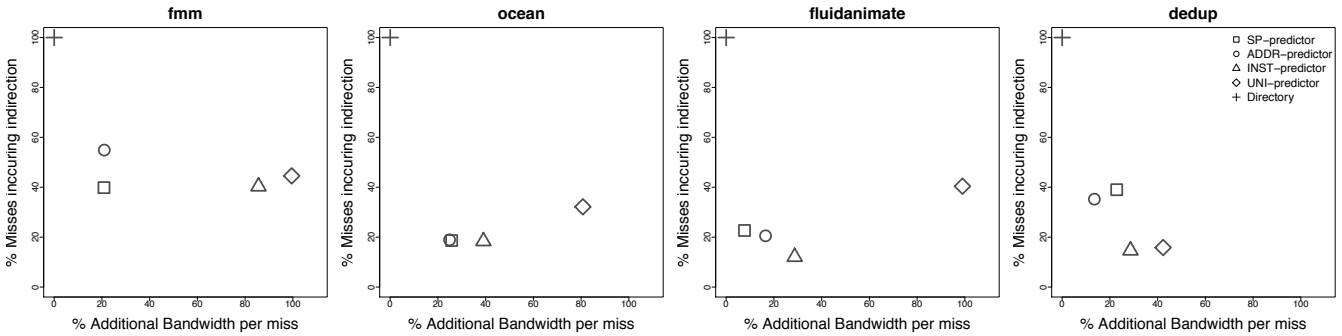

Figure 12: ***Performance/bandwidth trade-off comparison:*** **The lower-left corner represents the best point on the trade-off space. The results are expressed relative to the directory-based protocol, which is indicated with a "cross" symbol at the upper-left corner.**

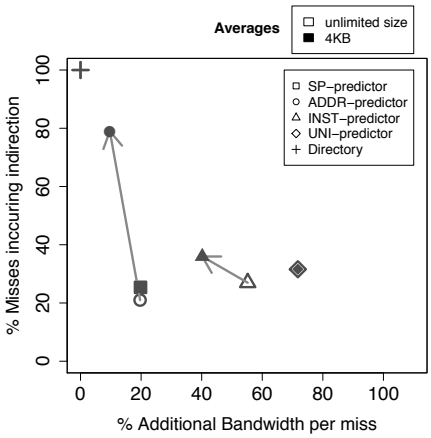

Figure 13: ***The effect of space requirements to prediction performance:*** **SP-prediction and UNI-prediction are not affected since they have significantly lower space requirements.**

results. Enabling SP-prediction over a directory protocol increases the energy requirements on network and cache lookups by 25% in total. Yet, this is substantially less compared with the energy requirements of snoop broadcasting (2.4×). Considering that a large fraction of traffic and snoop overhead could be filtered, as discussed previously, the new energy demands could be brought down to below 8%.

5.4. Comparison with other Predictors

We compare SP-prediction with address- and instruction-based prediction, implemented according to the "group" destination set prediction model proposed by Martin et al. [36]. In addition, we compare with a simple locality-based predictor that uses no index, i.e., predicts simply based on the coherence activity of previous misses, independent of their address or instruction. For abbreviation we will refer to them as ADDR-, INST-, and UNI-prediction, respectively. The ADDR and INST prediction models use both external coherence requests and coherence responses to train a predictor for each data block or instruction. The UNI predictor uses only the coherence responses, i.e., it is trained based on the targets of previous misses by the same core.

All the predictors return a group of possible sharers, aiming high prediction accuracy while making best efforts to keep the bandwidth requirements small[4]. Each predictor entry incorporates a two-bit counter per core that accumulates the recent activity towards each destination, and a train-down mechanism which ensures that the predictor eventually removes inactive destinations [36]. For a 16-core machine, each group predictor entry requires a total of 37 bits (tag not included): 32 bits for the train-up counters and a 5-bit roll-over counter for the train-down purposes. For SP-prediction, we consider an SP-table with two signatures per entry (total of 33 bits) as a fair setting for comparison. Note that SP-prediction requires also a set of communication counters (1-byte each) and a predictor register, which account for a fixed cost of 17 bytes per core.

Each predictor represents a point in the trade-off between latency and bandwidth. To effectively visualize this trade-off, we plot results on a two dimensional plane (Figures 12, 13). The horizontal dimension represents *request bandwidth per miss* (as additional to that of the based directory). The vertical dimension represents *latency*, measured as the *percent of misses that require indirection*. The chosen metrics provide a desirable level of detail for deriving insightful results for the performance of the predictors under consideration.

Figure 12 displays the results for the four predictors in four different applications for illustration. The results assume predictors with an infinite number of table entries for their indexed tables, i.e., they do not consider space efficiency. Overall, SP-prediction lays in the trade-off plane comparably to address- and instruction-based prediction. Among the examples, fmm presents a case in which SP-prediction outperforms all other predictors, achieving both higher accuracy and lower bandwidth. In contrast, dedup presents a counter case, where SP-prediction is weaker on the accuracy dimension. Accuracy levels between ADDR and INST appear to be similar, with the ADDR-predictor having more tendency towards lower bandwidth requirements. UNI-prediction is shown to have lower accuracy, which also negatively affects the bandwidth demands since incorrect predictions place unnecessary messages on the interconnect.

Each scheme has, however, a very different space demands to meet the illustrated maximum performance. A perfect ADDR-prediction scheme suggests storage requirements in proportion to the size of the memory blocks, which is prohibitively large. Common practice is for ADDR to consider, instead, predictors per macro block (e.g., 256-bytes in our implementation). This reduces the maximum space requirements, and also improves further the predictor by capturing spatial locality. However, even with macro-blocks, the number of entries required to achieve the maximum performance is in the order of Kilo. INST has been promoted for its low storage needs; however, it requires significantly more table entries than the SP-table (equal to static load/stores). UNI-prediction requires only a single prediction entry and represents the cheapest possible solution. The SP-prediction's storage requirements are inherently bounded by the

[4]Other prediction policies such as "owner" or "group/owner" can also be used and fairly compared as far as all predictors are tuned to the same base policy.

number of static sync-points of the application as shown in Table 1. This corresponds to substantially lower space demands compared to ADDR and INST. Assuming that the SP-table is easily implementable in the software layer, its hardware space requirements can be largely eliminated, reaching those of UNI-predictor.

To evaluate the sensitivity of the predictors to space requirements, we implement them with limited number of table entries. Figure 13 compares the performance of different predictors when table entries go from unlimited to a finite number of 512 ($\sim$4KB of storage space). To simplify the illustration, we show only the average results for each predictor, over all the studied applications. The results indicate that limited space yields lower accuracy for ADDR and INSTR compared to SP-prediction. Nonetheless, they present a corresponding decrease in bandwidth, since prediction is attempted on fewer misses.

The prediction performance per space requirements is in a sense the measure of how well the prediction information is encoded, or in other words, the measure of a predictor's space efficiency and cost. Considering that SP-prediction requires significantly smaller storage than ADDR and INST, we argue that the reported small performance differences are insignificant when space and power requirements are a primary design constraint, as is the clear case in modern and emerging CMP implementations [19]. In conclusion, from the space requirements perspective, an SP-predictor with $\sim$256 entries can achieve performance equivalent to INST with $\sim$1K entries, or macro-block ADDR with $\sim$8K entries, on average.

5.5. Discussion

Predictor's power consumption comparison. Prediction tables consume static and dynamic power. Static power is proportional to the table size, which is substantially smaller with SP-prediction. Dynamic power is primarily affected by the associativity, and the access frequency of the predictor tables. While the ADDR and INST access their tables on every miss, SP-predictor keeps the prediction set in a single register, and accesses the SP-table for updates only on sync-points. This directly translates into power savings. Based on an overall observation, the SP-table would be accessed once for every $\sim$300 accesses of an ADDR- or INST-based table.

Thread migration. So far we have assumed that communication signatures and predictors consist of bit vectors representing target physical cores. If thread movements are allowed between cores, then those representations should track a "logical core-ID" (e.g., thread-id) rather than physical ID. The logical-to-physical destination mapping must be known at the core side, and could be applied before or after the formation of the predictor, depending on the coherence controller implementation.

Projections for commercial workloads. Database, server, and OS workloads are mostly based on lock synchronization and as a result have less regular and predictable communication patterns [42]. The proposed SP-predictor can effectively predict the communication activity within critical sections since it can retrieve communication signatures on lock points that include the cores (or the sequence of cores) holding the lock previously in time. Results from applications with a high count of critical sections (e.g., fluidanimate and water-ns) show high prediction accuracy for the misses occurring within critical sections (Figure 7). Therefore, although we have not performed experiments on such workloads, we expect our predictor to work reasonably well.

6. Related Work

Address and instruction-based indexing have been the basis of hardware coherence predictors [27,28,31,39]. In the context of destination set prediction, Acacio et al. [2] studied a two-level owner predictor where the first level decides whether to predict an owner and the second level decides which node might be the owner. In a similar work, they study a single-level design to predict sharers for an upgrade request [3]. Bilir et al. [8] studied multicast snooping using a "Sticky Spatial" predictor. Martin et al. [36] explored different policies for destination set predictors to improve the latency/bandwidth trade-off under ordered interconnects. Other studies have further explored the impact of predictor caches [40] and perceptron-based predictors [34].

There have been numerous other efforts to improve coherence performance. Many protocols were developed or extended to optimize for specific sharing patters, such as pairwise sharing [22], migratory sharing [13, 44], producer-consumer sharing [11] and some mix of those [20]. Dynamic self-invalidation was proposed to eliminate the invalidation overhead [31, 33]. Alternatively, software-driven approaches have proposed programming models or utilized compilers to effectively *prefetch* or *forward* shared data to reduce miss latencies [1, 29, 47]. A thorough characterization of data sharing patterns and inter-processor communication behavior in emerging workloads is presented in a work by Barrow et al. [5].

More recent work has exploited properties relevant to CMP architectures to accelerate coherence, such as core proximity and fast and flexible on-chip interconnect. Brown et al. [9] describe an extension to the directory-based coherence protocol where requests are first sent to neighboring cores. Barrow et al. [6] propose adding new dedicated links for forwarding the requests to the nearby caches, delegating directory functions in case of proximity hits. Various other proposals, such as Token Coherence [37], examine novel approaches on maintaining coherence in unordered interconnects without requiring directory indirection. Eisley et al. [16] propose to embed directories within the network routers that manage and steer requests towards nearby sharers. Jerger et al. [18] propose a virtual tree structure to maintain coherence in an unordered interconnect, with the root of the tree acting as an ordering point for requests. In Circuit-Switch Coherence [25], the same authors show how coherence predictors can leverage existing circuits to optimize pairwise sharing between cores. Similar to virtual tree coherence, DiCo-CMP [41] delegates directory responsibilities to the owner caches.

Synchronization points have also been utilized by other recently proposed techniques to direct hardware-level optimization. In BarrierWatch [14], the authors identify the relation between barriers and time-varying program behavior and propose the use of this relation to guide run-time optimizations in CMP architectures. Under the MPI model, Ioannou et al. [24] propose tracking MPI calls to guide phase-based power management in Intel's MPI Cloud processor research prototype. In heterogeneous architectures, locks and other synchronization points may trigger scheduling/migration actions to accelerate critical sections [45] and other critical bottlenecks [26]. Work on memory scheduling for parallel applications has also made use of loop-based synchronization to effectively manage inter-thread DRAM interference [15]. Lastly, exposing shared-memory synchronization primitives to the hardware has been the underlying support for software based coherence enforcement, e.g., [10].

7. Conclusions

This paper proposed and studied *Synchronization Point based Coherence Prediction* (SP-Prediction), a novel run-time technique for predicting communication destinations of misses in cache-coherent shared-memory systems. SP-prediction employs mechanisms that capture synchronization points at run time, track the communication

activity between them, and extract simple communication signatures that guide target prediction for future misses. SP-prediction is substantially simpler than existing techniques because it exploits the inherent characteristics of an application to predict communication patterns. Compared with address- and instruction-based predictors, SP prediction requires smaller area and consumes less energy while achieving comparative high accuracy. We anticipate that the synchronization point driven prediction approach could be applicable to further communication optimization cases, and this work will be basis for future investigation towards this direction.

Acknowledgments

We thank our shepherd Prof. Milos Prvulovic, members of Pitt's XCG (formerly CAST) group, and the anonymous reviewers for their constructive comments and suggestions. This work was supported in part by the US NSF grants: CCF-1064976, CCF-1059283 and CNS-1012070.

References

[1] H. Abdel-Shafi *et al.*, "An evaluation of fine-grain producer-initiated communication in cache-coherent multiprocessors," in *Proc. of the 3rd IEEE Symp. on High-Performance Computer Architecture*, 1997.

[2] M. E. Acacio *et al.*, "Owner prediction for accelerating cache-to-cache transfer misses in a CC-NUMA architecture," in *Proc. of Conf. on Supercomputing*, 2002.

[3] ——, "The use of prediction for accelerating upgrade misses in CC-NUMA multiprocessors," in *Proc. Int'l Conf. on Parallel Architectures and Compilation Techniques*, 2002.

[4] A. Banerjee *et al.*, "An energy and performance exploration of network-on-chip architectures," *IEEE Trans. Very Large Scale Integr. Syst.*, 2009.

[5] N. Barrow-Williams *et al.*, "A communication characterisation of SPLASH-2 and PARSEC," in *Proc. Int'l Symp. on Workload Characterization*, 2009.

[6] ——, "Proximity coherence for chip multiprocessors," in *Proc. Int'l Conf. on Parallel Aarchitectures and Compilation Techniques*, 2010.

[7] C. Bienia *et al.*, "The PARSEC benchmark suite: characterization and architectural implications," in *Proc. Int'l Conf. on Parallel Architectures and Compilation Techniques*, 2008.

[8] E. E. Bilir *et al.*, "Multicast snooping: a new coherence method using a multicast address network," in *Proc. Int'l Symp. on Computer Architecture*, 1999.

[9] J. A. Brown *et al.*, "Proximity-aware directory-based coherence for multi-core processor architectures," in *Proc. Int'l Symp. on Parallel Algorithms and Architectures*, 2007.

[10] J. B. Carter *et al.*, "Implementation and performance of munin," in *Proc. Int'l Symp. on Operating Systems Principles*, 1991.

[11] L. Cheng *et al.*, "An adaptive cache coherence protocol optimized for producer-consumer sharing," in *Proc. of the Int'l Symp. on High Performance Computer Architecture*, 2007.

[12] S. Cho and L. Jin, "Managing distributed, shared l2 caches through os-level page allocation," in *Proc. Int'l Symp. on Microarchitecture*, 2006.

[13] A. L. Cox and R. J. Fowler, "Adaptive cache coherency for detecting migratory shared data," in *Proc. of the 20th Int'l Symp. on Computer Architecture*, 1993.

[14] S. Demetriades and S. Cho, "Barrierwatch: characterizing multithreaded workloads across and within program-defined epochs," in *Proc. of the 8th ACM Int'l Conf. on Computing Frontiers*, 2011.

[15] E. Ebrahimi *et al.*, "Parallel application memory scheduling," in *Proc. of the 44th Annual IEEE/ACM Int'l Symp. on Microarchitecture*, 2011.

[16] N. Eisley *et al.*, "In-network cache coherence," in *Proc. Int'l Symp. on Microarchitecture*, 2006.

[17] M. Ekman *et al.*, "TLB and snoop energy-reduction using virtual caches in low-power chip-multiprocessors," in *Proc. of the 2002 Int'l Symp. on Low power electronics and design*, 2002.

[18] N. D. Enright Jerger *et al.*, "Virtual tree coherence: Leveraging regions and in-network multicast trees for scalable cache coherence," in *Proc. Int'l Symp. on Microarchitecture*, 2008.

[19] H. Esmaeilzadeh *et al.*, "Dark silicon and the end of multicore scaling," in *Proc. of the 38th annual Int'l Symp. on Computer architecture*, 2011.

[20] H. Hossain *et al.*, "Improving support for locality and fine-grain sharing in chip multiprocessors," in *Proc. Int'l Conf. on Parallel Architectures and Compilation Techniques*, 2008.

[21] http://quid.hpl.hp.com:9081/cacti/, "CACTI 5.3."

[22] IEEE Computer Society, "IEEE standard for scalable coherent interface (SCI)." 1992.

[23] Intel Co., "MESIF protocol," uS Patent 6922756.

[24] Ioannou and et al, "Phase-based application-driven hierarchical power management on the single-chip cloud computer," in *Proc. of the Int'l Conf. on Parallel Architectures and Compilation Techniques*, 2011.

[25] N. D. E. Jerger *et al.*, "Circuit-switched coherence," in *IEEE 2nd Network on Chip Symp.*, 2008.

[26] J. A. Joao *et al.*, "Bottleneck identification and scheduling in multithreaded applications," in *Proc. of the Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 2012.

[27] S. Kaxiras and C. Young, "Coherence communication prediction in shared-memory multiprocessors," in *Proc. Int'l Symp. on High-Performance Computer Architecture*, 2000.

[28] S. Kaxiras and J. Goodman, "Improving CC-NUMA performance using instruction-based prediction," in *Proc. Int'l Symp. on High-Performance Computer Architecture*, 1999.

[29] D. A. Koufaty *et al.*, "Data forwarding in scalable shared-memory multiprocessors," in *Proc. Int'l Conf. on Supercomputing*, 1995.

[30] A. Lai and B. Falsafi, "Memory sharing predictor: The key to a speculative coherent DSM," in *Proc. Int'l Symp. on Computer Architecture*, 1999.

[31] ——, "Selective, accurate, and timely self-invalidation using last-touch prediction," in *Proc. Int'l Symp. on Computer Architecture*, 2000.

[32] J. Laudon and D. Lenoski, "The SGI Origin: A CC-NUMA highly scalable server," in *Proc. Int'l Symp. on Computer Architecture*, 1997.

[33] A. R. Lebeck and D. A. Wood, "Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors," in *Proc. Int'l Symp. on Computer Architecture*, 1995.

[34] S. Leventhal and M. Franklin, "Perceptron based consumer prediction in shared-memory multiprocessors," in *Int'l Conf. on Computer Design*, 2006.

[35] P. S. Magnusson *et al.*, "Simics: A full system simulation platform," *IEEE Computer*, 2002.

[36] M. M. K. Martin *et al.*, "Using destination-set prediction to improve the latency/bandwidth tradeoff in shared-memory multiprocessors," in *Proc. Int'l Symp. on Computer Architecture*, 2003.

[37] ——, "Token coherence. decoupling performance and correctness," in *Proc. of the 30th Annual Int'l Symp. on Computer Architecture*, 2003.

[38] A. Moshovos, "Regionscout: Exploiting coarse grain sharing in snoop-based coherence," in *Proc. Int'l Symp. on Computer Architecture*, 2005.

[39] S. Mukherjee and M. Hill, "Using prediction to accelerate coherence protocols," in *Proc. Int'l Symp. on Computer Architecture*, 1998.

[40] J. Nilsson *et al.*, "The coherence predictor cache: a resource-efficient and accurate coherence prediction infrastructure," in *Proc. of the Int'l Parallel and Distributed Processing Symp.*, 2003.

[41] A. Ros *et al.*, "A direct coherence protocol for many-core chip multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, 2010.

[42] S. Somogyi *et al.*, "Memory coherence activity prediction in commercial workloads," in *Workshop on Memory Performance Issues*, 2004.

[43] D. J. Sorin *et al.*, "Specifying and verifying a broadcast and a multicast snooping cache coherence protocol," *IEEE Transactons on Parallel and Distributed Systems*, 2002.

[44] P. Stenström *et al.*, "An adaptive cache coherence protocol optimized for migratory sharing," in *Proc. of the Int'l Symp. on Computer Architecture*, 1993.

[45] M. A. Suleman *et al.*, "Accelerating critical section execution with asymmetric multi-core architectures," in *Proc. of the Int'l Conf. on Architectural Support for Programming Languages and Op. Syst.*, 2009.

[46] Tilera Co. and http://www.tilera.com, "Tilera TILE64 processor."

[47] P. Trancoso and J. Torrellas, "The impact of speeding up critical sections with data prefetching and forwarding," in *Proc. Int'l Conf. on Parallel Processing*, 1996.

[48] S. C. Woo *et al.*, "The SPLASH-2 programs: characterization and methodological considerations," in *Proc. Int'l Symp. on Computer Architecture*, 1995.