Appears in the proceedings of the
30th Annual International Symposium on Computer Architecture (ISCA-30)
San Diego, CA, June 9-11, 2003

Using Destination-Set Prediction to Improve the Latency/Bandwidth Tradeoff in Shared-Memory Multiprocessors

Milo M. K. Martin, Pacia J. Harper, Daniel J. Sorin[‡], Mark D. Hill, and David A. Wood

Computer Sciences Department
University of Wisconsin-Madison

[‡]*Department of Electrical and Computer Engineering*
Duke University

http://www.cs.wisc.edu/multifacet/

Abstract

Destination-set prediction can improve the latency/bandwidth tradeoff in shared-memory multiprocessors. The destination set is the collection of processors that receive a particular coherence request. Snooping protocols send requests to the maximal destination set (i.e., all processors), reducing latency for cache-to-cache misses at the expense of increased traffic. Directory protocols send requests to the minimal destination set, reducing bandwidth at the expense of an indirection through the directory for cache-to-cache misses. Recently proposed hybrid protocols trade-off latency and bandwidth by directly sending requests to a predicted destination set.

This paper explores the destination-set predictor design space, focusing on a collection of important commercial workloads. First, we analyze the sharing behavior of these workloads. Second, we propose predictors that exploit the observed sharing behavior to target different points in the latency/bandwidth tradeoff. Third, we illustrate the effectiveness of destination-set predictors in the context of a multicast snooping protocol. For example, one of our predictors obtains almost 90% of the performance of snooping while using only 15% more bandwidth than a directory protocol (and less than half the bandwidth of snooping).

1 Introduction

In a cache-coherent shared-memory multiprocessor, the *destination set* is the collection of processors (or nodes) that receive a particular coherence request. Destination-set size represents a key factor in the trade-off between latency and bandwidth in a multiprocessor system. Directory protocols first send all requests to a directory (often co-located with memory) that forwards the request as needed to the owner and/or sharers of the block. This approach conserves bandwidth, but it adds indirection

This work is supported in part by the National Science Foundation (EIA-9971256, EIA-0205286, CDA-9623632, and CCR-0105721), a Norm Koo Graduate Fellowship and an IBM Graduate Fellowship (Martin), a Los Alamos Computer Science Institute Fellowship (Harper), an Intel Graduate Fellowship (Sorin), a Warren Faculty Scholarship (Sorin), Spanish Secretaría de Estado de Educación y Universidades (Hill sabbatical), two Wisconsin Romnes Fellowships (Hill and Wood), and donations from Intel Corporation, IBM, and Sun Microsystems.

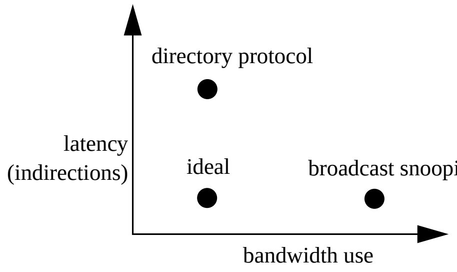


Figure 1. Tradeoff Between Latency and Bandwidth

latency to *cache-to-cache misses*[1] that must be serviced from other caches. Traditional snooping systems use a maximal destination set of all processors, since every coherence request is broadcast to all processors. Broadcasting optimizes cache-to-cache miss latency by eliminating indirection, but it requires (end-point) bandwidth proportional to the square of the number of processors.

As illustrated in Figure 1, system designers must choose between the high bandwidth use of snooping protocols and the high sharing latency of directory protocols. An ideal protocol would directly send requests to only those processors that need to observe them, combining the low latency of broadcast snooping with the bandwidth efficiency of a directory protocol. This latency/bandwidth tradeoff is especially important for the commercial workloads that dominate the current use of multiprocessor servers, since many of these workloads exhibit high cache miss rates and a large fraction of cache-to-cache misses [5, 18, 30].

One emerging approach for improving this latency/bandwidth trade-off is *destination-set prediction*. Multicast Snooping [7] reduces bandwidth compared to broadcast snooping, by multicasting a coherence request to a predicted destination set. If the destination set is sufficient (*e.g.*, includes the processor or memory module that currently owns the block), the request avoids indirection (like all requests in a broadcast snooping system). Similarly, Acacio *et al*. add prediction to a conventional directory protocol, converting some misses from three interconnection network hops to only two [1, 2]. Moreover, the

1. Cache-to-cache misses are also called dirty misses, 3-hop misses, or cache-to-cache transfers. Cache-to-cache misses are closely related to sharing misses and are often the result of true or false sharing.

Table 1. Benchmark Descriptions

Static Web Content Serving: Apache. Web servers such as Apache are an important enterprise server application. We use Apache 2.0.39 for SPARC/Solaris 8 configured to use pthread locks and minimal logging at the web server. Our experiments use a repository of 20,000 files (~500 MB) and 160 simulated users (10 per processor). The system is warmed up for 1.6 million requests, and results are based on runs of 5,000 requests.
Scientific Applications: Barnes-Hut and Ocean. We selected two applications from the SPLASH-2 benchmark suite [34]: *barnes-hut* with 64k bodies and *ocean* with a 514 x 514 grid. We begin measurement at the start of the parallel phase to avoid measuring thread forking.
Java Server Workload: SPECjbb. SPECjbb2000 is a server-side Java benchmark that models a 3-tier system, focusing on the middleware server business logic. We use Sun's HotSpot 1.4.0 Server JVM, and our experiments use 24 driver threads and 24 warehouses (with a data size of approximately 500MB). The system is warmed up for 100,000 transactions, and our results are based on runs of 100,000 transactions.
Online Transaction Processing (OLTP): DB2 with a TPC-C-like workload. The TPC-C benchmark models the database activity of a wholesale supplier. Our OLTP workload is based on the TPC-C v3.0 benchmark using IBM's DB2 v7.2 EEE database management system. Our experiments use 128 simulated users (8 per processor) that access an 800MB database with 4,000 warehouses stored on five raw disks. The database is warmed up for 10,000 transactions before taking measurements, and our results are based on runs of 1,000 transactions.
Dynamic Web Content Serving: Slashcode. Our Slashcode benchmark is based on an open-source dynamic web message posting system used by slashdot.org. We use Slashcode 2.0, Apache 1.3.20 and Apache's `mod_perl` 1.25 module for the web server, and we use MySQL 3.23.39 as the database engine. A multithreaded user emulation program simulates browsing and posting behavior of 48 users (3 per processor). The database contains 3,000 messages. The system is warmed up for 240 transactions before taking measurements, and our results are based on runs of 100 transactions.

recently-proposed Token Coherence protocol [23] allows systems to implement destination-set prediction without requiring a totally-ordered interconnect [7] or introducing difficult-to-debug protocol races [1, 2, 7].

This paper is the first to explore several predictors in the destination-set predictor design space and the first to evaluate their effectiveness using commercial workloads. Section 2 explores the potential of destination-set prediction by analyzing the sharing patterns of multiple commercial workloads. We examine the degree of instantaneous sharing, the degree of sharing over time, and the temporal and spatial locality of cache-to-cache misses. In particular, we show that cache-to-cache miss patterns in commercial workloads have substantial temporal and spatial locality that can be captured by destination-set predictors.

Section 3 introduces destination-set predictors inspired by our analysis of sharing patterns, and each predictor targets different points in the latency/bandwidth design space. Proposed predictors send to only the previous owner (emphasizing bandwidth over latency), broadcast if data appear shared (latency over bandwidth), or multicast to recent sharing groups (balancing latency and bandwidth).

In Sections 4 and 5 we evaluate these destination-set predictors using trace-driven and full-system execution-driven timing simulation, respectively. We compare these predictors against a broadcast snooping protocol, a directory protocol, and multicast snooping's original destination-set predictor [7]. For our workloads on 16 processor systems, these results show that:

- destination-set predictors can reduce indirections by up to 90%, with respect to a directory protocol, while using less than one third the request bandwidth of a broadcast snooping system;
- destination-set predictors benefit from aggregating information from spatially-related data by using macroblock indexing (*e.g.*, 1024-byte macroblocks);
- destination-set predictors perform well with relatively few entries (*e.g.*, 8192 entries);
- destination-set prediction can substantially reduce execution time (compared to directories) while greatly reducing bandwidth (versus broadcast snooping).

Section 6 summarizes related work on hybrid protocols and destination-set predictors that has focused on single points in the destination-set predictor design space using scientific workloads (*e.g.*, SPLASH-2 benchmarks [34]).

2 Commercial Workload Sharing Behaviors

In this section, we analyze the sharing behaviors of several commercial workloads. We use this analysis to guide destination-set predictor designs in subsequent sections. Table 1 describes the workloads we use as benchmarks. Due to the growing prevalence of information services in our society, commercial workloads are increasingly important for high performance multiprocessor systems. Thus, we concentrate on commercial workloads, such as database and web servers, but also include two scientific workloads for comparison. We refer interested readers to Alameldeen *et al.* [3] for more detailed description and characterization of these commercial workloads. We begin by presenting our methodology and general characteristics of our workloads before delving into detailed sharing behavior analysis.

2.1 Methodology

We use Simics [22] to perform full-system simulation of systems running commercial workloads. The simulated machine is a 16-processor SPARC system running unmodified Solaris 8. For the results in this section and Section 4, we collected traces of second-level cache misses for our workloads by running simulations with a MOSI coherence protocol (the simulated target system is described in Section 5.1). We use the first one million misses in the

Table 2. Workload Properties

Workload	Memory touched (64 byte blocks)	Memory touched (1024 byte blocks)	Static instrs that cause L2 misses	Total L2 misses	L2 misses per 1,000 instructions	Directory indirections
Apache	46 MB	71 MB	18,745	22 M	5.9	89%
Barnes-Hut	11 MB	13 MB	7,912	3 M	0.4	96%
Ocean	52 MB	61 MB	11,384	5 M	0.5	58%
OLTP	57 MB	125 MB	21,921	18 M	7.0	73%
Slashcode	181 MB	316 MB	42,770	13 M	1.0	35%
SPECjbb	341 MB	558 MB	24,023	21 M	3.3	41%

trace to warm up the caches (and later our destination-set predictors). For each coherence request, trace records contain the data address, program counter (PC) address, requester, and request type.

Traces allow for quick workload characterization and exploration of the predictor design space and enable deterministic and precise comparisons. However, traces capture neither the effects of timing races nor their impact on overall performance. In Section 5 we address these limitations by presenting execution-driven timing results from full-system simulations using a detailed performance model.

2.2 General Properties

Studies of multiprocessor commercial workloads and their properties have found that second-level (L2) cache misses, especially misses due to sharing, can dominate performance. Table 2 shows that our commercial workloads have large data footprints, in terms of total memory touched in 64-byte blocks (column 2, second from the left) and 1024-byte macroblocks (column 3), and a large number of static instructions that cause cache misses (column 4). The commercial workloads have relatively high cache miss rates from a 4 MB, 4-way set associate L2 cache (columns 5 and 6). The rightmost column in the table (column 7) lists the percent of L2 cache misses that would cause indirections in a directory protocol. As discussed in the next section, these workloads have a larger percentage of indirections, providing ample opportunity for destination-set prediction to improve their performance.

2.3 Cache-to-Cache Misses

Prior studies have shown that commercial workloads incur a large fraction of cache-to-cache misses [5, 18, 30]. Our results, shown in the rightmost column of Table 2 (column 7), corroborate these previous results by finding that 35-96% of all L2 cache misses for our commercial workloads would suffer from indirection in a directory protocol. The high miss rate and high rate of indirections in commercial workloads provide sufficient opportunity for destination-set prediction. While Barnes-Hut and Ocean also incur a large fraction of indirections, their low miss rates result in lower rates of directory indirections *per instruction*. Thus, workload analysis based on Barnes-Hut and Ocean alone would be a poor basis for designing commercial servers.

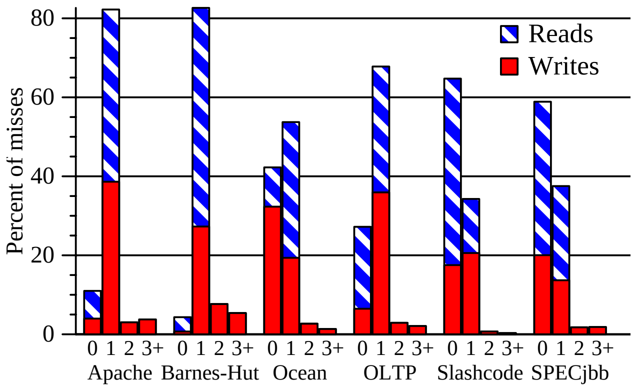


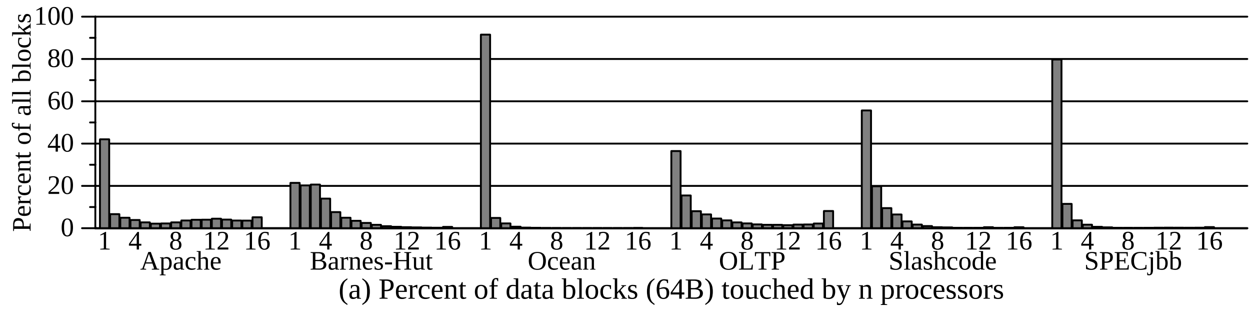
Figure 2. Sharing Histogram. This graph shows the number of processors that must see an indirection in a directory protocol for read and write requests.

2.4 Instantaneous Sharing

While the majority of misses for our workloads cannot be completed without contacting at least one other processor, the number of misses that need to contact many processors is relatively small. For example, at most one other processor (the owner) needs to observe a request to obtain a read-only copy of a block. Figure 2 shows the percentage of requests that need to contact various numbers of processors. For our workloads, many requests require directory indirections, but only about 10% of all requests need to be sent to more than one other processor. This result highlights the inefficiency of broadcast snooping, in which all processors in the system observe all requests.

2.5 Degree of Sharing

While the instantaneous number of processors that need to observe a request is small, the number of processors that read or write a block during the execution is larger. In Figure 3(a), we plot a histogram of the number of unique processors that access a block at least once during the execution. The data show a non-uniform distribution; most of the blocks are touched by only one processor. In Figure 3(b), we weight each block by the number of misses (*i.e.*, if a block had ten misses and four processors accessed it, we add ten to the four-processor bin of the histogram). The scaled data show that most of the misses are concentrated on the small number of blocks that are accessed by most or all of the processors. Ocean is an exception; the majority of its misses are to blocks that have been touched by four or fewer processors, a direct result of its column-blocked stencil structure [34].



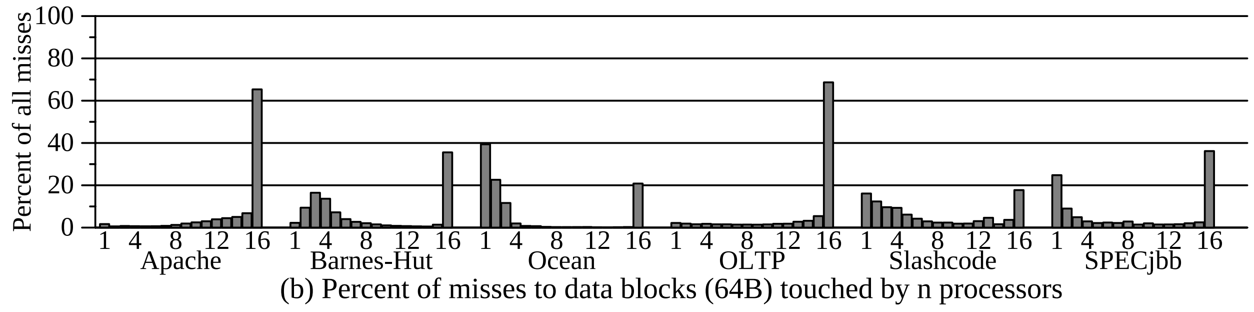


Figure 3. Number of blocks touched by various numbers of processors during execution. Part (a) shows a histogram with one entry for each unique block (64B). In part (b), the data is weighted by the number of misses to the block.

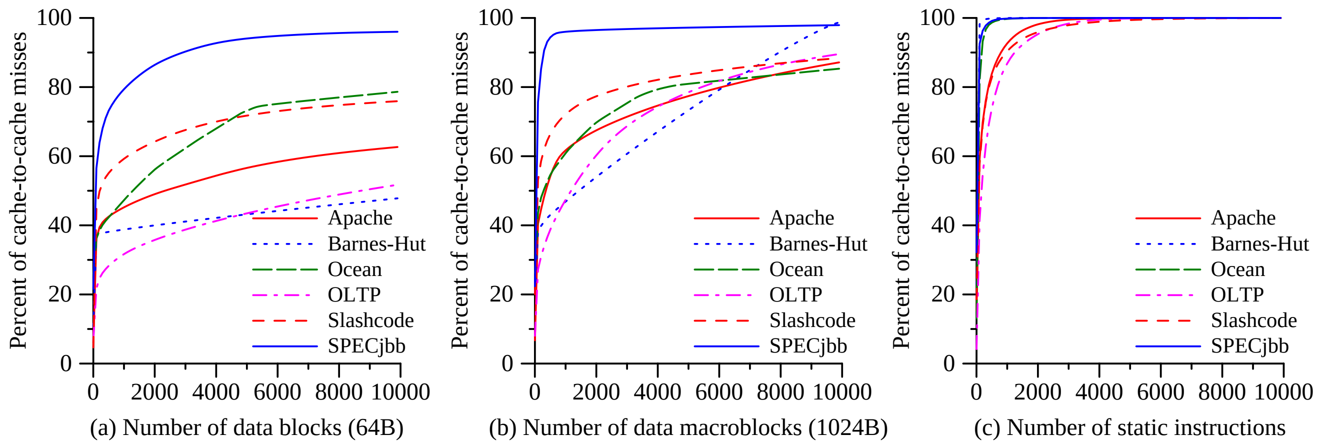


Figure 4. Sharing Locality

2.6 Sharing Locality

Our workloads exhibit a high degree of locality among cache-to-cache misses. Figure 4(a) shows the cumulative distribution of cache-to-cache misses for 64-byte data blocks. These data show, for example, that the hottest 1,000 data blocks in SPECjbb account for 80% of all cache-to-cache misses. Figure 4(b) shows the distribution of cache misses for 1024-byte macroblocks (*i.e.*, aligned regions of 16 64-byte cache blocks), and we observe even more locality. For all of our workloads, the 10,000 hottest macroblocks account for over 80% of all cache-to-cache misses. Figure 4(c) shows the cumulative distribution of unique instructions that cause cache-to-cache misses. These figures reveal significant amounts of temporal and spatial locality in the cache-to-cache miss stream, a result that corroborates prior work [18, 30]. Predictors that exploit the locality in data blocks or instructions (unique PCs) can capture the sharing working sets of these workloads without requiring prohibitive storage.

3 Destination-Set Prediction

Destination-set predictors exploit sharing patterns to guess which processors must observe a particular coherence request. For MOESI write-invalidate protocols, a request to read (*i.e.*, *request for shared)* must find the current owner, while a request to write (*i.e.*, *request for exclusive)* must find the owner and all sharers. With accurate destination-set prediction, a hybrid protocol can use bandwidth comparable to directory-based systems while achieving the low cache-to-cache miss latency of snooping.

Table 3. Predictor Policies

Name		Owner	Broadcast-If-Shared	Group
Entry Structure		*Owner* ID and *Valid* bit	2-bit saturating counter, *Counter*	N 2-bit saturating counters, *Counters[0..N-1]* 5-bit saturating *RolloverCounter*
Entry Size		$\log_2 N$ bits + 1 bit + tag (approximately 4 bytes)	2 bits + tag (approximately 4 bytes)	2N bits + 5 bits + tag (approximately 8 bytes)
Prediction Action (for Shared or Exclusive)		If *Valid*, predict *Owner* Otherwise, minimal set	If *Counter* > 1, broadcast Otherwise, minimal set	For each processor *n*, if *Counters[n]* > 1, add *n* to minimal set
Training Action	**Data Response**	If response from memory, clear *Valid*. Else, set *Owner* to responder, and set *Valid*	If response from memory, decrement *Counter*. Else, increment *Counter*	If response not from memory, increment *Counters[responder]*. Increment *RolloverCounter*†
	Other Request (Exclusive)	Set *Owner* to requester and set *Valid*	Increment *Counter*	Increment *Counters[requester]*. Increment *RolloverCounter*†
	Other Request (Shared)	ignore		

N is the number of processor nodes in the system. †If *RolloverCounter* rolls over, decrement *Counter[i]* for all *i*.

Predictor design involves a trade-off between accuracy (latency) and bandwidth. Predicting too many processors increases bandwidth usage with no increase in accuracy (decrease in latency). Predicting too few processors may reduce bandwidth (depending upon the protocol specifics), but it decreases accuracy (increases latency). Snooping and directory protocols are effectively the two extremes: snooping always predicts broadcast (perfect accuracy, but high bandwidth usage), while directory protocols always initially "predict" the minimal destination set and rely upon the directory to forward the request as necessary (low bandwidth usage, but low accuracy). In this section, we present a predictor framework and a set of policies that target different points in this design space.

3.1 Predictor Model

Each L2 cache controller in the system contains a destination-set predictor. Since only coherence controllers are responsible for interacting with the predictor, we require no modifications to the processor core, but in Section 3.4 we explore an optional enhancement of exporting the program counter of an instruction that misses. Predictors are tagged, set-associative, and (by default) indexed by data block address. The coherence controller performs the predictor access in parallel with the cache access. In the event of a cache miss, the controller uses the predicted destination set when initiating the resulting coherence transaction. If the predictor hits, it generates a prediction according to the policies discussed below. On a predictor miss, the predictor returns by default the *minimal destination set* (*e.g.*, depending on the specific protocol, the set might include only the home module of the block).

Since a small set of data blocks account for most cache-to-cache misses (recall Figure 4), the predictor can improve its effective capacity by allocating predictor entries only for blocks likely to be shared. In our experiments, the predictor allocates an entry only if the minimal destination set proves insufficient to directly locate the requested block.

3.2 Training Information

The policies we discuss use two types of training cues to predict sharing behavior: external coherence requests and coherence responses. In both cases, the predictor learns the identity of one or more other processors that have recently accessed a block. On external coherence requests, the predictor automatically receives the requesting processor's identity (since this information is required to permit a response). For responses, we extend data-response messages to include the sender's identity. Specific policies, described next, use this information either to "train up" or "train down", *i.e.*, increase or decrease the destination set.

3.3 Prediction Policies

Different prediction policies can use some or all of the training information to target different points in the bandwidth/latency spectrum. This section describes three general policies, specified in Table 3, and one hybrid policy.

The *Owner* predictor. *Owner* targets scenarios in which either (a) only one other processor needs to be in the destination set (*e.g.*, pairwise sharing) or (b) bandwidth is limited. The predictor records the last processor to invalidate or respond with a block. On a prediction, the predictor returns the union of the predicted owner and the minimal destination set. *Owner* works well for pairwise sharing, because both processors include each other in their predictions. *Owner* also works well under limited bandwidth because it sends at most one additional control message for each request, independent of the number of processors in the system.

The *Broadcast-If-Shared* predictor. *Broadcast-If-Shared* targets scenarios in which either (a) most shared data are widely shared, (b) most data are not shared, or (c) bandwidth is plentiful. *Broadcast-If-Shared* selects either a destination set that includes all processors (if the block is predicted shared) or the minimal destination set (otherwise). A two-bit saturating counter—incremented on requests and responses from other processors and decremented otherwise—determines which prediction to make. *Broadcast-If-Shared* performs comparably to snooping, but it uses less bandwidth by not broadcasting all requests.

The *Group* predictor. *Group* targets scenarios in which (a) groups of processors (less than all processors) share blocks and (b) bandwidth is neither extremely limited nor plentiful. Each predictor entry contains a two-bit counter per processor in the system. On each request or response, the predictor increments the corresponding counter. *Group* also increments the entry's 5-bit *rollover counter*; on overflow, the predictor decrements all 2-bit counters in the entry. This training-down mechanism ensures that the predictor eventually removes inactive processors (*i.e.*, processors no longer accessing the block) from the destination set. *Group* should work well on a large multiprocessor in which not all processors are working on the same aspect of the computation or if the system is logically partitioned.

The *Owner/Group* hybrid predictor. *Owner/Group* targets (a) stable sharing patterns and (b) more limited bandwidth than *Group*. *Owner/Group* uses a *Group* predictor to handle requests for exclusive and an *Owner* predictor to handle requests for shared. This policy works well for stable sharing patterns because all processors in the sharing set observe all requests for exclusive, and thus they can track the current owner in most cases. Thus, requests for shared can be sent only to the current predicted owner, reducing the bandwidth demand.

3.4 Alternative Indexing

By default, the predictors use data-block address indexing, but we also explore program counter (PC) indexing and "coarse-grain" macro-block indexing.

Program counter indexing. Figure 4(c) showed that a small number of static instructions cause most cache-to-cache misses. This observation, supported by prior work (*e.g.*, [16]), suggests that we index the predictor with the PC. To enable this indexing alternative, the processor supplies the PC of the load or store instruction causing the miss. The cache controller includes this PC in the coherence request (extending the message format) and remembers the PC until the coherence response arrives (used for predictor training).

Macroblock indexing. Figure 4(b) showed that cache-to-cache misses exhibit significant spatial locality. For example, consider a processor reading a large buffer that was recently written by another processor. The last processor to write the buffer may be difficult to predict; however, once a processor observes that several data blocks of the buffer were supplied by one processor, a macroblock-based predictor can learn to find other spatially related blocks at that same processor. To exploit this spatial locality, we index the predictor with macroblock addresses by simply dropping the least significant bits. Macroblock indexing also increases the effective reach of the predictor, thereby reducing pressure on the predictor's capacity.

3.5 Prior Work: Sticky-Spatial(1)

We compare our predictors to a variant of the original multicast snooping predictor developed by Bilir *et al*. [7]. The *Sticky-Spatial(1)* predictor is "sticky" because it only trains up, relying on predictor replacements to reduce the destination-set size. It is "spatial" because it aggregates information from neighboring predictor entries (restricting it to a direct-mapped implementation). *Sticky-Spatial* trains up by observing responses and retries from the memory controller (described in Section 4.1).

Our predictors improve upon *Sticky-Spatial* in two important ways. First, macroblock addressing captures spatial locality with a single entry. This approach reduces pressure on finite predictors, allows set-associative implementations, and eliminates aliasing (*Sticky-Spatial* ignores the tag when making predictions). Second, all of our predictors have explicit mechanisms to train down.

4 Evaluation of Destination-Set Predictors

This section summarizes the multicast snooping protocol we evaluate, describes our method of analyzing the latency/bandwidth tradeoff, and presents trace-driven results for our prediction policies using the methodology described earlier in Section 2.1.

4.1 Multicast Snooping Protocol

To evaluate our destination-set predictors in a concrete context, we implemented them as part of a multicast snooping system [7, 32]. Processors in this system act like they do in broadcast snooping, except that processors multicast coherence requests to a predicted destination set (called a *multicast mask* in the original paper). To enforce the necessary ordering requirements, requests are sent on a totally-ordered interconnect and the minimal destination set includes both the requester and the home node for the requested block. The home node maintains a directory structure to track the owner and sharers of each block, allowing it to detect if a request was *sufficient* (*i.e.*, sent to all necessary processors). A destination set is sufficient in multicast snooping if it includes the requester, the home node, the owner of the block, and, if the request is for write permission, all processors sharing the block.

If a destination set is sufficient, the request is successful. In this case (1) the owner (which could be the memory) responds to the requester with data, (2) the directory updates its state, and (3) if the request was for read/write access, all sharers invalidate their copies of the block.

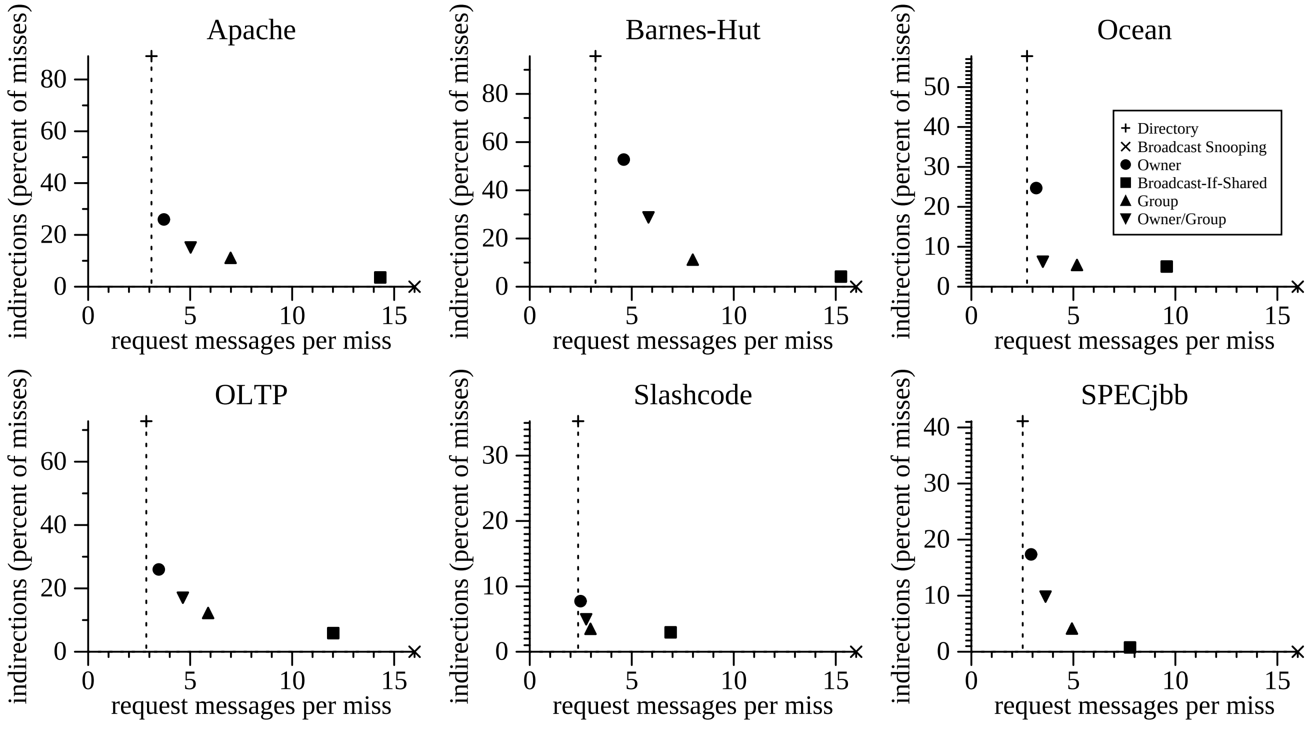


Figure 5. Standout Predictor Results (8,192 entries, 1,024-byte block indexing)

If a destination set is insufficient, the request must be retried. Our implementation uses the optimization proposed by Sorin *et al*. [32], in which the directory re-issues the coherence request with an improved destination set that reflects the current owner and sharers. A reissued request has latency similar to an indirected (3-hop) request in a directory protocol.

As in the original protocol, however, a window of vulnerability exists between the retry's issue and when the interconnection network orders it. During this window, a racing request can intervene, changing the owner and/or sharers such that the retry's destination set is now insufficient. The directory must detect this infrequent race condition and retry the request again. To avoid deadlock and livelock in pathological cases, on the third retry the directory resorts to broadcasting, which is guaranteed to succeed.

4.2 Predictor Evaluation Methodology

Each predictor and base protocol represent one point in the trade-off between latency and bandwidth. To visualize this tradeoff, we plot results on a two dimensional plane. The horizontal dimension represents request bandwidth per miss (*i.e.,* the bandwidth per miss used by requests, forwards, and retries). The vertical dimension represents latency, measured as the percent of misses that require indirection (*i.e.,* three-hop requests in a directory protocol or requests retried by the directory in multicast snooping). The dashed vertical line represents the directory protocol bandwidth, which is the best case for multicast snooping.

We compare our predictors against base snooping and directory protocols. We assume a typical MOSI broadcast snooping protocol (denoted by ✖ in our results) that relies on a totally-ordered broadcast interconnect. We model a bandwidth-efficient MOSI directory protocol (denoted by ✚) based on the AlphaServer GS320 [11]. The GS320 also uses a totally-ordered interconnection network, eliminating the need for explicit acknowledgment messages. Using such a directory protocol allows us to assume the same interconnect configuration for all protocols.

4.3 Predictor Policy Evaluation

Figure 5 displays the results for the four predictor policies (note the different y-axis for each workload). These predictors use 1024-byte macroblock indexing and have 8,192 entries. Since our predictor entry size (shown in Table 3) ranges from approximately four to eight bytes (including tags), the total predictor size ranges from 32kB to 64kB (less than 2% of our L2 cache size).

We find that destination-set prediction provides a favorable bandwidth/latency tradeoff over a range of workloads. The best predictors approach the low latency of a snooping protocol (by substantially reducing indirections), while reducing the request bandwidth by a factor of three. Alternatively, the predictors allow for systems that use bandwidth comparable to a directory protocol while substantially reducing indirections (and hence latency).

***Owner* predictor (denoted by ●).** Figure 5 shows that *Owner* achieves its goal of reducing indirections while

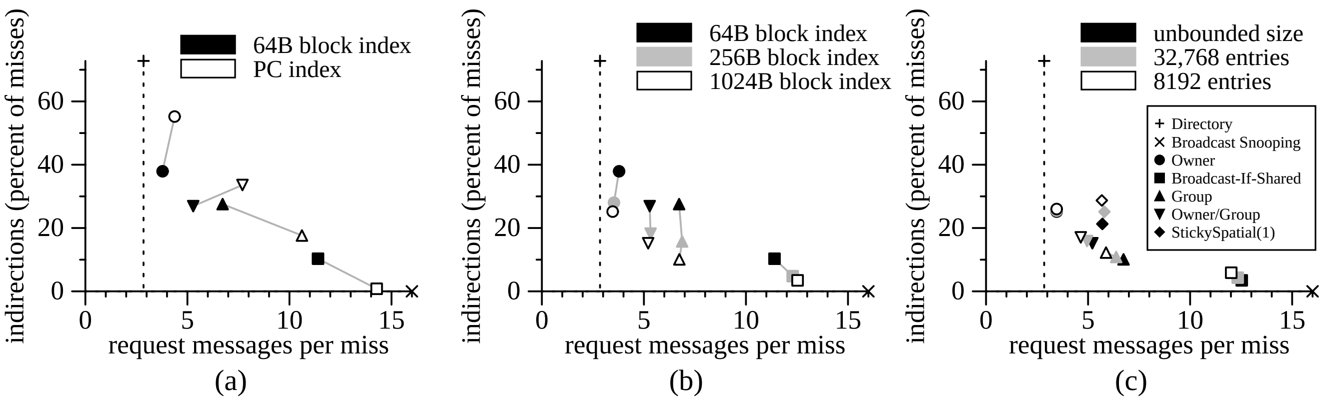


Figure 6. Sensitivity Analysis Using OLTP. (a) The effect of program counter (PC) versus data block indexing. (b) The effect of macroblock indexing. (c) Sensitivity to size with 1024B macroblocks and comparison to *StickySpatial(1)*.

using only incrementally more bandwidth than the directory protocol. In five of our six benchmarks, *Owner* reduces the rate of indirections to less than 25% of all misses. The reduction of indirections comes at the cost of less than a 25% increase in request traffic for five of six benchmarks (less than a 15% increase in total traffic).

***Broadcast-If-Shared* predictor (■).** In contrast to the *Owner* predictor, the goal of *Broadcast-If-Shared* is to achieve performance similar to broadcast snooping systems while using less bandwidth. *Broadcast-If-Shared* meets its goal by keeping indirections to less than 6% of misses for all of our benchmarks while using less bandwidth. In those workloads with a low percentage of cache-to-cache misses (Slashcode and SPECjbb), *Broadcast-If-Shared* reduces the request bandwidth used by more than half. In those workloads with a high percentage of cache-to-cache misses (Apache, Barnes-Hut, and OLTP), the predictor broadcasts most requests and performs like broadcast snooping.

***Group* predictor (▲).** While *Owner* and *Broadcast-If-Shared* are often too conservative or aggressive, respectively, *Group* provides an attractive alternative to these two extreme predictors. For all workloads, *Group* reduces request traffic to no more than half that of snooping, while keeping indirections below 15% of misses. *Group* works particularly well on Slashcode, using one fifth the request bandwidth of snooping with only 4% of requests requiring indirection (a factor of ten improvement).

***Owner/Group* predictor (▼).** *Owner/Group* performs like *Group*, but it uses less bandwidth at the cost of more indirections. Not surprisingly, for most of our benchmarks, the results for this predictor lie between those of *Group* and *Owner*. However, for Ocean, *Owner/Group* incurs only 6% indirections, while using one fifth the request bandwidth of broadcast snooping. As revealed in Figure 3(b), Ocean has a large number of misses to blocks that are only shared among a small number of processors (a consequence of its column-blocked data layout [34]). The "Group" aspect of *Owner/Group* detects this stable, limited sharing, and the "Owner" aspect reduces the bandwidth even further by sending requests for shared only to the predicted owner.

Predictor policy conclusions. There is no "best choice" among these four destination-set predictors for all workloads. The right choice will depend upon the relative importance of latency and the cost of bandwidth in the system being designed. However, for many systems, *Owner* and *Owner/Group* appear to present attractive options.

4.4 Sensitivity Analysis

We now examine the sensitivity of these results to indexing method and predictor size. To limit the number of graphs, we only present data for the OLTP workload, noting significant differences in other workloads. To facilitate comparisons between predictors using the same policy, Figure 6 "connects the dots" for those data points with similar prediction policies but different configurations.

Program counter indexing. Figure 6(a) illustrates (for OTLP with unbounded predictors) the trade-off between using data block or PC-based indexing. These results, and others not shown, indicate that data block indexing yields better predictions in many cases (*e.g.*, for *Owner* and *Owner/Group*). In other cases, the choice between PC and data block indexing creates a bandwidth/latency tradeoff (*e.g.*, for *Group* and *Broadcast-If-Shared*). These results indicate that PC-indexing does not provide sufficient benefit for these simple predictors to justify the additional design cost and complexity required to send miss PCs from the processor core to the coherence controller. PC indexing performs relatively better for finite predictors (not shown), but this effect is dwarfed by macroblock indexing, discussed next.

Macroblock indexing. Macroblock indexing exploits the spatial predictability of coherence requests (as described in Section 3.4). Figure 6(b) shows (for OLTP with unbounded predictors) that using 256-byte or 1024-byte macroblock indexing improves prediction by reducing both traffic and indirections in most cases. Apache and Slashcode exhibit performance similar to OLTP; however, using macroblocks with SPECjbb and Ocean has little effect due to an already low percentage of indirections.

For unbounded predictors, most of the benefit of capturing spatial locality is achieved by 256-byte macroblocks, but 1024-byte macroblocks perform still better while further increasing predictor reach (in the case of a finite size predictor). Experiments with even larger macroblocks and unbounded predictors (not shown) indicate little additional benefit.

Finite sized predictors. Figure 6(c) compares (for OLTP) the performance of unbounded predictors to those with 8,192 and 32,768 entries. The results show that predictors in this range perform comparably to unbounded predictors (for these workloads). Limited experiments with smaller predictors (not shown) indicate an increase in indirections but a corresponding decrease in bandwidth. This result is expected, since on a miss, our predictors default to predicting the minimal destination set (reducing traffic, but also increasing indirections).

Comparison to previous predictors. The original destination-set predictor is *Sticky-Spatial(1)*, which was described in Section 3.5. *Sticky-Spatial(1)* is also shown in Figure 6(c), denoted by ◆, for a range of predictor sizes. For OLTP, our predictors perform better than *Sticky-Spatial*(1) (*e.g.,* our *Owner/Group* predictor uses less bandwidth and has fewer indirections). In general, our predictors either perform similarly or better than *Sticky-Spatial* in one or both criteria.

5 Runtime Performance Evaluation

This section evaluates the impact of destination-set prediction policies on runtime performance. We first present the methodology and then summarize the key results.

5.1 Target System

We evaluate 16-node systems in which each node contains a dynamically scheduled processor core, split first level instruction and data caches, unified second level cache, cache controller, and memory controller for part of the globally shared memory. Table 4 lists the parameters for the memory system and the processor. We choose memory system parameters to approximate the published latencies of systems like the Alpha 21364 [13]. These assumed latencies result in a 180 ns latency to obtain a block from memory, a 112 ns latency for a cache-to-cache transfer for both a broadcast snooping and a successful multicast snooping request, and a 242 ns latency for both a cache-to-cache transfer in the directory protocol and a retried multicast snooping request. All request, forwarded request, and retried request messages are 8 bytes, and data responses are 72 bytes (64 byte data with an 8 byte header).

Destination-set predictors are accessed in parallel with second level caches. Predictor updates complete in a single cycle, and the predictors train only on data responses or on requests from other processors. Since multiple misses are generated in parallel, later misses do not always benefit from the training responses from the earlier misses before being issued into the memory system.

Table 4. Target System Parameters

Coherent Memory System	
L1 instruction cache	128kBytes, 4-way, 2 cycles
L1 data cache	128kBytes, 4-way, 2 cycles
L2 cache (unified)	4MBytes, 4-way, 12ns
block size	64 Bytes
memory	2 GBytes total, 80ns
interconnect link bandwidth	10 GBytes/s
interconnect latency	50ns traversal
Dynamically Scheduled Processor	
clock frequency	2 Ghz
reorder buffer	64 entry
pipeline width	4-wide fetch & issue
pipeline stages	11
direct branch predictor	1kBytes YAGS
indirect branch predictor	64 entry (cascaded)

5.2 Simulation Methods

We simulate our target systems with the Simics full-system multiprocessor simulator [22], and we extend Simics with detailed processor, memory hierarchy, and interconnection network models to compute execution times [3].

Full-system simulation. Simics is a system-level architectural simulator that can run unmodified commercial applications and operating systems. Simics is a functional simulator only, but it provides an interface to support our detailed timing models.

Processor models. We present results using two different processor models. For some results we use TFsim [25] to model superscalar processor cores that are dynamically scheduled, exploit speculative execution, and generate multiple outstanding coherence requests. We configured TFsim to model the processor described in Table 4 and use an aggressive implementation of sequential consistency. For other results, due to excessive simulation runtimes, we use a faster (by an order of magnitude) but simple, in-order, blocking processor model that would complete four billion instructions per second if the L1 caches were perfect. The results using the detailed processor model capture effects due to parallel misses and speculative execution, while the simple processor model allows us to simulate a larger number of cycles for all workloads.

Memory system model. Our memory system simulator captures timing races and all coherence protocol state transitions (including non-stable states). To warm up the simulated caches and predictors before beginning the timing simulations, we use traces similar to those used in Sections 2 and 4. The processor/memory nodes are connected via a single physical link to an interconnection network. Since

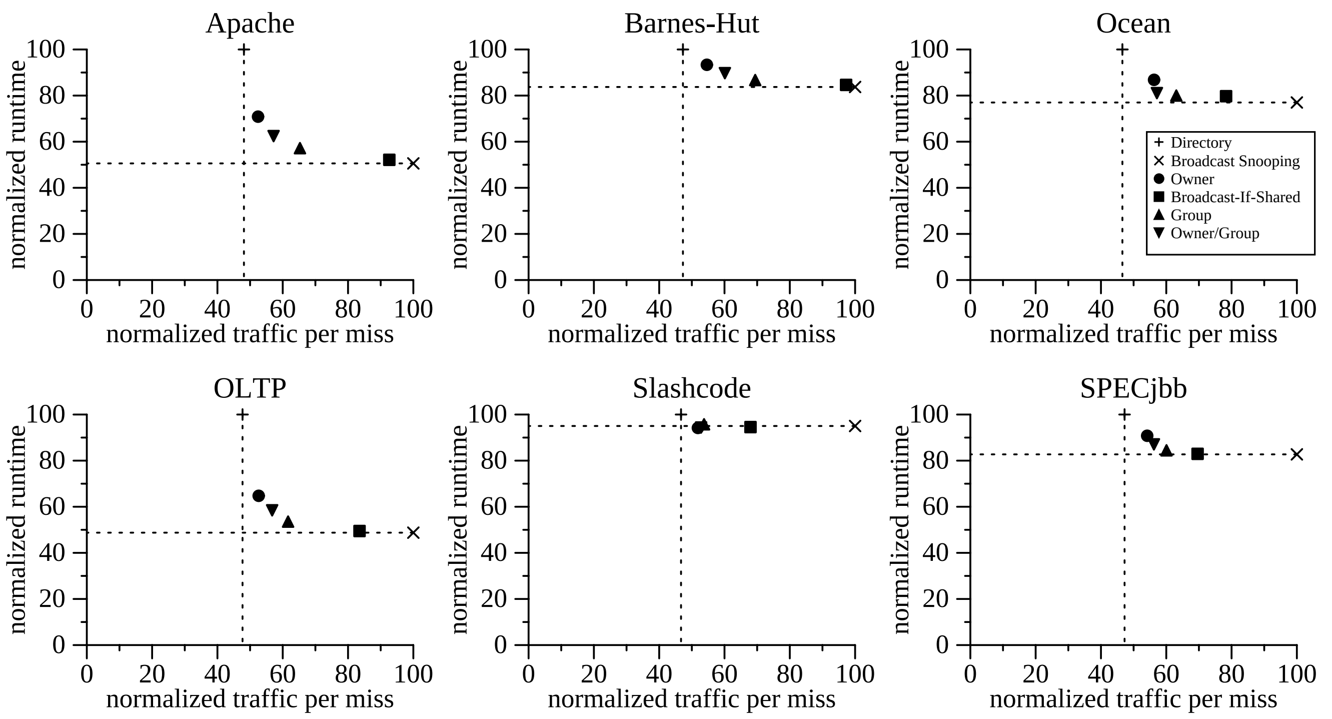


Figure 7. Simple Processor Model Runtime Performance Results

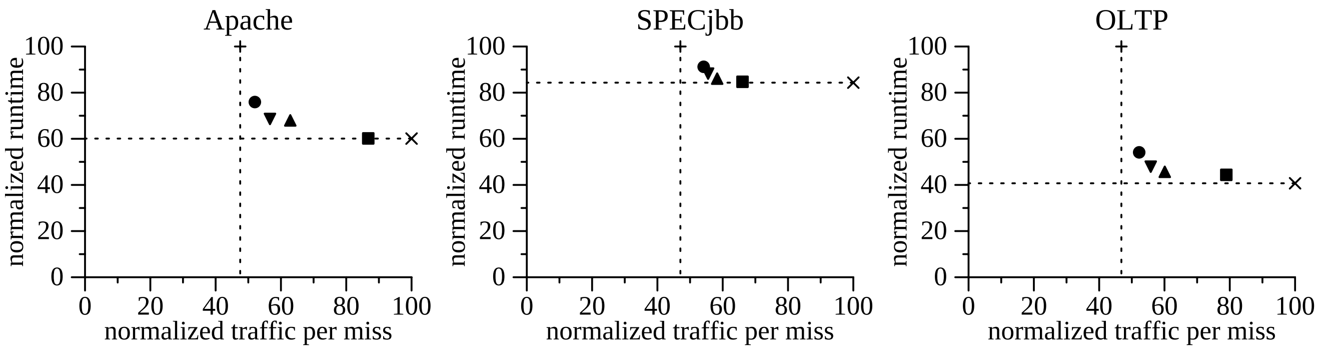


Figure 8. Detailed Processor Model Runtime Performance Results

all of the coherence protocols we consider—broadcast snooping, multicast snooping, and directory—require a total order of requests, we model a single crossbar switch. This interconnect model includes contention effects caused by limited link bandwidth.

Workload variability. To address the runtime variability of commercial workloads, we simulate each design point multiple times with small, pseudo-random perturbations as described by Alameldeen *et al.* [3]. The reported runtime results are averages of these multiple simulations.

5.3 Results

While trace-driven simulation (Section 4) allows rapid exploration of the design space, this section presents the bottom line: execution time and interconnect traffic. However, which protocol performs best depends upon the number of processors and the available interconnect bandwidth. Rather than evaluate these protocols for a particular design point (and arbitrarily pick a winner), we simulate a system with ample bandwidth (10 GB/s links) and examine the tradeoff between execution time and bandwidth. Although snooping always performs best for such a system, we believe these results provide more insight than arbitrarily picking a single bandwidth-constrained design point that would have many critics.

Simple processor model results. Figure 7 shows results generated using the simple processor model that compare the runtime (normalized to the directory protocol) and interconnect traffic (bytes of traffic per miss normalized to broadcast snooping). The dotted lines indicate the ideal cases: the traffic usage of a directory protocol and the runtime of a snooping protocol. For our particular system configuration, snooping uses about twice the interconnect bandwidth of the directory protocol, but it also outperforms the directory protocol by up to a factor of two. Snooping only uses twice the interconnect bandwidth, since point-to-point response messages (72 bytes) are much larger than request messages (8 bytes) that are broadcast to all 16 processors. Not surprisingly, the workloads that benefit most from snooping (OLTP and Apache) have the highest miss rates and cache-to-cache miss rates (recall Table 2). Even those benchmarks with relatively low miss rates improve by approximately 10% to 25%.

Figure 7 also shows that the runtime/bandwidth tradeoff qualitatively mirrors the indirection/request-message tradeoff in Figure 5. The quantitative benefits are somewhat smaller for reasons analogous to why cache miss ratio reductions translate to more modest runtime gains; the predictors help cache-to-cache misses but not private misses or computation. As before, our predictors capture most of the performance benefit of snooping protocols while using significantly less bandwidth. For example, our predictors obtain almost 90% of the performance of snooping while using only approximately 15% more bandwidth than a directory protocol (and less than half the bandwidth of snooping).

Detailed processor model results. Figure 8 displays similar results using our complex processor model for three of our workloads. To enable reasonable simulation runtimes, we only simulated three workloads, and we simulated an order of magnitude fewer transactions for these runs than the earlier trace-based or simple processor model simulations. Normalized runtime and bandwidth numbers are similar to results with the simple processor model, although the absolute runtimes are different.

6 Related Work

Several papers have examined shared memory behavior. Gupta and Weber [12] analyzed invalidation patterns in parallel scientific and engineering applications and observed that different data structures exhibit specific sharing patterns and that most invalidations affect few processors. Recent research has studied commercial workloads (*e.g.*, [5, 18, 28, 29, 30]) but not the distribution of sharers. To our knowledge, this paper is the first paper to perform a detailed analysis of sharing patterns for commercial workloads and their impact on multiple destination-set predictors.

Previous work on destination-set predictors has focused on the correctness of the hybrid protocols and single points in the destination-set predictor design space using scientific workloads (*e.g.*, SPLASH-2 benchmarks [34]). Acacio *et al*. studied a two-level owner predictor, with the first level deciding *whether* to predict an owner and the second level deciding *which node* might be the owner [1]. In a second paper, Acacio *et al*. studied a single-level predictor to predict sharers [2]. Bilir *et al*. [7] studied multicast snooping with a 4K-entry StickySpatial(1) destination-set predictor.

Many papers have examined or exploited other forms of coherence prediction (*e.g.*, dynamic self-invalidation [20, 21]). Coherence predictors have been indexed with addresses [27], program counters [16], message history [19], and other state [17]. Researchers have also developed protocols that optimize for specific sharing behaviors [6], read-modify-write sequences [28, 29], and migratory sharing [8, 33]. Other hybrid protocols adapt between write-invalidate and write-update [4, 9, 15, 26, 31], by migrating data near to where it is being used [10, 14, 35] or by adapting to available bandwidth [24].

7 Conclusions

In this paper, we used commercial workloads to demonstrated the potential of destination-set prediction to improve the latency/bandwidth tradeoff in coherence protocols. While broadcast snooping protocols optimize latency and directory protocols optimize bandwidth, they represent the extreme points in the design space. Even simple destination-set predictors, used in the context of multicast snooping, can (a) greatly reduce the bandwidth usage, with respect to snooping, for a small cost in extra indirections, or (b) greatly reduce the number of indirections, with respect to directory protocols, for a small cost in extra bandwidth. While commercial workloads have larger footprints and more cache-to-cache misses than scientific workloads, we have shown that reasonably-sized predictors can still achieve high accuracy.

Acknowledgments

We thank Virtutech AB, the Wisconsin Condor group, and the Wisconsin Computer Systems Lab for their help and support. We thank Alaa Alameldeen, Brad Beckmann, Carl Mauer, Kevin Moore, the Wisconsin Multifacet group, and the Wisconsin Computer Architecture Affiliates for their comments on this work.

References

[1] M. E. Acacio, J. González, J. M. García, and J. Duato. Owner Prediction for Accelerating Cache-to-Cache Transfers in a cc-NUMA Architecture. In *Proceedings of SC2002*, Nov. 2002.

[2] M. E. Acacio, J. González, J. M. García, and J. Duato. The Use of Prediction for Accelerating Upgrade Misses in cc-NUMA Multiprocessors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 155–164, Sept. 2002.

[3] A. R. Alameldeen, M. M. K. Martin, C. J. Mauer, K. E. Moore, M. Xu, D. J. Sorin, M. D. Hill, and D. A. Wood. Simulating a $2M Commercial Server on a $2K PC. *IEEE Computer*, 36(2):50–57, Feb. 2003.

[4] C. Anderson and A. R. Karlin. Two Adaptive Hybrid Cache Coherency Protocols. In *Proceedings of the Second IEEE Symposium on High-Performance Computer Architecture*, Feb. 1996.

[5] L. A. Barroso, K. Gharachorloo, and E. Bugnion. Memory System Characterization of Commercial Workloads. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 3–14, June 1998.

[6] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Adaptive Software Cache Management for Distributed Shared Memory Architectures. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 125–135, May 1990.

[7] E. E. Bilir, R. M. Dickson, Y. Hu, M. Plakal, D. J. Sorin, M. D. Hill, and D. A. Wood. Multicast Snooping: A New Coherence Method Using a Multicast Address Network. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 294–304, May 1999.

[8] 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.

[9] F. Dahlgren. Boosting the Performance of Hybrid Snooping Cache Protocols. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 60–69, June 1995.

[10] B. Falsafi and D. A. Wood. Reactive NUMA: A Design for Unifying S-COMA and CC-NUMA. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 229–240, June 1997.

[11] K. Gharachorloo, M. Sharma, S. Steely, and S. V. Doren. Architecture and Design of AlphaServer GS320. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 13–24, Nov. 2000.

[12] A. Gupta and W.-D. Weber. Cache Invalidation Patterns in Shared-Memory Multiprocessors. *IEEE Transactions on Computers*, 41(7):794–810, July 1992.

[13] L. Gwennap. Alpha 21364 to Ease Memory Bottleneck. *Microprocessor Report*, Oct. 1998.

[14] E. Hagersten and M. Koster. WildFire: A Scalable Path for SMPs. In *Proceedings of the Fifth IEEE Symposium on High-Performance Computer Architecture*, pages 172–181, Jan. 1999.

[15] A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator. Competitive Snoopy Caching. *Algorithmica*, 3(1):79–119, 1988.

[16] S. Kaxiras and J. R. Goodman. Improving CC-NUMA Performance Using Instruction-Based Prediction. In *Proceedings of the Fifth IEEE Symposium on High-Performance Computer Architecture*, Jan. 1999.

[17] S. Kaxiras and C. Young. Coherence Communication Prediction in Shared-Memory Multiprocessors. In *Proceedings of the Sixth IEEE Symposium on High-Performance Computer Architecture*, Jan. 2000.

[18] S. Kunkel, B. Armstrong, and P. Vitale. System Optimization for OLTP Workloads. *IEEE Micro*, pages 56–64, May/June 1999.

[19] A.-C. 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.

[20] A.-C. Lai and B. Falsafi. Selective, Accurate, and Timely Self-Invalidation Using Last-Touch Prediction. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 139–148, June 2000.

[21] A. R. Lebeck and D. A. Wood. Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 48–59, June 1995.

[22] P. S. Magnusson et al. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.

[23] M. M. K. Martin, M. D. Hill, and D. A. Wood. Token Coherence: Decoupling Performance and Correctness. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, June 2003.

[24] M. M. K. Martin, D. J. Sorin, M. D. Hill, and D. A. Wood. Bandwidth Adaptive Snooping. In *Proceedings of the Eighth IEEE Symposium on High-Performance Computer Architecture*, pages 251–262, Feb. 2002.

[25] C. J. Mauer, M. D. Hill, and D. A. Wood. Full System Timing-First Simulation. In *Proceedings of the 2002 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 108–116, June 2002.

[26] F. Mounes-Toussi and D. J. Lilja. The Potential of Compile-Time Analysis to Adapt the Cache Coherence Enforcement Strategy to the Data Sharing Characteristics. *IEEE Transactions on Parallel and Distributed Systems*, 6(5):470–481, May 1995.

[27] S. S. Mukherjee and M. D. Hill. Using Prediction to Accelerate Coherence Protocols. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 179–190, June 1998.

[28] J. Nilsson and F. Dahlgren. Improving Performance of Load-Store Sequences for Transaction Processing Workloads on Multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, pages 246–255, Sept. 1999.

[29] J. Nilsson and F. Dahlgren. Reducing Ownership Overhead for Load-Store Sequences in Cache-Coherent Multiprocessors. In *Proceedings of the 2000 International Parallel and Distributed Processing Symposium*, May 2000.

[30] P. Ranganathan, K. Gharachorloo, S. Adve, and L. Barroso. Performance of Database Workloads on Shared-Memory Systems with Out-of-Order Processors. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 307–318, Oct. 1998.

[31] A. Raynaud, Z. Zhang, and J. Torrellas. Distance-Adaptive Update Protocols for Scalable Shared-Memory Multiprocesors. In *Proceedings of the Second IEEE Symposium on High-Performance Computer Architecture*, Feb. 1996.

[32] D. J. Sorin, M. Plakal, M. D. Hill, A. E. Condon, M. M. K. Martin, and D. A. Wood. Specifying and Verifying a Broadcast and a Multicast Snooping Cache Coherence Protocol. *IEEE Transactions on Parallel and Distributed Systems*, 13(6):556–578, June 2002.

[33] P. Stenström, M. Brorsson, and L. Sandberg. Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 109–118, May 1993.

[34] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–37, June 1995.

[35] Q. Yang, G. Thangadurai, and L. N. Bhuyan. Design of Adaptive Cache Coherence Protocol for Large Scale Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 3(3):281–293, May 1992.