Computer Sciences Department

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
David A. Wood

Technical Report #1458 November 2002

UNIVERSITY OF WISCONSIN MADISON

Technical Report #1458, Computer Sciences Department, University of Wisconsin-Madison — November 2002

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

Prediction forms the basis of many optimizations in computer architecture. In this paper, we explore using destination-set prediction to improve the latency/bandwidth tradeoff of shared-memory multiprocessors. The destination set is the collection of processors that receive a particular coherence request. Snooping protocols send coherence requests to the maximal destination set (i.e., all processors), reducing latency for sharing misses at the expense of bandwidth. Directory protocols send requests to the minimal destination set, reducing bandwidth at the expense of an indirection through the directory. Recently proposed hybrid protocols trade-off latency and bandwidth by sending requests to a predicted destination set less than all processors.

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 using full-system simulations of a 16-way multiprocessor. We corroborate prior results that indicate a large fraction of cache-to-cache misses, and further show that these sharing misses exhibit a large degree of temporal and spatial locality.

Second, we propose a set of predictors that exploit the observed sharing behavior to target different points in the latency/bandwidth tradeoff. We show that small predictors (e.g., ≤ 64K bytes) can use macroblock indexing (e.g., 1024-byte macroblocks) to capture the spatial locality in these workload's sharing patterns.

Third, we illustrate the effectiveness of destination-set predictors in the context of a multicast snooping protocol. Using a combination of traces and detailed full-system timing simulations of a 16-processor system, we show that the Owner/Group predictor 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

Prediction forms the basis of many optimizations in processors (e.g., branch prediction) and memory systems (e.g., hardware prefetching). In this paper, we explore using *destination-set prediction* to improve the latency/bandwidth tradeoff of shared-memory multiprocessors. The destination set is the collection of processors that receive a particular coherence request. For example, broadcast snooping protocols send coherence requests to all processors in a system. Conversely, directory protocols send coherence requests only to the home node, which forwards requests to the minimum number of processors.

The destination-set size represents a key factor in the latency/bandwidth trade-off in shared-memory multiprocessors. Traditional snooping systems optimize sharing miss[1] latency by broadcasting all coherence requests. Broadcasts avoid the latency of indirections but require (end-point) bandwidth proportional to the square of the number of processors. Directory protocols require less bandwidth, by first sending a request to the directory, that then responds directly with data or redirects the request to a limited number of possible sharers and/or a remote owner. Directory protocols reduce the required bandwidth but increase the latency of sharing misses because of the indirection.

This latency/bandwidth tradeoff is especially important for the many commercial workloads that exhibit a high frequency of sharing misses [3, 5, 18, 30]. As illustrated in Figure 1, system designers must choose between the high bandwidth usage of snooping protocols or the higher sharing latency of directory protocols. An ideal protocol would combine the low latency of broadcast snooping with the bandwidth efficiency of a directory scheme, by having a processor directly send a request to only those processors that need to see it.

Recently proposed hybrid protocols seek to achieve this ideal [1, 2, 7, 32]. For example, *Multicast Snooping* [7] reduces bandwidth compared to broadcast snooping, by

1. Sharing misses are those that move data directly between caches (a.k.a., dirty misses, 3-hop transfers, and cache-to-cache transfers).

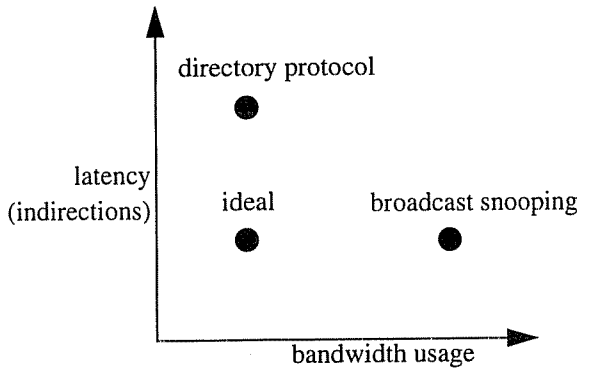

Figure 1. Tradeoff Between Latency and Bandwidth

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 owner prediction to a conventional directory protocol, reducing some sharing misses from three network hops to only two [1, 2].

This paper is the first to examine the efficacy of destination-set predictors for the commercial workloads that dominate the current use of multiprocessor servers. Commercial workloads can benefit substantially from destination-set prediction due to their generally high miss rates and high incidence of sharing misses. To evaluate this potential, in Section 2 we analyze the sharing patterns of four commercial workloads, including an online transaction processing (OLTP) workload, a static web server, a dynamic web server, and a Java middleware workload. We examine the degree of instantaneous sharing, the degree of sharing of data blocks over time, and the temporal and spatial locality of sharing misses.

Section 3 introduces several destination-set predictors that target different points in the latency/bandwidth design space. These predictors exploit different aspects of the sharing behaviors by training and predicting using different information. We also describe some implementation issues, including how to organize predictor resources and when to allocate these resources.

In Section 4, we use request traces to evaluate the destination-set predictors by comparing their latency and bandwidth profiles. We compare these predictors against a broadcast snooping protocol, a directory protocol, and the original multicast snooping destination-set predictor [7]. Results show that our 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.

Section 5 presents results from full-system timing simulations—that model dynamically scheduled out-of-order processors and a detailed memory system—of commercial workloads running on a 16-processor multiprocessor. Results for these timing runs, in terms of relative bandwidth and performance, track the trace results from Section 4. Results show that the our predictors can improve performance by up to 100%, with respect to a directory protocol, while using as little as half the bandwidth of a broadcast snooping system.

Section 6 summarizes related work. Previous work on hybrid protocols and destination-set predictors has focused on single points in the destination-set predictor design space using scientific workloads (e.g., SPLASH-2 benchmarks [35]). This paper makes the following contributions:

- It shows that sharing miss patterns in commercial workloads have substantial temporal and spatial locality that can be captured by destination-set predictors.
- It presents the first comparison of multiple destination-set predictors and shows that they can exploit spatial locality using macroblock indexing (e.g., 1024-byte macroblocks) and achieve good performance with relatively few entries (e.g., 8K entries).
- It presents the first detailed timing-simulation results for a multicast snooping system running commercial workloads, and it shows that destination-set prediction can substantially reduce execution time (compared to directories) while greatly reducing bandwidth (compared to broadcast snooping).

2 Commercial Workload Sharing Behaviors

In this section, we analyze the sharing behaviors of 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 workloads. We begin by presenting our methodology and presenting some general characteristics of our workloads such as miss rates, sharing misses, and data footprint. We then show that the number of instantaneous sharers is small, most of the misses are to a blocks touched by all processors, and there is a large amount of locality among sharing misses.

2.1 Methodology

We use Simics [23] to perform full-system simulation of systems running commercial workloads. The simulated machine is a 16-processor SPARC system running Solaris 8. We collected traces of second-level cache misses

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 (totalling ~500 MB) and 160 simulated users (10 per processor). The system is warmed up for 1.6 million requests, and our 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 [35]: *barnes-hut* with 64k bodies, and *ocean* with a 514 x 514 grid. The benchmarks were compiled with Sun's WorkShop C compiler. 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. The benchmark includes driver threads to generate transactions. Our experiments use 24 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, with many concurrent users performing transactions. Our OLTP workload is based on the TPC-C v3.0 benchmark using IBM's DB2 v7.2 EEE database management system. We use an 800MB database with 4000 warehouses stored on five raw disks and an additional dedicated database log disk. The number of districts per warehouse, items per warehouse, and customers per district were reduced to allow for concurrency provided by a larger number of warehouses. There are 128 simulated users (8 per processor). 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 MySQL 3.23.39 as the database engine. A multi-threaded user emulation program is used to simulate user browsing and posting behavior. The database is a snapshot of the slashcode.org site, and it contains ~3,000 messages. There are 48 simulated users (3 per processor). The system is warmed up for 240 transactions before taking measurements, and our results are based on runs of 100 transactions.

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

2.2 General Properties

Studies of multiprocessor commercial workloads and their properties have found that 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 L2 cache miss rates (columns 5 and 6) and many of the misses in our workloads are caused by the operating system (column 7). Note that although the scientific workloads spend little time in the kernel, their user-level miss rates are so low that a high percentage of total misses occur in the kernel. The rightmost column in the table (column 8) lists the percent of L2 misses that would cause indirections in a directory protocol. As discussed in the next section, these workloads have a large percentage of indirections, providing ample opportunity for destination-set prediction to improve their performance.

2.3 Sharing Misses and Instantaneous Sharing

Previous studies have shown that commercial workloads suffer from a large fraction of sharing misses [3, 5, 30]. Our results, shown in column 8 of Table 2, concur with previous results by finding that 35-96% of misses for our commercial workloads are due to sharing. While Barnes-Hut and Ocean both incur a large fraction of sharing misses, their low miss rates result in low rates of sharing misses *per instruction* when compared with our commercial workloads. Thus, workload analysis based on Barnes-Hut and Ocean alone would be a poor basis for designing commercial servers.

While the majority of misses for our workloads cannot be completed without contacting at least one other processor, the number 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 shared copy of a block. Figure 2 shows the percentage of requests that needs to contact various numbers of processors. For our workloads, many requests require directory indirections, but only about 5% of all requests need to be sent to more then one other processor. This result highlights the inefficiency of broadcast snooping, in which all processors in the system observe all requests.

2.4 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

Table 2. Workload Properties

Workload	Memory touched (64 bytes)	Memory touched (1 Kbyte)	Static instructions that cause L2 misses	Total L2 misses (millions)	L2 cache misses per 1,000 instructions	Supervisor L2 misses (percent of total)	Directory indirections
Apache	46 MB	71 MB	18,745	22 M	5.9	88%	89%
Barnes-Hut	11 MB	13 MB	7,912	3 M	0.4	26%	96%
Ocean	52 MB	61 MB	11,384	5 M	0.5	46%	58%
OLTP	57 MB	125 MB	21,921	18 M	7.0	42%	73%
Slashcode	181 MB	316 MB	42,770	13 M	1.0	50%	35%
SPECjbb	341 MB	558 MB	24,023	21 M	3.3	21%	41%

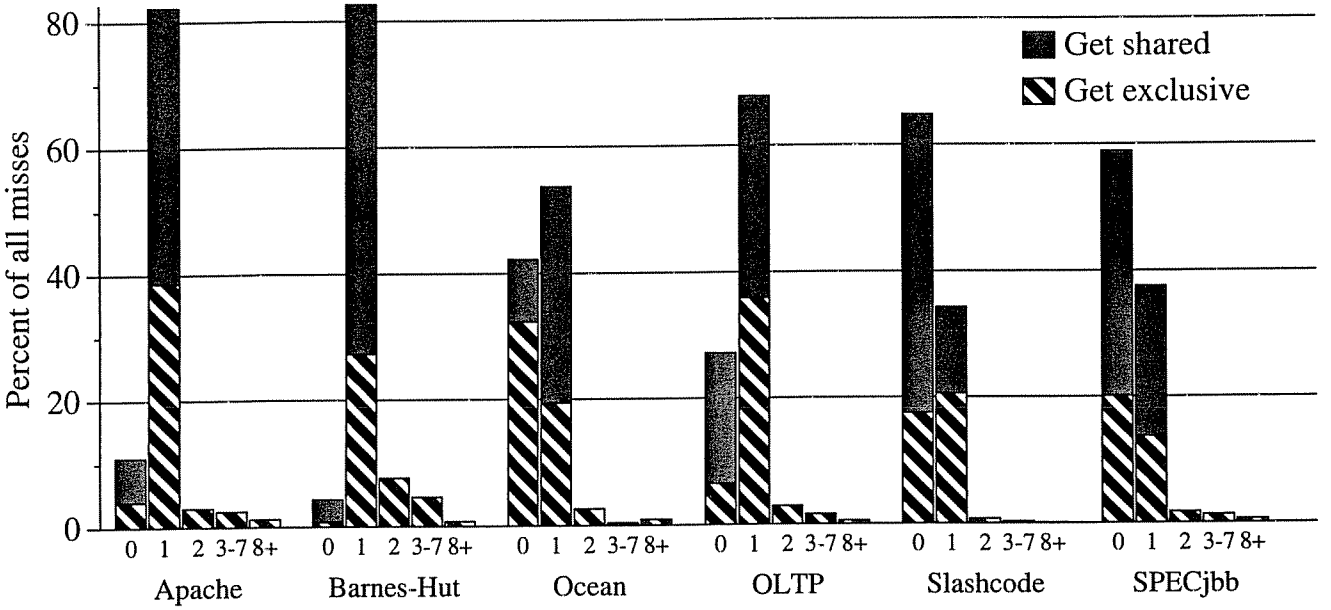

Figure 2. Histogram of the number of processors that need to see an indirection in a directory protocol for requests for shared and exclusive.

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 was touched by four processors, we add ten to the four-processor bin of the histogram). The scaled data show that the majority 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 [35]. When we generated similar data (not shown) using 1024-byte *macroblocks* (*i.e*, 16 blocks), we observed that the macroblock results are similar for our commercial workloads, but the weight shifts to the right for Barnes-Hut and Ocean, indicating that blocks within some macroblocks are accessed by different sets of processors.

2.5 Sharing Locality

Our workloads exhibit a high degree of locality among sharing misses. Figure 4(a) shows the cumulative distribution of sharing 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 sharing 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 sharing misses. Figure 4(c) shows the cumulative distribution of unique instructions that cause sharing misses. These figures reveal significant amounts of temporal and spacial locality in the sharing miss stream, a result that corroborates prior work [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.

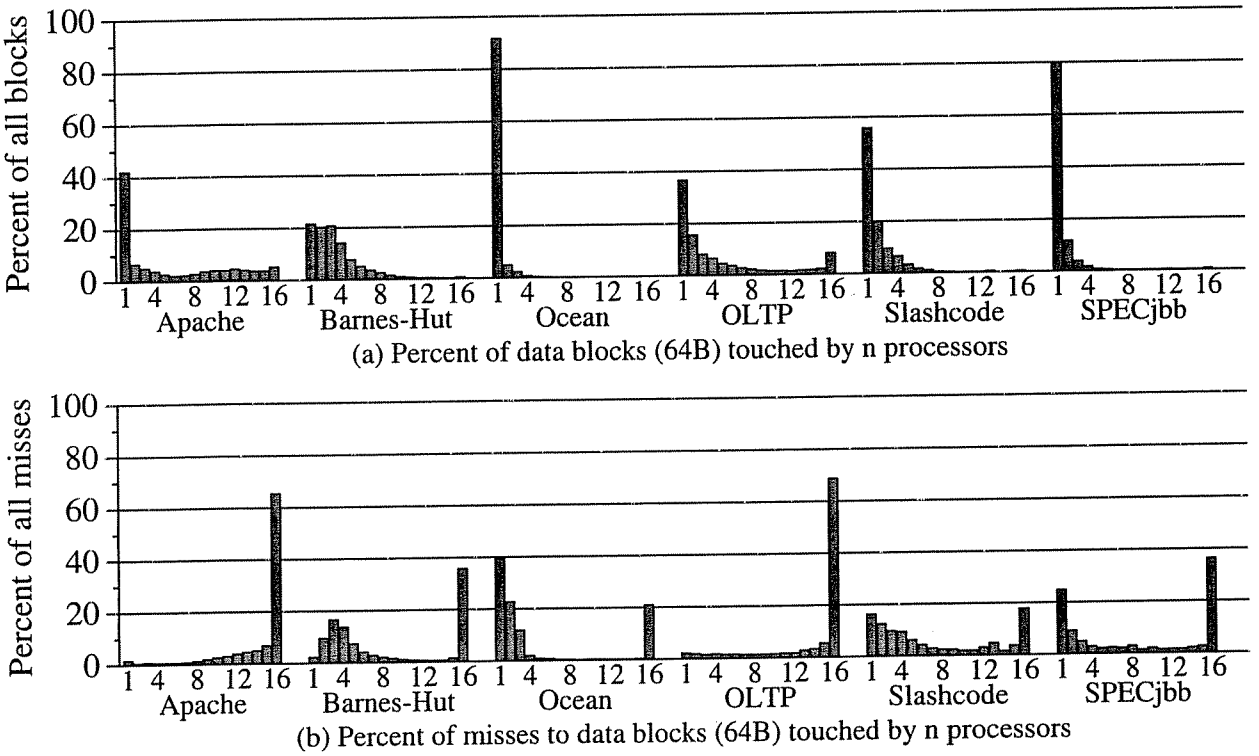

Figure 3. Number of blocks touched by various numbers of processors during execution. Part (a) shows a histogram with one entry for each unique block (64B). In part (b), the data is weighted by the number of misses to the block.

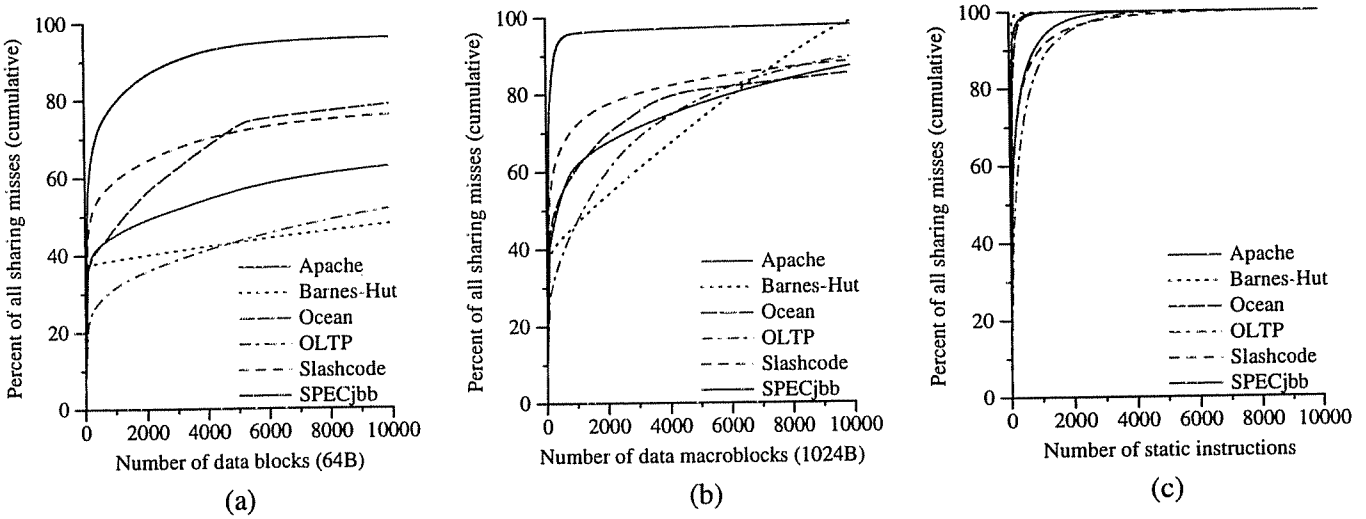

Figure 4. Sharing Locality

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 for shared* must find the current owner, while a *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 transfer latencies of snooping systems.

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

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 (~4 bytes)	2 bits + tag (~4 bytes)	2N bits + 5 bits + tag (~8 bytes)
Prediction Action (for Shared or Exclusive)		If *Valid*, predict *Owner* Otherwise, *Default*	If *Counter* > 1, broadcast Otherwise, *Default*	For each processor n, if *Counters[n]* > 1, add n to *Default*
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 requestor and set *Valid*	Increment *Counter*	Increment *Counters[requestor]*. Increment *RolloverCounter*†
	Other Request (Shared)	ignore		

† If *RolloverCounter* rolls over, decrement *Counter[i]* for all *i*.

directories are effectively the two extremes: snooping always predicts broadcast (perfect accuracy, but high bandwidth usage), while directories always predict the minimal destination set (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

We assume each L2 (or L3) 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 we will explore an *optional* enhancement of exporting the program counter of an instruction that misses in Section 3.4. Predictors are tagged, set-associative, and (by default) indexed by data block address. The controller accesses the predictor before initiating a coherence transaction. On a tag match, the controller uses the prediction according to the policies discussed below. On a miss, the predictor returns a *default destination set* that depends upon the specific hybrid protocol.

Since a small subset of data blocks account for most sharing misses (recall Figure 4), the controller can reduce pressure on the predictor by only allocating predictor entries for blocks likely to be shared. The controller allocates an entry only if the default 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 policies can use some or all of the training information to optimize for different points in the bandwidth/latency spectrum. This section describes three general policies, specified in Table 3, and one hybrid.

The *Owner* predictor. The *Owner* predictor 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 adds this processor to the default 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. The *Broadcast-If-Shared* predictor 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 default 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 consumes less bandwidth by not broadcasting all requests.

The *Group* predictor. The *Group* predictor 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. The predictor 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 inactive processors are eventually removed from the destination set. The *Group* predictor should work well on a large multiprocessor in which not all of the processors are working on the same aspect of the computation or if the system is separated into multiple logical partitions.

The *Owner/Group* predictor. The *Owner/Group* policy targets (a) stable sharing patterns and (b) more limited bandwidth that *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 are indexed with data block addresses. In this paper, we also explore using the program counter (PC) and "coarse-grain" macro-block addresses.

Program counter indexing. Figure 4(c) showed that a small number of static instructions cause most sharing misses. This observation, supported by prior work [16], suggests that we index the predictor with the PC. To do this, the processor supplies the PC of the load/store instruction causing the miss. The cache controller appends this PC to the coherence request (extending the message format) and remembers it until the coherence response arrives.

Macroblock indexing. Figure 4(b) showed that sharing 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 locality, we index the predictor with macroblock addresses by simply dropping the least significant bits. Macroblock indexing reduces the number of unique entries thereby reducing pressure on the predictor.

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(1)* 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 evaluates the predictors using the trace-driven methodology described earlier. Section 4.1 summarizes the multicast snooping protocol we evaluate, Section 4.2 describes how we analyze the latency/bandwidth tradeoff, Section 4.3 presents results for our prediction policies, and Section 4.4 presents some sensitivity analyses.

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 a such a system act much like they do in broadcast snooping, except that coherence requests are multicast 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 interconnection network and the default 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 memory module, the owner of the block, and, if the request is for read/write permission, all processors that share the block.

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

If a destination set is insufficient, the request must be retried. Our implementation uses the optimization proposed by Sorin, *et al.* [32], where the directory re-issues the coherence request[2] with an improved destination set that reflects the current owner and sharers. As in the original protocol,

2. To avoid deadlock, if a retry cannot use the request network due to limited buffering, the directory uses the response network to send a negative acknowledgment (nack) for the request [32].

however, a window of vulnerability exists between the retry's issue and when the request 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 livelock in pathologic cases, the directory resorts to broadcasting on the third retry, which is guaranteed to succeed. When a retry succeeds, the system behaves as described for a successful request.

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, retries, and forwarded requests). 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 our predictors.

We compare our predictors against the 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 network. We model an aggressive, bandwidth-efficient MOSI directory protocol (denoted by ✚) based on the AlphaServer GS320 [11]. The GS320 uses a totally ordered interconnection network, eliminating the need for explicit acknowledgment messages. Using such a directory protocol allows us to assume the same interconnection network configuration for all protocols.

4.3 Predictor Policy Evaluation

Figure 5 displays the results for the four predictor policies, assuming 8,192 entries indexed using 1024-byte macroblock addressing (note the different y-axis for each workload). 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 using only incrementally more bandwidth than the directory protocol. Since the *Owner* predictor only includes one additional destination in the predicted set, it prevents the system from using too much bandwidth. In five of our six benchmarks, the *Owner* predictor 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 sharing misses (Slashcode and SPECjbb), the *Broadcast-If-Shared* predictor reduces the request bandwidth used by more than half. In those workloads with a high percentage of sharing misses (Apache, Barnes-Hut, and OLTP), the predictor broadcasts most requests. Thus, for those workloads, *Broadcast-If-Shared* has similar traffic and indirection characteristics as broadcast snooping.

***Group* predictor (▲).** While the *Owner* and *Broadcast-If-Shared* predictors are often too conservative or too aggressive, respectively, the *Group* predictor provides an attractive alternative to these two extreme predictors. For all workloads, the *Group* predictor reduces request traffic to no more than half that of snooping, while keeping indirections below 15% of misses. This predictor configuration works particularly well on Slashcode, using one fifth the request bandwidth of snooping with less than 5% of requests requiring indirection (a factor of ten improvement).

***Owner/Group* predictor (▼).** The *Owner/Group* predictor for this configuration performs much 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* (*i.e.*, it is neither clearly worse nor better than either predictor). However, for Ocean, the *Owner/Group* predictor incurs only 6% indirections, while using one fifth the request bandwidth of broadcast snooping. The explanation for this commendable performance comes directly from data shown in Figure 3(b). Ocean has a large number of misses to macroblocks that are only shared among a small number of processors (a consequence of its column-blocked data layout [35]). The "Group" aspect of the *Owner/Group* predictor detects this stable, limited sharing, and the "Owner" aspect reduces the bandwidth even further by sending requests for shared to the current owner.

Predictor policy conclusions. For most workloads, there is no "best choice" among these four destination-set predictors. For a system designer, 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 alternatives.

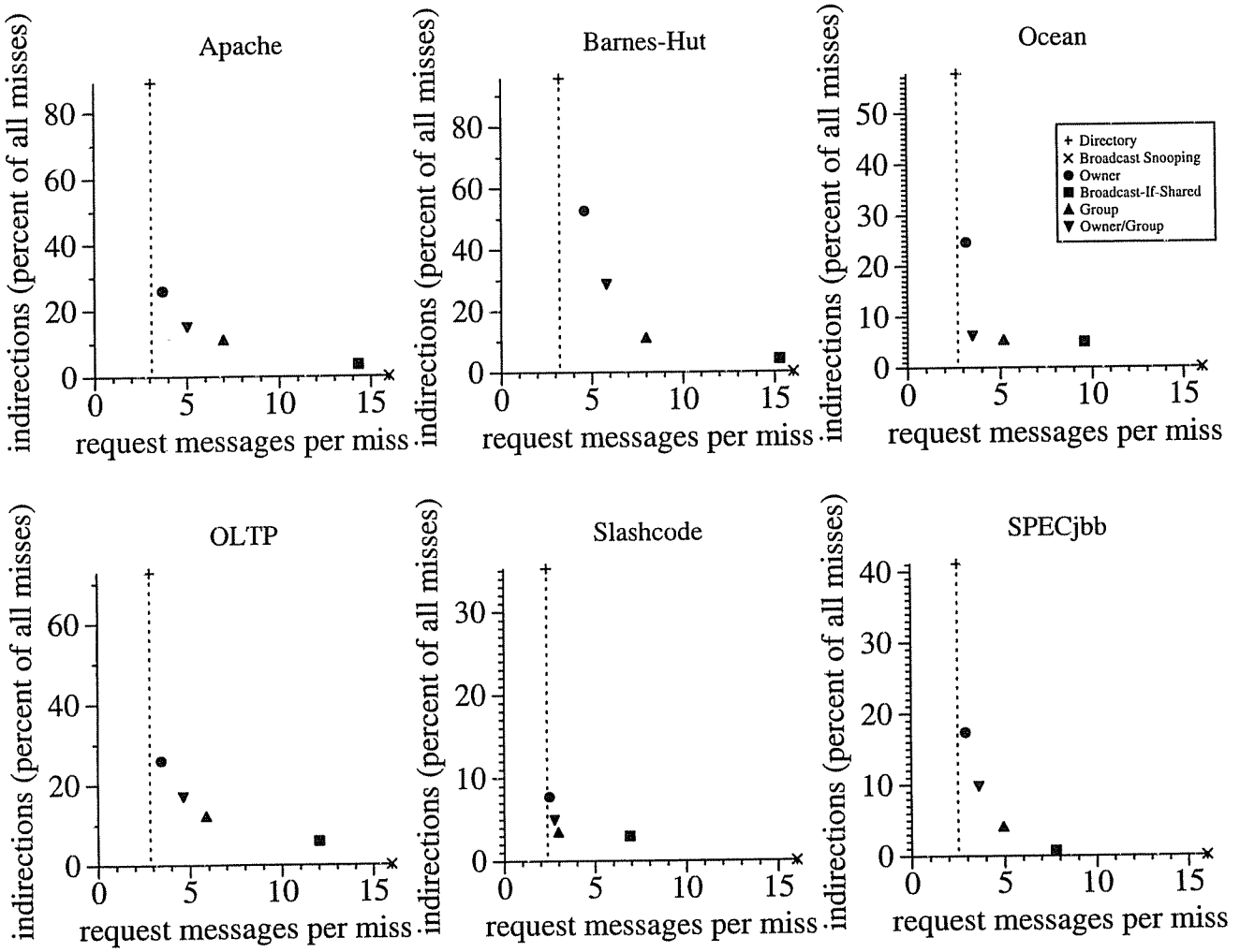

Figure 5. Predictor Results: Standout Predictors (8,192 entries, 1,024-byte block indexing)

4.4 Sensitivity Analysis

We now examine the sensitivity of these results to indexing 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, Figures 6–8 "connect the dots" for those data points with similar prediction policies but different configurations.

Program Counter Indexing. Figure 6 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 the *Owner* and *Owner/Group* policies). In other cases, the choice between PC and data block indexing creates a bandwidth/latency tradeoff (*e.g.*, for the *Group* and *Broadcast-If-Shared* policies). These results indicate that PC-indexing does not provide sufficiently better performance to justify the additional design cost and complexity required to send miss PCs from the processor core to the cache controller. PC-based indexing performs slightly better for finite size predictors, but these effects are dwarfed by macroblock indexing, discussed next.

Macroblock Indexing. Figure 7 shows (for OLTP with unbounded predictors) the result of using 256-byte and 1024-byte macroblock indexing. Macroblock indexing improves prediction by reducing both traffic and indirections in most cases. In addition, macroblock indexing simultaneously reduces the number of entries required in the predictor. For unbounded predictors, most of the benefit of capturing spatial locality is achieved by 256-byte mac-

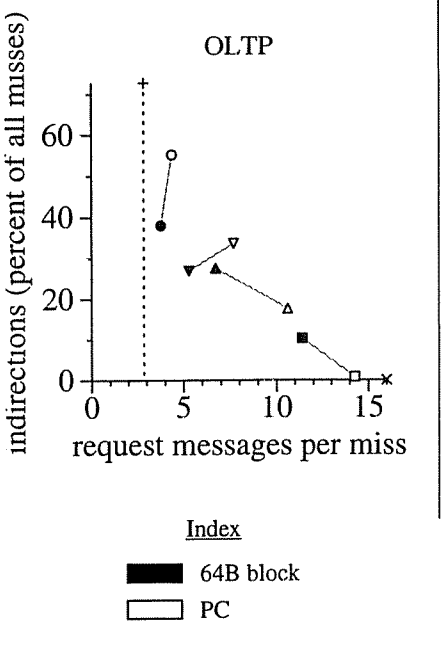

Figure 6. Predictor Results: Program Counter vs. Data Block Indexing for OLTP

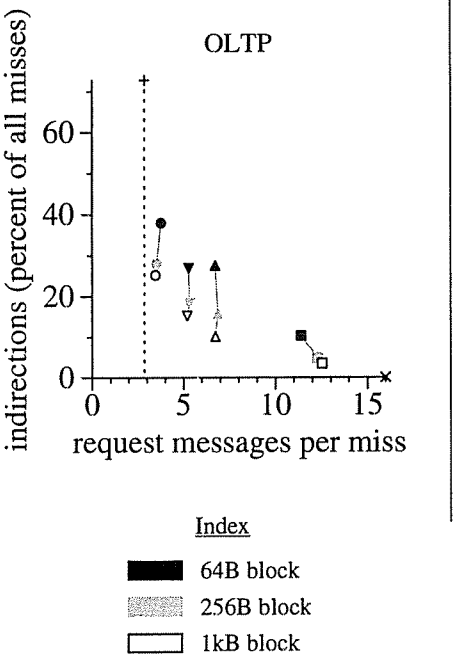

Figure 7. Predictor Results: Macroblock Indexing for OLTP

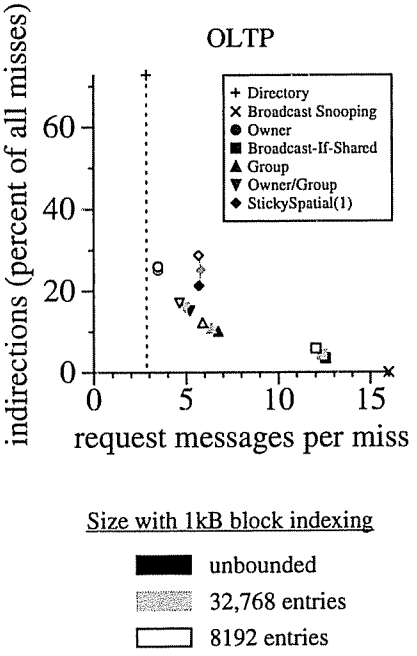

Figure 8. Predictor Results: Sensitivity to size and comparison to *StickySpatial(1)* for OLTP

roblocks, but 1024-byte macroblocks perform still better while further reducing predictor pressure. Experiments with even larger macroblocks and unbounded predictors (not shown) indicate little additional benefit. 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.

Finite Sized Predictors. Figure 8 compares (for OLTP) the performance of unbounded predictors to those with 8,192 and 32,768 entries. Since our predictor entry sizes range from ~4 to ~8 bytes, the total predictor sizes range from 32kB to 64kB (less than 2% of our L2 cache size). The results show that predictors in this range perform comparably to unbounded predictors, for these workloads. Limited experiments with smaller predictors (not shown) show an increase in indirections but a corresponding decrease in bandwidth. We expect this result, since on a miss, our predictors resort to predicting the default destination set (reducing traffic, but also increasing indirections).

Comparison to Previous Predictors. The original destination-set prediction is *Sticky-Spatial(1)*, which was described in Section 3.5. *Sticky-Spatial(1)* is also shown in Figure 8 (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 the runtime performance of multicast snooping running commercial workloads. We first present the evaluation methodology and then summarize the key results.

5.1 Target System

We evaluate 16-node SPARC V9 systems running unmodified Solaris 8. 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 system parameters for both 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 a cache-to-cache transfer for a directory 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 block with an 8 byte header).

Destination-set predictor updates complete in a single cycle. However, the predictors train only on data responses

Table 4. Target System Parameters

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

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.

5.2 Simulation Methods

We simulate our target systems with the Simics full-system multiprocessor simulator [23], and we extend Simics with detailed processor, memory hierarchy, and network models to compute execution times.

Full-system simulation. Simics is a system-level architectural simulator developed by Virtutech AB 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 simulation.

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 to support SPARC's total store order (TSO) memory consistency model. For other results, we use a faster (by an order of magnitude) but simple, in-order, sequentially consistent, 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 cycles to avoid possible bias introduced by executing only short segments of a workload.

Memory hierarchy model. Our memory hierarchy simulator captures timing races and all state transitions (including transient states) of the coherence protocols in cache and memory controllers. We use traces, similar to those used in Sections 2 and 4, to warm up the simulated caches and predictors before beginning the timing simulations.

Interconnection network model. We consider integrated processor/memory nodes connected via a single link to an interconnection network. Since all of the coherence protocols we consider—broadcast snooping, multicast snooping, and directory—require a total order of requests, we model a crossbar switch including limited bandwidth and contention.

Variability in Workload Performance. To address the variability in runtime performance of commercial workloads, we use the methodology described by Alameldeen, et al. [3]. We simulate each design point multiple times with small, pseudo-random perturbations of memory latencies to cause alternative scheduling paths. The results reported in this section 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. Clearly, a directory protocol will excel in a large or bandwidth-starved system, while a snooping system will dominate at the other extreme. 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.

Figure 9 shows results generated using the simple processor model that compare the runtime (normalized to the directory protocol) and bandwidth (traffic per miss normalized to broadcast snooping). The dotted lines indicate the bandwidth and runtime requirements of the directory and snooping protocols, respectively. For our particular system configuration, the snooping protocol uses about twice the bandwidth of the directory protocol, but it also outperforms the directory protocol by up to a factor of two. The snooping protocol only uses twice the bandwidth, since the point-to-point response messages (64 bytes) are much larger than the request messages (8 bytes) that are broadcast to all 16 processors. Not surprisingly, the workloads that show the most benefit from snooping (OLTP and Apache) have the

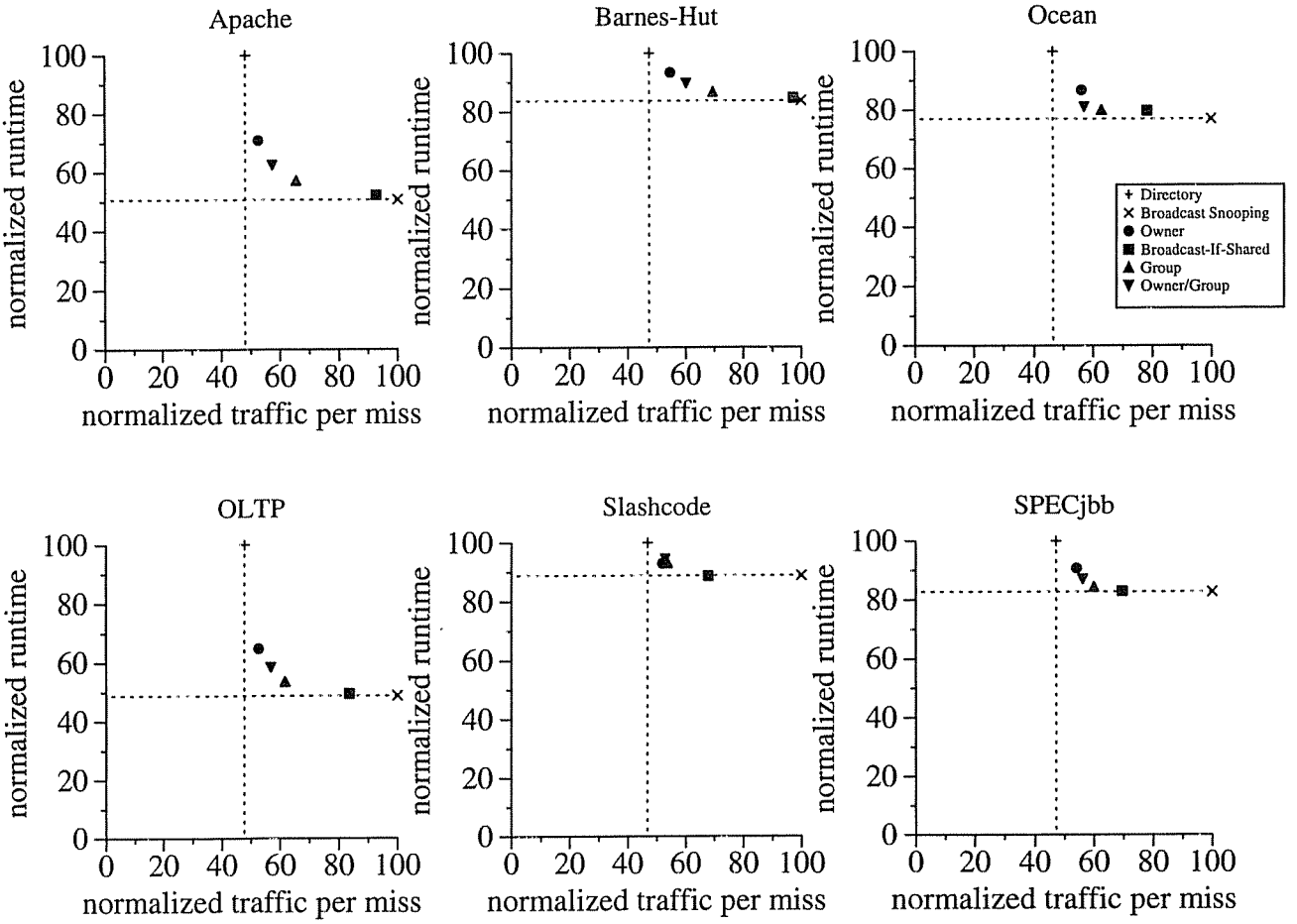

Figure 9. Simple Processor Model Runtime Performance Results

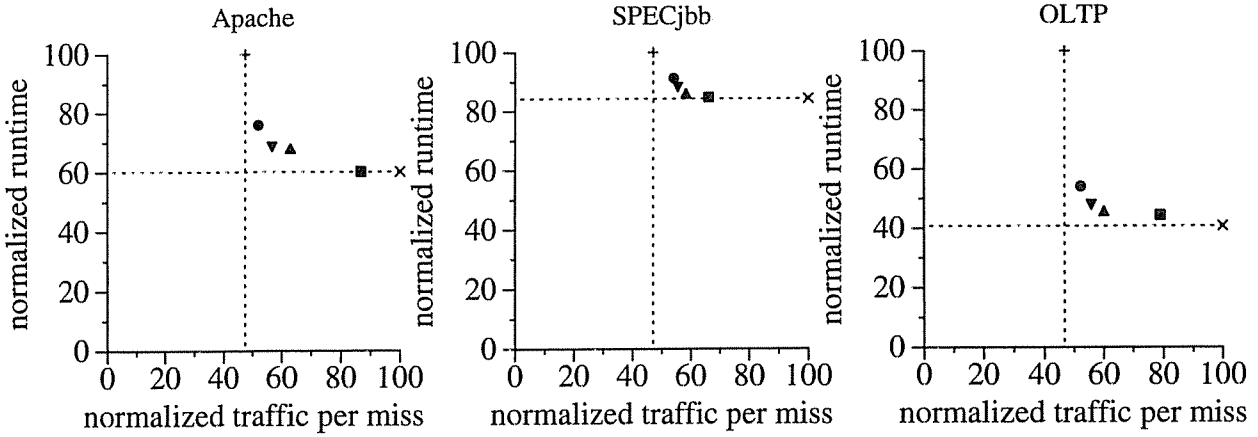

Figure 10. Detailed Processor Model Runtime Performance Results

highest miss rates and sharing miss rates (recall Table 2). Even those benchmarks with relatively low miss rates improve by approximately 10% to 25%.

Figure 9 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 sharing misses, but not private misses or computation. As before, our predictors capture most of the performance benefit of snooping 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).

Figure 10 displays similar results using our complex processor model and the three most important commercial workloads. To enable reasonable simulation runtimes, 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 analyzed invalidation patterns in parallel scientific and engineering applications and observed that different data structures exhibit specific sharing patterns and that most invalidations affect a small number of processors [12]. Recent research has studied commercial workloads [5, 28, 29, 30, 34] but not the distribution of sharers. To our knowledge, this 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 [35]). 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. Researchers have developed protocols that adapt to specific sharing behaviors [6], including read-modify-write sequences [21, 28], migratory sharing [8, 33], and dynamic self-invalidation [20, 22]. More recently, research has focused on separating the predictor from the coherence protocol [27]. Coherence predictors have been indexed with addresses [27], program counters [16], message history [19], and other state [17]. Other hybrid protocols adapt between write-invalidate and write-update [4, 26, 31], write-through and write-invalidate subprotocols [9, 15], or by migrating data near to where it is being used [10, 14, 36]. Another protocol adapts to available bandwidth but not sharing patterns [24].

7 Conclusions

In this paper, we have demonstrated the potential to use destination-set prediction to improve the latency/bandwidth tradeoff in coherence protocols for commercial workloads. While broadcast snooping optimizes latency and directory protocols optimizes 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 sharing 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, Carl Mauer, Kevin Moore, and the Wisconsin Computer Architecture Affiliates for their comments on this work.

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), Spanish Secretaría de Estado de Educación y Universidades (Hill sabbatical), two Wisconsin Romnes Fellowships (Hill and Wood), and donations from Compaq Computer Corporation, Intel Corporation, IBM, and Sun Microsystems.

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, C. J. Mauer, M. Xu, P. J. Harper, M. M. Martin, D. J. Sorin, M. D. Hill, and D. A. Wood. Evaluating Non-deterministic Multi-threaded Commercial Workloads. In *Proceedings of the Fifth Workshop on Computer Architecture Evaluation Using Commercial Workloads*, Feb. 2002.

[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*, 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*, 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] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.

[22] 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*, June 1995.

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

[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), 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*, 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. 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] P. Trancoso, J.-L. L. Pey, Z. Zhang, and J. Torrellas. The Memory Performance of DSS Commercial Workloads in Shared-Memory Multiprocessors. In *Proceedings of the Third IEEE Symposium on High-Performance Computer Architecture*, pages 250–260, Feb. 1997.

[35] 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*, June 1995.

[36] 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.