Flexible Snooping: Adaptive Forwarding and Filtering of Snoops in Embedded-Ring Multiprocessors*

Karin Strauss **Xiaowei Shen**† **Josep Torrellas**

Dept. of Computer Science
University of Illinois, Urbana-Champaign
{kstrauss, torrellas}@cs.uiuc.edu
http://iacoma.cs.uiuc.edu

†IBM T. J. Watson Research Center
Yorktown Heights, NY
xwshen@us.ibm.com

Abstract

A simple and low-cost approach to supporting snoopy cache coherence is to logically embed a unidirectional ring in the network of a multiprocessor, and use it to transfer snoop messages. Other messages can use any link in the network. While this scheme works for any network topology, a naive implementation may result in long response times or in many snoop messages and snoop operations.

To address this problem, this paper proposes *Flexible Snooping* algorithms, a family of adaptive forwarding and filtering snooping algorithms. In these algorithms, a node receiving a snoop request may either forward it to another node and then perform the snoop, or snoop and then forward it, or simply forward it without snooping. The resulting design space offers trade-offs in number of snoop operations and messages, response time, and energy consumption. Our analysis using SPLASH-2, SPECjbb, and SPECweb workloads finds several snooping algorithms that are more cost-effective than current ones. Specifically, our choice for a high-performance snooping algorithm is faster than the currently fastest algorithm while consuming 9-17% less energy; our choice for an energy-efficient algorithm is only 3-6% slower than the previous one while consuming 36-42% less energy.

1. Introduction

The wide availability of Chip Multiprocessors (CMPs) is enabling the design of inexpensive, multi-CMP shared-memory machines of medium size (32-128 cores). However, as in traditional, less-integrated designs, supporting hardware cache coherence in these machines requires a major engineering effort.

There are several known approaches to build cache coherence support in medium-sized shared-memory machines [5]. One of them is a snoopy protocol with one or several buses to broadcast coherence operations. Another is a directory-based protocol, which uses a distributed directory to record the location and state of cached lines. Another scheme is Token Coherence [10], which extends a protocol with tokens to make it easier to serialize concurrent transactions in any network topology. Finally, another approach is to use a snoopy protocol with a unidirectional ring [2]. In this case, coherence transactions are serialized by sending snoop messages along the ring.

This last approach is particularly attractive if we *logically embed* the ring in whatever network topology the machine uses. Snoop messages use the logical ring, while other messages can use any link in the network. The resulting design is simple and low cost. Specifically, it places no constraints on the network topology or timing. In addition, it needs no expensive hardware support such as a broadcast bus or a directory module. Moreover, the ring's serialization properties enable the use of a simple cache coherence protocol. Finally, while it is not highly scalable, it is certainly appropriate for medium-range machines — for example, systems with 8-16 nodes.

Perhaps the main drawback of this approach is that snoop requests may suffer long latencies or induce many snoop messages and operations. For example, a scheme where each CMP snoops the request before forwarding it to the next CMP in the ring induces long request latencies. Alternatively, a scheme where each CMP immediately forwards the request and then performs the snoop will be shown to induce many snoop messages and snoop operations. This is energy inefficient. Unfortunately, as technology advances, these shortcomings become more acute: long latencies are less tolerable to multi-GHz processors, and marginally-useful energy-consuming operations are unappealing in energy-conscious systems.

Ideally, we would like to forward the snoop request as quickly as possible to the CMP that will provide the line while consuming as little energy as possible. To this end, this paper proposes *Flexible Snooping* algorithms, a family of adaptive forwarding and filtering snooping algorithms. In these algorithms, depending on certain conditions, a CMP node receiving a snoop request may either forward it to another CMP and then perform the snoop, or snoop and then forward it, or simply forward it without snooping.

We examine the design space of these algorithms and, based on the analysis, describe four general approaches for these algorithms. They represent different trade-offs in number of snoop operations and messages, snoop response time, energy consumption, and implementation difficulty.

Our analysis using SPLASH-2, SPECjbb, and SPECweb workloads finds that several of these snooping algorithms are more cost-effective than current ones. Specifically, our choice for a high-performance snooping algorithm is faster than the currently

*This work was supported in part by the National Science Foundation under grants EIA-0072102, EIA-0103610, CHE-0121357, and CCR-0325603; DARPA under grant NBCH30390004; DOE under grant B347886; and gifts from IBM and Intel. Karin Strauss was supported by an Intel PhD Fellowship.

fastest algorithm while consuming 9-17% less energy; moreover, our choice for an energy-efficient algorithm is only 3-6% slower than the previous one while consuming 36-42% less energy.

The contribution of our paper is three-fold. First, we introduce a family of adaptive snooping algorithms for embedded-ring multiprocessors and describe the primitive operations they rely on. Second, we analyze the design space of these algorithms. Finally, we evaluate them and show that some of them are more cost-effective than current snooping algorithms.

This paper is organized as follows: Section 2 provides background information; Section 3 describes the hardware primitives for Flexible Snooping algorithms; Section 4 presents the design space and implementation of the algorithms; Sections 5 and 6 evaluate the algorithms; and Section 7 discusses related work.

2. Message Ordering & Ring-Based Protocols

2.1. Arbitration of Coherence Messages

A key requirement for correct operation of a cache coherence protocol is that concurrent coherence transactions to the same address must be completely serialized. Moreover, all processors must see the transactions execute in the same order. To see why, consider Figure 1. Processors *A*, *B*, and *C* cache line *L* in state shared (*S*). Suppose that *A* writes *L* and that, after the invalidation from *A* reaches *B*, *B* reads *L*. In this case, if the read from *B* obtains the line from processor *C* before the invalidation from *A* reaches *C*, the system becomes incoherent: *A* will cache the data in state dirty (*D*) and *B* will cache it in state shared (*S*). This problem occurs because the system has failed to serialize the two transactions.

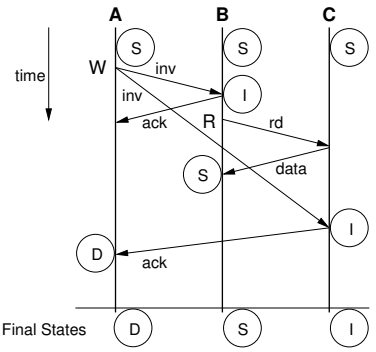


Figure 1. Incorrect execution due to failure to serialize two transactions to the same address.

Different types of coherence protocols ensure transaction serialization differently. Next, we examine snoopy protocols with broadcast link, directory protocols, token coherence, and snoopy protocols with embedded unidirectional ring.

2.1.1. Snoopy with Broadcast Link

In this approach, transactions obtain a shared broadcast link, which all processors snoop [5]. Only one transaction can use the link each cycle. If two transactions to the same line *L* attempt to grab the bus, only one succeeds, and the other is forced to wait, serializing the transactions. Modern designs use advanced implementations, such as split-transaction buses and multiple buses. For example, Sun's Starfire [21] uses four fast buses for snoop messages; data transfers are performed on another network.

A drawback of buses is their limited scalability. Buses can only support one transaction per cycle, require global arbitration cycles to be allocated, and have physical effects that limit their frequency, such as signal propagation delays, signal reflection and attenuation. While manufacturers have engineered ever better designs, it is hard to imagine buses as the best interconnect for a high-frequency multi-CMP machine with 64-128 processors.

2.1.2. Directory

In directory protocols, all transactions on a memory line *L* are directed to the directory at the home node of that line [5]. The directory serializes the transactions — for example, by bouncing or buffering the transaction that arrives second until the one that arrived first completes. While directory protocols such as that in Silicon Graphics's Altrix [19] are scalable, they add non-negligible overhead to a mid-range machine — directories introduce a time-consuming indirection in all transactions. Moreover, the directory itself is a complicated component, with significant state and logic.

2.1.3. Token Coherence

In Token Coherence, each memory line, whether in cache or in memory, is associated with N tokens, where N is the number of processors [10]. A processor cannot read a line until its cache obtains at least one of the line's tokens. A processor cannot write to a line unless its cache obtains all of its tokens. This convention ensures that two transactions to a line are serialized, irrespective of the network used. Partial overlap results in failure of one or both transactions to obtain all necessary tokens. These transactions then retry.

While conceptually appealing, the scheme has some potentially difficult implementation issues. One of them is that retries may result in continuous collisions, potentially creating live-lock. A solution based on providing some queuing hardware to ensure that colliding transactions make progress is presented in [11]. Another issue is that every line needs token storage in main memory, since some of the line's tokens may be stored there. Unless special actions are taken, such token memory may need to be accessed at write transactions. Finally, in multiprocessors with multiple CMPs, the scheme needs to be extended with additional storage and state to allow a local cache in the CMP to supply data to another local cache. Some of these issues are addressed in [11].

2.1.4. Snoopy with Embedded Unidirectional Ring

In this approach, coherence transactions are serialized by sending control messages along a unidirectional ring connecting all processors. Collisions of two transactions directed to the same line *L* are easily detectable because all snoop requests go around the ring and can be seen by all the processors. A collision can be detected by a processor that issued one of the two requests, or by the processor supplying a response. At that point, the processor marks one of the messages as squashed. Squashed messages are retried later. With a given algorithm of message priorities, collisions are resolved with the squash of only one message; there is no need to retry both.

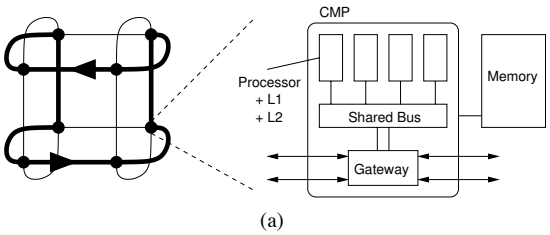
Older designs use a ring network *topology* [2, 24] and constrain the timing of transactions with time slots. However, in practice, this approach works with any network topology — as long as we

embed a unidirectional ring in the network connecting all nodes. Snoop messages are forced to use this network, while other messages can use any links in the network. This general approach is used in IBM's Power 4 [22].

An advantage of this approach is its simplicity and low cost: it places no constraints on the network topology, can use a simple cache coherence protocol, and needs no extra modules like a directory. While it is not scalable to large numbers of processors, it is appropriate for CMP-based machines with 64-128 processors. A drawback is that this approach may induce long snoop latencies or need many snoop operations and additional traffic. For example, if each node snoops a request before passing it on to the next node in the ring, the snoop request takes long to go around the ring. Alternatively, if each node passes on the snoop request, then snoops, and finally sends a reply, we may end up snooping all nodes and generating extra traffic, which is inefficient. In this paper, we address this problem.

2.2. Multi-CMP Multiprocessor with Embedded Ring

Before we address the problem described, we briefly outline the snoopy protocol in the embedded unidirectional ring that we consider. The multiprocessor is built out of CMPs (Figure 2-(a)). Each CMP has several cores, each with a private L2 cache, and is attached to a portion of the shared memory. The CMPs can be interconnected by any physical network, on which we embed one or more unidirectional rings for snoop requests (Figure 2-(a)). If more than one unidirectional ring is embedded, snoop requests may be mapped to different rings according to their memory address. This helps to balance the load on the underlying physical network. Data-transfer messages do not use the logical ring.



(a)

State	Compatible States
I	I, S, S_L, S_G, E, D, T
S	I, S, S_L, S_G, T
S_L	I, S, S_L^*, S_G^*, T^*
S_G	I, S, S_L^*
E	I
D	I
T	I, S, S_L^*

(b)

Figure 2. Machine architecture modeled (a) and matrix of compatible cache states in the coherence protocol used (b). In the network, the darker line shows the embedded ring. In the protocol table, "*" means that a line can be in this state only if it is in a *different* CMP.

We use a MESI coherence protocol [5] enhanced with additional states. In addition to the typical Invalid (I), Shared (S), Exclusive (E), and modified (or Dirty (D)) states of a MESI scheme, we add the Global/Local Master qualifier to the Shared state (S_G and S_L) and the Tagged (T) state.

To understand the Global Master qualifier, consider a set of caches with a Shared line. If a cache outside the set reads the line, at most one of the caches can supply the line — the one with the Global Master qualifier (S_G). In our protocol, the cache that brought the line from memory retains the Global Master qualifier until it evicts the line or gets invalidated.

Since the machine has multiple CMPs, performance would improve if reads were satisfied by a local cache (i.e., one in the same CMP), even if it did not have the Global Master qualifier for that line. Consequently, we allow one cache per CMP to have the Local Master qualifier (S_L). The cache that brings in the line from outside the CMP retains the Local Master qualifier until it evicts the line or gets invalidated.

The T state is used to support the sharing of dirty data. In T state, the line is dirty, but coherent copies can also be found in other caches (in S or S_L state). On eviction, a line in T state is written back to memory.

Figure 2-(b) shows which states are compatible with which other ones. It should be noted that, for each read request, at most one cache (or none) can supply the data. In the following, we give two examples of transactions.

Read Satisfied by Another Cache. When a processor issues a read, the local caches are snooped. If the line is found in S_L, S_G, E, D, or T state, it is supplied. Otherwise, the snoop request *R* is forwarded to the ring. As *R* reaches a node, it enters the CMP and checks all the caches. If no cache has the line in state S_G, E, D, or T, the request moves to the next node, repeating the process. Otherwise, a copy of the line is sent to the requester using the regular routing algorithm (not the snoop ring). In parallel, *R* is marked as a reply message and traverses the remainder of the ring without inducing any more snoops until it reaches the requester. It can be shown that, when the data line (not the snoop reply) reaches the requester, the processor can use it, since it is guaranteed that the transaction will not be squashed. More details on the protocol can be found in [17].

Read Satisfied by Memory. If *R* returns to the requester with a negative response, the requester sends a read message to the memory at the home node. Both this message and the memory reply use the regular routing algorithm. To minimize the latency of this DRAM access, we may choose (with certain heuristics) to initiate a memory prefetch when *R* is snooped at its home node. This would reduce the latency of a subsequent memory access.

In the rest of the paper, we focus on *read* snoop requests. While our contribution also applies to write snoop requests, it is more relevant to reads due to their higher number and criticality.

3. Toward Flexible Snooping

3.1. Eager and Lazy Forwarding

We call the algorithm described in Section 2.2 *Lazy Forwarding* or *Lazy*. The actions of a snoop request in this algorithm are shown in Figure 3-(a). In the figure, a requester node sends a request that snoops at every node until it reaches the supplier node.

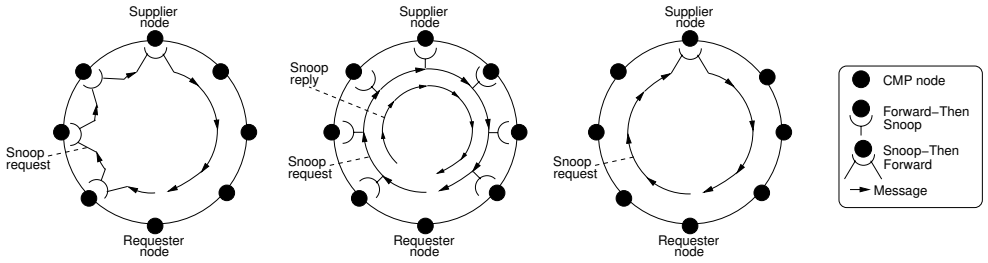


Figure 3. Actions of a snoop request in three different algorithms. Each node is a CMP.

Algorithm	Snoop Request Latency (Unloaded Machine)	Avg. # of Snoop Operations per Snoop Request	Avg. # of Messages per Snoop Request
Lazy Forwarding	High	(N-1)/2	1
Eager Forwarding	Low	N-1	≈ 2
Oracle	Low	1	1
Adaptive Forwarding & Filtering	Between Oracle and Lazy	Between Oracle and Eager	Between Oracle and Eager

Table 1. Comparing different snooping algorithms. The table assumes a perfectly-uniform distribution of the accesses and that one of the nodes can supply the data. *N* is the number of CMP nodes in the machine.

After that, the message proceeds without snooping until it reaches the requester.

Lazy has two limitations: the long latency of snoop requests and the substantial number of snoop operations performed. The first limitation slows down program execution because lines take long to be obtained; the second one results in high energy consumption and may also hinder performance by inducing contention in the CMPs.

Table 1 shows the characteristics of Lazy. If we assume a perfectly-uniform distribution of the accesses and that one of the nodes can supply the data, the supplier is found half-way through the ring. Consequently, the number of snoop operations is (N-1)/2, where N is the number of CMP nodes in the ring.

An alternative is to forward the snoop request from node i to node $i + 1$ before starting the snoop operation on i. We call this algorithm *Eager Forwarding* or *Eager*. It is used by Barroso and Dubois for a slotted ring [2]. For the non-slotted, embedded ring that we consider, we slightly change the implementation.

When a snoop request arrives at a node, it is immediately forwarded to the next one, while the current node initiates a snoop operation. When the snoop completes, the outcome is combined with a *Snoop Reply* message coming from the preceding nodes in the ring, and is then forwarded to the next node (Figure 3-(b)). Some temporary buffering may be necessary for the incoming snoop reply or for the outcome of the local snoop. All along the ring, we have two messages: a snoop request that moves ahead quickly, initiating a snoop at every node, and a snoop reply that collects all the replies.

Table 1 compares Eager to Lazy. Eager reduces snoop request latency. Since the snoop operations proceed in parallel with request forwarding, the supplier node is found sooner, and the data is returned to the requester sooner. However, the disadvantages of Eager are that it causes many snoop operations and messages. Indeed, Eager snoops all the nodes in the ring (N-1). Moreover, what was one message in Lazy, now becomes two — except for the first ring segment (Figure 3-(b)). These two effects increase the energy to service a snoop request.

A third design point is the *Oracle* algorithm of Figure 3-(c). In this case, we know which node is the supplier and only snoop there. As shown in Table 1, Oracle's latency is low and there is a single snoop operation and message.

3.2. Adaptive Forwarding and Filtering

We would like to develop snooping algorithms that have the low request latency of Eager, the few messages per request of Lazy, and the few snoop operations per request of Oracle. Toward this end, we propose *Adaptive Forwarding and Filtering* algorithms (*Adaptive*). These algorithms use two techniques. First, when a node receives a snoop request, depending on the likelihood that it can provide the line, it will perform the snoop operation first and then the request-forwarding operation or vice-versa (Adaptive Forwarding). Second, if the node can prove that it will not be able to provide the line, it will skip the snoop operation (Adaptive Filtering).

Adaptive forwarding is original. Adaptive filtering has been proposed in schemes such as JETTY [14] and Destination-Set Prediction [9], but our proposal is the first to integrate it with adaptive forwarding in a taxonomy of adaptive algorithms.

Adaptive hopes to attain the behavior of Oracle. In practice, for each of the metrics listed in Table 1, *Adaptive* will exhibit a behavior that is somewhere between Oracle and the worst of Lazy and Eager for that metric.

Adaptive works by adding a hardware *Supplier Predictor* at each node that predicts if the node has a copy of the requested line in any of the supplier states (S_G, E, D, and T). When a snoop request arrives at a node, this predictor is checked and, based on the prediction, the hardware performs one of three possible primitive operations: *Forward Then Snoop*, *Snoop Then Forward*, or *Forward*.

Table 2 describes the actions taken by these primitives. At a high level, *Forward Then Snoop* divides a snoop message into a Snoop Request sent before initiating the snoop operation and a Snoop Reply sent when the current node and all its predecessors in the ring have completed the snoop. On the other hand, *Snoop Then Forward* combines the incoming request and reply messages into a single message issued when the current snoop completes. We call this message *Combined Request/Reply (R/R)*. Consequently, in a ring

Primitive	Number of Ring Messages Arriving and Leaving a Node per Request	Action				
			CMP Can Supply Line		CMP Cannot Supply Line	
		Snoop Request or Combined R/R Arrives at Node	Snoop Operation Completes at Node (if Applicable)	Snoop Reply Arrives at Node (if Applicable)	Snoop Operation Completes at Node (if Applicable)	Snoop Reply Arrives at Node (if Applicable)
Forward Then Snoop	Arriving: 1 or 2 Leaving: 2	Forward snoop request, then snoop	Send snoop reply	Discard snoop reply	If had received combined R/R then send snoop reply else wait for snoop reply	Forward snoop reply
Snoop Then Forward	Arriving: 1 or 2 Leaving: 1	Snoop	Send combined R/R	Discard snoop reply	If had received combined R/R then send new combined R/R else wait for snoop reply	Forward it as combined R/R
Forward	Arriving: 1 or 2 Leaving: same as arriving	Forward	N/A	Forward	N/A	Forward

Table 2. Actions taken by each of the primitive operations *Forward Then Snoop*, *Snoop Then Forward*, and *Forward*. In the table, R/R stands for Request/Reply. Recall that at most one CMP has the line in a supplier state.

where different nodes choose a different primitive, a snoop message can potentially be divided into two and recombined multiple times. In all cases, as soon as the requested line is found in a supplier state, the line is sent to the requester through regular network paths.

Consider *Forward Then Snoop* in Table 2. As soon as the node receives a snoop request or a combined R/R, it forwards a snoop request and initiates the snoop operation. If the CMP can supply the line, the node sends a snoop reply message through the ring with this information and the line through regular network paths; if the node later receives a snoop reply message, it simply discards it because it contains no new information[1]. If, instead, the snoop operation shows that the CMP cannot supply the line, the node sends a negative snoop reply (if it had received a combined R/R message) or waits for a reply message (if it had received a request message). In the latter case, when it receives the reply, it augments it with the negative outcome of the local snoop and forwards it. Overall, the node will always send a snoop request and a snoop reply.

On the other hand, with *Snoop Then Forward*, when a node receives a snoop request or a combined R/R, it starts to snoop. If the CMP can supply the line, the node sends a combined R/R message through the ring with this information and the line through regular network paths; if the node later receives a reply from the preceding nodes in the ring, it discards it. If, instead, the snoop shows that the CMP cannot provide the line, the node sends a combined R/R with the negative information (if it had received a combined R/R message) or waits for a reply message (if it had received a request message). In the latter case, when it receives the reply, it augments it as usual and forwards it as a combined R/R. Overall, the node will always send a single message, namely a combined R/R.

The *Forward* primitive simply forwards the two messages (snoop request and reply) or one message (combined R/R) that constitute this request.

It is possible to design different *Adaptive* algorithms by simply choosing between these three primitives at different times or conditions. In the following, we examine the design space of these algorithms.

[1]It only contains the information "I have not been able to find the line", which is already known.

4. Algorithms for Flexible Snooping

4.1. Design Space

To understand the design space for these algorithms, consider the types of Supplier Predictors that exist. One type is predictors that keep a *strict subset* of the lines that are in supplier states in the CMP. These predictors have no false positives, but may have false negatives. They can be implemented with a cache. We call the algorithm that uses these predictors *Subset*.

A second type of predictors are those that keep a *strict superset* of the supplier lines. Such predictors have no false negatives, but may have false positives. They can be implemented with a form of hashing, such as a Bloom filter [3]. We call the algorithm that uses these predictors *Superset*. Predictors of this type have been used in JETTY [14] and RegionScout [13] to save energy in a broadcast-based multiprocessor.

The third type of predictors are those that keep the *exact set* of supplier lines. They have neither false positives nor false negatives. They can be implemented with an exhaustive table. We call the algorithm that uses these predictors *Exact*. Note that there is a fourth type of predictors, namely those that suffer both false positives and false negatives. These predictors are uninteresting because they are less precise than all the other types, while those that suffer either false positives or false negatives are already reasonably inexpensive to implement.

Table 3 compares these algorithms in terms of latency of snoop requests, number of snoop operations, snoop traffic, and implementation difficulty. In the following, we examine them in detail (Section 4.2) and then present implementations (Section 4.3).

4.2. Description of the Algorithms

We first consider the *Subset* algorithm (Table 3). On a positive prediction, since the supplier is guaranteed to be in the CMP, the algorithm uses *Snoop Then Forward*. On a negative prediction, since there is still chance that the supplier is in the CMP, the algorithm selects *Forward Then Snoop*. The latency of the snoop requests is low because requests are not slowed down by any snoop operation as they travel from the requester to the supplier nodes. The number of snoops is higher than in Lazy because at least all the CMPs up to the supplier are snooped. In addition, if the supplier node is falsely predicted negative, more snoops may occur on subsequent nodes. Consequently, Table 3 shows that the number of snoops is *Lazy* +

Algorithm		False Pos?	False Neg?	Action If Predict Positive	Action If Predict Negative	Characteristics			
						Snoop Request Latency (Unloaded Machine)	Avg # Snoop Operations per Snoop Request	Avg # Msgs per Snoop Request	Implementation Difficulty
Subset		N	Y	*Snoop Then Forward*	*Forward Then Snoop*	Low	Medium (Lazy + $\alpha \times FN$)	1-2	Low
Superset	*Con*	Y	N	*Snoop Then Forward*	*Forward*	Medium	Low ($1 + \alpha \times FP$)	1	Medium
	Agg			*Forward Then Snoop*	*Forward*	Low	Low ($1 + \alpha \times FP$)	1-2	Medium
Exact		N	N	*Snoop Then Forward*	*Forward*	Low	1	1	Medium

Table 3. Proposed Flexible Snooping algorithms. This characterization assumes that one of the nodes supplies the data and, therefore, the request does not go to memory. In the table, FN and FP stand for the number of false negatives and false positives, respectively.

$\alpha\times$ *FalseNegatives*. The number of messages per snoop request is between 1 and 2 because negative predictions produce 2 messages but the positive prediction combines them.

Figure 4-(a) shows, in a shaded pattern, the design space of Flexible Snooping algorithms in a graph of snoop request latency versus the number of snoop operations per request. The figure also shows the placement of the baseline algorithms. Figure 4-(b) repeats the figure and adds the data points corresponding to the algorithms proposed. Based on the previous discussion, we place *Subset* in the Y axis and above Lazy.

Consider the *Superset* algorithm in Table 3. On a negative prediction, since there are no false negatives, it uses *Forward*. On a positive prediction, since there is still a chance that the supplier is not in the CMP, we have two choices. A conservative approach (*Superset Con*) assumes that the CMP has the supplier and performs *Snoop Then Forward*. An aggressive approach (*Superset Agg*) performs *Forward Then Snoop*. The latency of the snoop requests is medium in *Superset Con* because false positives introduce snoop operations in the critical path of getting to the supplier node; the latency is low in *Superset Agg* because requests are not slowed down by any snoop operation as they travel to the supplier node. The number of snoops is low in both algorithms: 1 plus a number proportional to the number of false positives. Such term is lower in *Superset Con* than in *Superset Agg*: *Superset Con* only checks the Supplier Predictor in the nodes between the requester and the supplier, while *Superset Agg* checks the Supplier Predictor in all the nodes. The number of messages per snoop request is 1 in *Superset Con* and 1-2 in *Superset Agg*. Based on this analysis, we place these algorithms in the design space of Figure 4-(b).

The *Exact* algorithm uses *Snoop Then Forward* on a positive prediction and *Forward* in a negative one (Table 3). Since it has perfect prediction, the snoop request latency is low and the number of snoops and messages is 1. Figure 4-(b) places it in the origin with the Oracle algorithm.

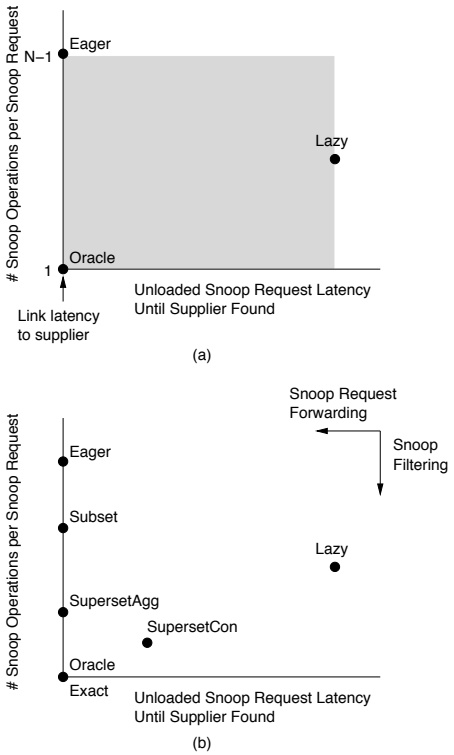


Figure 4. Design space of Flexible Snooping algorithms according to the snoop request latency and the number of snoop operations per request. Chart (a) shows the baseline algorithms, with the shade covering the area of Flexible Snooping algorithms. Chart (b) places the proposed algorithms. The charts assume that one of the nodes provides the line.

4.3. Implementation of the Algorithms

The proposed snoopy algorithms enhance the gateway module of each CMP (Figure 2-(a)) with a hardware-based Supplier Predictor. When a snoop request arrives at the CMP, the predictor is checked. The predictor predicts whether the CMP contains the requested line in any of the supplier states (S_G, E, D, or T). Based on the predictor's outcome and the algorithm used, an action from Table 3 is taken. Next, we describe possible implementations of the Supplier Predictors we study.

4.3.1. Subset Algorithm

The predictor for *Subset* can be implemented with a set-associative cache that contains the addresses of lines known to be in supplier states in the CMP (Figure 5-(a)). When a line is brought to the CMP in a supplier state, the address is inserted into the predictor. If possible, the address overwrites an invalid entry in the predictor. If all the lines in the corresponding set are valid, the LRU one is overwritten. This opens up the possibility of false negatives.

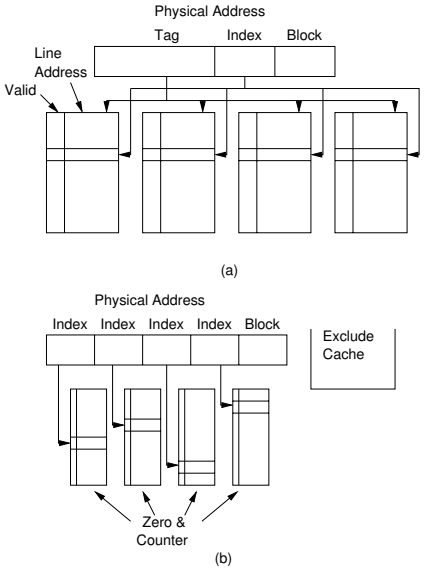


Figure 5. Implementation of the Supplier Predictor.

A line in a supplier state in the CMP can only lose its state if it is evicted or invalidated. Note that, at any time, only one copy of a given line can be in a supplier state. Consequently, when any of these lines is evicted or invalidated, the hardware removes the address from the Supplier Predictor if it is there. This operation eliminates the possibility of false positives.

4.3.2. Superset Algorithm

The predictor for the two *Superset* algorithms of Table 3 can be implemented with a Bloom filter [3] (Figure 5-(b)). The Bloom filter is implemented by logically breaking down the line address into *P* fields. The bits in each field index into a separate table. Each entry in each table has a count of the number of lines in a supplier state in the CMP whose address has this particular bit combination. Every time that a line in supplier state is brought into the CMP, the corresponding *P* counters are incremented; when one such line is evicted or invalidated, the counters are decremented. When a snoop request is received, the address requested is hashed and the corresponding entries inspected. If at least one of the counters is zero, the address is guaranteed not to be in the CMP. However, due to aliasing, this scheme can incur false positives.

To reduce the number of false positives, we can follow the JETTY design [14] and augment the Bloom filter with an Exclude cache. The latter is a set-associative cache storing the addresses of lines that are known not to be in supplier states in the CMP. Every time a false positive is detected, the corresponding address is inserted in the Exclude cache. Every time that a line in supplier state is brought into the CMP, the Exclude cache is checked and potentially one of its entries is invalidated. With this support, when a snoop request is received, its address is checked against both the Bloom filter and the Exclude cache. If one the counters in the filter is zero or the address is found in the Exclude cache, the prediction is declared negative.

4.3.3. Exact Algorithm

The predictor for *Exact* can be implemented by enhancing the *Subset* design of Section 4.3.1. We eliminate false negatives as follows: every time that a valid entry in the Supplier Predictor is overwritten due to a conflict, the hardware downgrades the supplier state of the corresponding line in the CMP to a non-supplier state. Specifically, if the line is in S_G or E, it is silently downgraded to S_L; if the line is in D or T state, the line is written back to memory and kept cached in S_L state.

This support eliminates false negatives but can hurt the performance of the application. Specifically, a subsequent snoop request for the downgraded line from any node will have to be serviced from memory. Moreover, if the downgraded cache attempts to write one of the lines downgraded from E, D, or T, it now needs one more network transaction to upgrade the line to its previous state again before the write can complete.

Overall, the performance impact of this implementation depends on two factors: the size of the predictor table relative to the number of *supplier lines* in a CMP, which affects the amount of downgrading, and the program access patterns, which determine whether or not the positive effects of downgrading dominate the negative ones.

4.3.4. Difficulty of Implementation

We claim in Table 3 that *Superset* and *Exact* are more difficult to implement than *Subset*. The reason is that these algorithms have no false negatives, whereas *Subset* has no false positives. To see why this matters, assume that a hardware race induces an unnoticed false negative in *Superset* and *Exact*, and an unnoticed false positive in *Subset*. In the first case, a request skips the snoop operation at the CMP that has the line in supplier state; therefore, execution is incorrect. In the second case, the request unnecessarily snoops a CMP that does not have the line; therefore, execution is slower but still correct.

Consequently, implementations of *Superset* and *Exact* have to guarantee that no hardware race ever allows a false negative to occur. Such races can occur at two time windows. The first window is between the time the CMP receives a line in supplier state and the time the *Superset* or *Exact* predictor tables are updated to include the line. Note that the line may be received from local memory (Figure 2-(a)) or through other network links that do not go through the gateway. The second race window is between the time the *Exact* predictor table removes an entry due to a conflict and the time the corresponding line in the CMP is downgraded. Careful design of the logic involved will ensure that these races are eliminated.

5. Evaluation Methodology

5.1. Architecture and Applications Evaluated

We evaluate the proposed algorithms for Flexible Snooping using detailed simulations of out-of-order processors and memory subsystems. The baseline system architecture is a multiprocessor

Processor and Private Caches		CMP	Global Network	Memory
Frequency: 6.0 GHz Branch penalty: 17 cyc (min) RAS: 32 entries BTB: 2K entries, 2-way assoc. Branch predictor: bimodal size: 16K entries gshare-11 size: 16K entries Int/FP registers: 176/130	Fetch/issue/comm width: 6/4/4 I-window/ROB size: 80/176 LdSt/Int/FP units: 2/3/2 Ld/St queue entries: 64/64 D-L1 size/assoc/line: 32KB/4-way/64B D-L1 RT: 2 cyc L2 size/assoc/line: 512KB/8-way/64B L2 RT: 11 cyc	Processors/CMP: SPLASH-2: 4 SPECjbb, SPECweb: 1 Per-processor L2 Intra-CMP network: Topology: shared bus Bandwidth: 64 GB/s RT to another L2: 55 cyc	Number of chips: 8 Embedded ring network: CMP to CMP latency: 39 cyc Link bandwidth: 8GB/s CMP bus access & L2 snoop time: 55 cyc Data network: Topology: 2D torus Link bandwidth: 32GB/s	RT to local memory: 350 cyc RT to remote memory (max): With prefetch: 312 cyc Without prefetch: 710 cyc Main memory: Frequency: 667MHz Width: 128bit DRAM bandwidth: 10.7GB/s DRAM latency: 50ns

Subset Predictor	*Superset Con* or *Superset Agg* Predictor		*Exact* Predictor
Size: 512, 2K, or 8K entries Entry size: 20, 18, or 16 bits Total size: 1.3, 4.8, or 17KB Access time: 2, 2, or 3 cyc Assoc: 8-way Ports: 1	Bloom filter: *n* filter: Fields: 9,9,6 bits. Total size: 2.3KB *y* filter: Fields: 10,4,7 bits. Total size: 2.5KB Size of filter entry: 16 bits + Zero bit Access time: 2 cyc	Exclude cache: Size: 512 or 2K entries Entry size: 20 or 18 bits Total size: 1.3 or 4.8KB Access time: 2 cyc Assoc: 8-way Ports: 1	Size: 512, 2K, or 8K entries Entry size: 20, 18 or 16 bits Total size: 1.3, 4.8 or 17KB Access time: 2, 2, or 3 cyc Assoc: 8-way Ports: 1

Table 4. Architectural parameters used. In the table, RT means minimum Round-Trip time from the processor. Cycle counts are in processor cycles.

with 8 CMPs, where each CMP has 4 processors (Figure 2-(a)). The CMPs are interconnected with a 2-dimensional torus, on which we embed two unidirectional rings for snoop messages. The snoop requests and replies are assigned to rings based on their address. The interconnect inside each CMP is a high-bandwidth shared bus. The cache coherence protocol used is described in Section 2.2. Table 4 details most of the architectural parameters used.

We estimate that a message in the embedded ring network needs 55 cycles at 6 GHz to access the CMP bus and snoop the caches. This includes 38 cycles for on-chip network transmission (transmission from gateway to arbiter, from arbiter to L2 caches, and from L2 caches to gateway), 10 cycles for on-chip network arbitration, and 7 cycles for L2 snooping plus buffering. All on-chip L2 caches are snooped in parallel. These numbers are consistent with those in [8].

We run 11 SPLASH-2 applications [26] (all except Volrend, which has calls that our simulator does not support), SPECjbb 2000, and SPECweb 2005 [20]. The SPLASH-2 applications run with 32 processors (8 CMPs of 4 cores each). Due to limitations in our simulation infrastructure, we can only run the SPECjbb and SPECweb workloads with 8 processors; we assume that they are in 8 different CMPs. The SPLASH-2 applications are simulated in execution-driven mode by SESC [15], while the SPEC applications are simulated in trace-driven mode by a combination of Simics [23] and SESC.

The SPLASH-2 applications use the input sets suggested in [26]. SPECjbb runs with 8 warehouses and is measured for over 750 million instructions after skipping initialization. SPECweb runs with the e-commerce workload for over 750 million instructions after skipping initialization. Since the SPEC workloads are simulated in trace-driven mode, we compare the different snooping algorithms with exactly the same traces, and do not re-run multiple instances of the same workload.

5.2. Supplier Predictors Used

For our experiments, we use three different Supplier Predictors for each of our Flexible Snooping algorithms. The predictors used are shown in the lower part of Table 4. The three predictors for *Subset* are an 8-way cache with either 512, 2K, or 8K entries. We call them *Sub512*, *Sub2k*, and *Sub8k*.

The three predictors for *Superset Con* and *Superset Agg* are as follows: *y512* has the *y* Bloom filter of Table 4 and a 512-entry Exclude cache; *y2k* has the same Bloom filter and a 2K Exclude cache; and *n2k* has the *n* Bloom filter of Table 4 and a 2K Exclude cache. We call the resulting predictors *SupCy512*, *SupCy2k*, and *SupCn2k* for the Conservative algorithm and *SupAy512*, *SupAy2k*, and *SupAn2k* for the Aggressive one.

Finally, the predictors for *Exact* are an 8-way predictor cache with 512, 2K, or 8K entries. We call them *Exa512*, *Exa2k*, and *Exa8k*.

5.3. Handling Write Snoop Requests

Write snoop requests cannot use our predictors of Section 4.3. The reason is that writes need to invalidate *all* the cached copies of a line. As a result, they would need a predictor of line *presence*, rather than one of line in supplier state.

In our simulations, we handle write snoop requests as follows. Recall from Table 3 that our Flexible Snooping algorithms fall into two classes: those that do not decouple read snoop messages into request and reply (*Superset Con* and *Exact*, together with *Lazy*), and those that do (*Subset* and *Superset Agg*, together with *Eager*). Consequently, for the former, we do not decouple write snoop messages either. For the latter, we think it is fair to always decouple write snoops into request and reply — it enables parallel invalidation of the nodes. Also, since we are not concerned with the implementation feasibility of *Oracle* and we use it to estimate the potential performance improvement of our techniques, we allow write snoop messages to decouple for *Oracle* as well. In any case, writes are less critical and numerous than reads.

6. Evaluation

In this section, we first compare our Flexible Snooping algorithms to each other and to Lazy, Eager, and Oracle. Then, we per-

form a brief sensitivity analysis of the impact of various Supplier Predictor organizations.

6.1. Comparison of Flexible Snooping Algorithms

For our main comparison between the schemes, we use the *Sub2k*, *SupCy2k*, *SupAy2k*, and *Exa2k* predictors for the *Subset*, *Superset Con*, *Superset Agg*, and *Exact* algorithms, respectively. In all cases, the per-node predictors have 2K entries in their cache or Exclude cache — although the Superset algorithms additionally add a Bloom filter. The predictor sizes are 4.8 Kbytes for *Subset* and *Exact*, and 7.3 Kbytes for the *Superset* algorithms.

In the following, we compare our Flexible Snooping algorithms to Lazy, Eager, and Oracle along four dimensions: number of snoop operations per read snoop request, number of read messages in the ring, total execution time, and energy consumption. We consider each dimension in turn.

6.1.1. Number of Snoops per Read Request

The number of snoop operations per read snoop request for the different algorithms is shown in Figure 6. The figure shows absolute values for the arithmetic mean of SPLASH-2 applications, SPECjbb, and SPECweb.

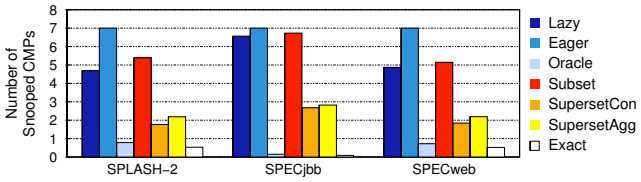


Figure 6. Average number of snoop operations per read snoop request for different algorithms.

The figure shows that *Eager* snoops the most. As expected, it snoops all 7 CMPs in every request. As a result, it consumes significant energy. If there is a supplier node, *Lazy* should snoop on average about half of the nodes, namely 3-4. In practice, since many requests do not find a supplier node and need to go to memory, *Lazy* snoops more. In particular, in SPECjbb, threads do not share much data, and many requests go to memory. For this workload, *Lazy* incurs an average number of snoops close to 7.

The relative snoop frequency of the Flexible Snooping algorithms follows the graphical representation of Figure 4-(b). For example, *Subset* snoops slightly more than *Lazy*. As indicated in Table 3, its snoops over *Lazy* depend on the number of false negatives. On the other hand, the *Superset* algorithms have many fewer snoops, typically 2-3. As indicated in Figure 4-(b), *Superset Con* snoops slightly less than *Superset Agg*.

Oracle has a very low value. Its number of snoops is less than one because when the line needs to be obtained from memory, *Oracle* does not snoop at all. Finally, *Exact* is very close to *Oracle*. It has in fact fewer snoops than *Oracle* because, as indicated in Section 4.3.3, its Supplier Predictor induces some line downgrades. These downgrades result in relatively fewer lines supplied by caches.

6.1.2. Number of Read Messages in the Ring

The total number of read snoop requests and replies in the ring for the different algorithms is shown in Figure 7. In the figure, the bars for SPLASH-2 correspond to the geometric mean. Within an application, the bars are normalized to *Lazy*.

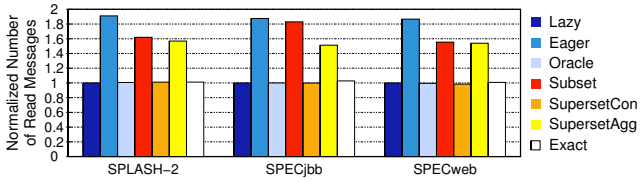


Figure 7. Total number of read snoop requests and replies in the ring for different algorithms. The bars are normalized to *Lazy*.

The figure shows that *Eager* has the most read messages in the ring. As indicated in Table 1, *Eager* generates nearly twice the number of messages of *Lazy*. The number is not exactly twice because request and reply travel together in the first ring segment (Figure 3-(b)). In any case, *Eager* consumes a lot of energy in the ring. *Lazy*, on the other hand, uses a single message per request and, therefore, is frugal.

The relative number of messages in Flexible Snooping algorithms follows the discussion in Table 3. The number of messages in *Subset* and *Superset Agg* is between that of *Eager* and *Lazy*. The reason is that, while *Subset* often produces two messages per request, it merges them when it predicts that a line can be supplied by the local node. *Superset Agg* allows the request and the reply to travel in the same message until it first predicts that the line may be supplied by the local node. From Figure 7, we see that these schemes induce a similar number of messages — except in SPECjbb.

Finally, as indicated in Table 3, *Superset Con* and *Exact* have the same number of read messages as *Lazy* (and *Oracle*). This gives these schemes an energy advantage. The figure also shows that downgrades do not seem to affect the number of read messages in *Exact* — only the location from where they are supplied.

6.1.3. Total Execution Time

The total execution time of the applications for the different algorithms is shown in Figure 8. In the figure, the SPLASH-2 bars show the geometric mean of the applications. The bars are normalized to *Lazy*.

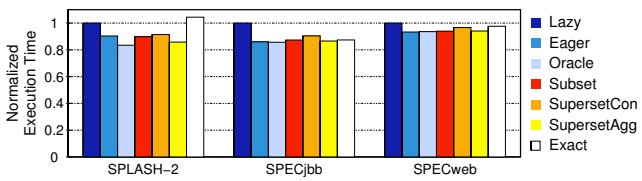


Figure 8. Execution time of the applications for different algorithms. The bars are normalized to *Lazy*.

To understand the results, consider Figure 4-(b), which qualitatively shows the relative snoop request latency in the different

algorithms. In there, *Lazy* has the longest latency, while *Superset Con* has somewhat longer latency than the other Flexible Snooping algorithms.

Figure 8 is consistent with these observations. The figure shows that, on average, *Lazy* is the slowest algorithm, and that most of the other algorithms track the performance of *Eager*. Of the Flexible Snooping schemes, *Exact* is slow when running SPLASH-2 and, to a lesser extent, SPECweb. This is because it induces downgrades in these workloads, which result in more requests being satisfied by main memory than before. On the other hand, *Exact* does not hurt the performance of SPECjbb because many accesses in this workload would not find the data in caches anyway.

Among the remaining three Flexible Snooping algorithms, *Superset Con* is the slightly slower one, as expected. When it runs the SPEC workloads, it suffers delays caused by false positives, which induce snoop operations in the critical path.

Superset Agg is the fastest algorithm. It is always very close to *Oracle*, which is a lower execution bound. On average, *Superset Agg* is faster than *Eager*. Compared to *Lazy*, it reduces the execution time of the SPLASH-2, SPECjbb, and SPECweb workloads by 14%, 13%, and 6%, respectively.

6.1.4. Energy Consumption

Finally, we compute the energy consumed by the different algorithms. We are interested in the energy consumed by the read and write snoop requests and replies. Consequently, we add up the energy spent snooping nodes other than the requester, accessing and updating the Supplier Predictors, and transmitting request and reply messages along the ring links. In addition, for *Exact*, we also add the energy spent downgrading lines in caches and, most importantly, the resulting additional cache line write backs to main memory and eventual re-reads from main memory. These accesses are counted because they are a direct result of *Exact*'s operation.

To estimate this energy we use several tools. We use models from CACTI [18] to estimate the energy consumed accessing cache structures (when snooping or downgrading lines) and predictors. We use Orion [25] to estimate the energy consumed to access the on-chip network. We use the HyperTransport I/O Link Specification [7] to estimate the energy consumed by the transmission of messages in the ring interconnect. Finally, we use Micron's System-Power Calculator [12] to estimate the energy consumed in main memory accesses.

As an example of the numbers obtained, transferring one snoop request message on a single ring link is estimated to consume 3.17 nJ. In contrast, the energy of a CMP snoop is estimated to be only 0.69 nJ. We can see, therefore, that a lot of the energy is dissipated in the ring links. Finally, reading a line from main memory is estimated to consume 24 nJ.

Figure 9 shows the resulting energy consumption for the different algorithms. As usual, the SPLASH-2 bar is the geometric mean of the applications, and all bars are normalized to *Lazy*. The figure shows that *Eager* consumes about 80% more energy than *Lazy*. This is because it needs more messages and more snoop operations than *Lazy*. Of the Flexible Snooping algorithms, *Subset* and *Superset Agg* are also less efficient than *Lazy*, as they induce more messages than *Lazy* and, in the case of *Subset*, more snoop operations as well. Still, *Superset Agg* consumes 14%, 17%, and 9% less energy than *Eager* for SPLASH-2, SPECjbb, and SPECweb, respectively.

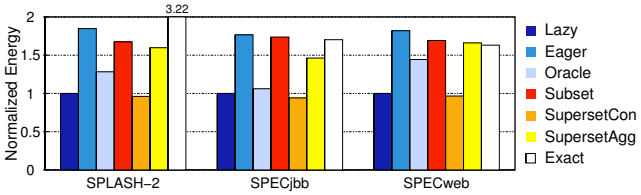


Figure 9. Energy consumed by read and write snoop requests and replies for different algorithms. The bars are normalized to *Lazy*.

Exact is not an attractive algorithm energy-wise. While it needs few snoop messages and snoop operations, it induces cache-line downgrading. As indicated in Section 4.3.3, some of these lines need to be written back to memory and later re-read from memory. As shown in Figure 9, this increases the energy consumption significantly, especially for applications with frequent cache-to-cache transfers such as SPLASH-2.

Finally, *Superset Con* is the most efficient algorithm. Its energy consumption is the lowest, thanks to needing the same number of messages as *Lazy* and fewer snoop operations. Compared to *Lazy*, however, it adds the energy consumed in the predictors. In particular, the predictors used by the *Superset* algorithms consume substantial energy in both training and prediction. As a result, *Superset Con*'s energy is only slightly lower than *Lazy*'s. Compared to *Eager*, however, it consumes 48%, 47%, and 47% less energy for SPLASH-2, SPECjbb, and SPECweb, respectively.

6.1.5. Summary of Results

Based on this analysis, we argue that *Superset Agg* and *Superset Con* are the most cost-effective algorithms. *Superset Agg* is the choice algorithm for a high-performance system. It is the fastest algorithm — faster than *Eager* while consuming 9-17% less energy than *Eager*. For an energy-efficient environment, the choice algorithm is *Superset Con*. It is only 3-6% slower than *Superset Agg* while consuming 36-42% less energy.

Interestingly, both *Superset Con* and *Superset Agg* use the same Supplier Predictor. The only difference between the two is the action taken on a positive prediction (Table 3). Therefore, we envision an adaptive system where the action is chosen dynamically. Typically, the action would be that of *Superset Agg*. However, if the system needs to save energy, it would use the action of *Superset Con*.

6.2. Sensitivity Analysis

In this section, we evaluate the impact of the Supplier Predictor size on the performance of the Flexible Snooping algorithms. We evaluate the predictors described in Section 5.2, namely *Sub512*, *Sub2k*, and *Sub8k* for *Subset*; *SupCy512*, *SupCy2k*, and *SupCn2k* for *Superset Con*; *SupAy512*, *SupAy2k*, and *SupAn2k* for *Superset Agg*; and *Exa512*, *Exa2k*, and *Exa8k* for *Exact*.

Figure 10 compares the execution time of the workloads for the different Supplier Predictor sizes and organizations. For each algorithm and application, we normalize the three bars to the one for the predictor used in Section 6.1 (in all cases, the central bar).

From the figure, we see that these environments are not very sensitive to the size and organization of the Supplier Predictor —

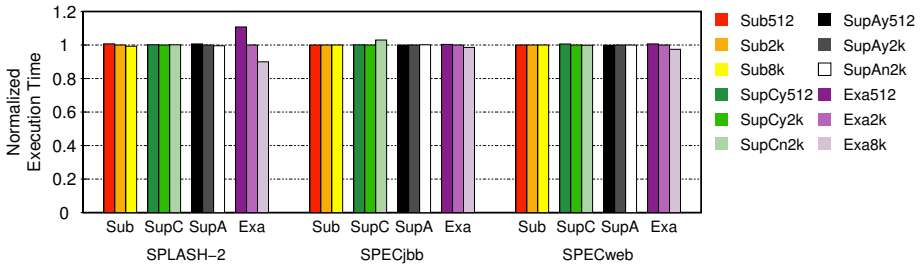


Figure 10. Execution time of *Subset*, *Superset Con*, *Superset Agg*, and *Exact* algorithms with different Supplier Predictor sizes and organizations.

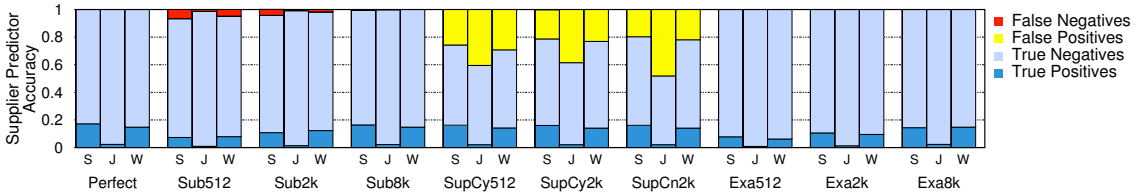


Figure 11. Supplier Predictor accuracy for the different implementations of the *Subset*, *Superset*, and *Exact* algorithms. In the figure, *S*, *J*, and *W* stand for SPLASH-2, SPECjbb, and SPECweb, respectively.

at least for the ranges considered. The only exception is the *Exact* algorithm for the SPLASH-2 applications. In this case, larger Supplier Predictors reduce the execution time noticeably. The reason is that small predictor caches cause many line downgrades. This hurts performance in applications with significant cache-to-cache data transfers such as SPLASH-2. Overall, based on Figure 10, we conclude that the predictor sizes used in Section 6.1 represent good design points.

To gain further insight into how these predictors work, Figure 11 shows the fraction of true positive, true negative, false positive, and false negative predictions issued by read snoop requests using each of these Supplier Predictors. The figure includes a perfect predictor that is checked by the snoop request at every node, until the request finds the supplier node. The predictors for the two *Superset* algorithms behave very similarly and, therefore, we only show one of them. In the figure, *S*, *J*, and *W* stand for SPLASH-2, SPECjbb, and SPECweb, respectively.

The perfect predictor in Figure 11 shows how soon a read snoop request finds the supplier node. Specifically, in SPLASH-2 and SPECweb, the predictor makes about 4 negative predictions for every positive prediction. This means that the supplier is found on average about 5 nodes away from the requester. In SPECjbb, however, there is rarely a supplier node, and the request typically gets the line from memory.

The next three sets of bars show that the *Subset* predictors have few false negatives. As we increase the size of the predictor from 512 entries to 8K entries, the number of false negatives decreases. For 8K entries, false negatives practically disappear.

On the other hand, the *Superset* predictors in the next three sets of bars have a significant fraction of false positives. For the best configuration (*SupCy2k*), false positives account for 20-40% of the predictions. While we have tried many different bit-field organizations for the Bloom filter, we find that it is difficult to reduce the number of false positives. The Exclude cache helps for SPLASH-2 and for SPECweb. However, it does not help for SPECjbb, where there are few cache-to-cache transfers — as seen from the perfect predictor. Since there are few cache-to-cache transfers, the Exclude cache thrashes and never helps.

Finally, the bars for the *Exact* predictors give an idea of the impact of downgrades. In Figure 11, the difference between these predictors and the perfect one is due to downgrades. Specifically, the more downgrades issued, the lower the fraction of true positives. We can see from the figure that, for an 8K entry predictor cache, the effect of the downgrades is relatively small. However, as we decrease the predictor size, more downgrades mean a lower fraction of true positives.

7. Related Work

Our work is related to several schemes that improve performance or energy consumption in coherence protocols. In Destination-Set Prediction [9], requester caches predict which other caches need to observe a certain memory request. Unlike our proposal, the prediction is performed at the source node, rather than at the destination node. Moreover, destination-set prediction targets a multicast network environment. It leverages specific sharing patterns like pairwise sharing to send multicasts to only a few nodes in the system.

JETTY [14] is a filtering proposal targeted at snoopy bus-based SMP systems. A data presence predictor is placed between the shared bus and each L2 cache, and filters part of the snoops to the L2 tag arrays. The goal of the mechanism is exclusively to save energy. While we used one of the structures proposed by JETTY, our work is more general: we leverage snoop forwarding in addition to snoop filtering; we use a variety of structures; we use these

techniques to improve both performance and energy; and finally we use a supplier predictor.

Power Efficient Cache Coherence [16] proposes to perform snoops serially on an SMP with a shared hierarchical bus. Leveraging the bus hierarchy, close-by caches are snooped in sequence, potentially reducing the number of snoops necessary to service a read miss. Our work is different in the following ways: our work focuses on a ring, on which we detail a race-free coherence protocol; we present a family of adaptive protocols; finally we focus on both high performance and low energy.

Ekman et al [6] evaluate JETTY and serial snooping in the context of a cache-coherent CMP (private L1 caches and shared L2 cache) and conclude these schemes are not appropriate for this kind of environment.

Owner prediction has been used to speed-up cache-to-cache interventions in a CC-NUMA architecture [1]. The idea is to shortcut the directory lookup latency in a 3-hop service by predicting which cache in the system would be able to supply the requested data and sending the request directly to it, only using the home directory to validate the prediction.

Barroso and Dubois [2] propose the use of a slotted ring as a substitute for a bus in an SMP system. As indicated in Section 2.1.4, their work is different in that they look at a ring network topology (while we use a logically-embedded ring) and that they use slotting (while we do not have these timing constraints). They use the Eager algorithm, which we use as a baseline here. Another system that uses a slotted ring topology is Hector [24]. Hector uses a hierarchy of rings.

Moshovos [13] and Cantin *et al*. [4] propose coarse-grain mechanisms to filter snoops on memory regions private to the requesting node. They differ from our work in that they are source-filtering mechanisms. In addition, these mechanisms work at a coarser granularity and target only a certain category of misses (cold misses or misses to private data). These techniques may be combined with our techniques to further improve performance and energy savings.

8. Conclusions

While snoopy protocols using logically-embedded rings in the network are simple and low cost, straightforward implementations may suffer from long snoop request latencies or many snoop messages and operations. To address this problem, this paper made three contributions. First, it introduced *Flexible Snooping* algorithms, a family of adaptive forwarding and filtering snooping algorithms, and described the primitive operations they rely on. Second, it analyzed the design space of these algorithms and described four general approaches, namely *Subset*, *Superset Con*, *Superset Agg*, and *Exact*. These approaches have different trade-offs in number of snoop operations and messages, snoop response time, energy consumption, and implementation difficulty.

Finally, we used SPLASH-2, SPECjbb, and SPECweb workloads to evaluate these approaches. Our analysis found several snooping algorithms that are more cost-effective than current ones. Specifically, our choice for a high-performance snooping algorithm (*Superset Agg* with a 7.3-Kbyte per-node predictor) was faster than the currently fastest algorithm (*Eager*), while consuming 9-17% less energy; moreover, our choice for an energy-efficient algorithm (*Superset Con* with the same predictor) was only 3-6% slower than *Superset Agg*, while consuming 36-42% less energy.

Acknowledgments

We thank the anonymous reviewers and members of the I-ACOMA group for their valuable comments. Special thanks go to Luis Ceze, Paul Sack, Smruti Sarangi and James Tuck for their insights and help with the simulation infrastructure and energy consumption estimations. Karin Strauss thanks IBM Research for her internship at the Scalable Server Network and Memory System Department at the IBM T. J. Watson Research Center.

References

[1] M. E. Acacio, J. González, J. M. García, and J. Duato. Owner Prediction for Accelerating Cache-to-Cache Transfer Misses in a cc-NUMA Architecture. In *High Performance Computing, Networks and Storage Conference (SC)*, Nov 2002.

[2] L. Barroso and M. Dubois. The Performance of Cache-Coherent Ring-based Multiprocessors. In *International Symposium on Computer Architecture*, May 1993.

[3] B. Bloom. Space/time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 11(7):422–426, July 1970.

[4] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Improving Multiprocessor Performance with Coarse-Grain Coherence Tracking. In *International Symposium on Computer Architecture*, June 2005.

[5] D. E. Culler and J. P. Singh. *Parallel Computer Architecture; A Hardware/Software Approach*. Morgan Kaufmann, 1999.

[6] M. Ekman, F. Dahlgren, and P. Stenström. Evaluation of Snoop-Energy Reduction Techniques for Chip-Multiprocessors. In *Workshop on Duplicating, Deconstructing, and Debunking*, May 2002.

[7] HyperTransport Technology Consortium. *HyperTransport I/O Link Specification*, 2.00b edition, April 2005.

[8] R. Kumar, V. Zyuban, and D. M. Tullsen. Interconnections in Multi-Core Architectures: Understanding Mechanisms, Overheads and Scaling. In *International Symposium on Computer Architecture*, June 2005.

[9] M. Martin, P. Harper, D. Sorin, M. Hill, and D. Wood. Using Destination-Set Prediction to Improve the Latency/Bandwidth Tradeoff in Shared-Memory Multiprocessors. In *International Symposium on Computer Architecture*, June 2003.

[10] M. Martin, M. Hill, and D. Wood. Token Coherence: Decoupling Performance and Correctness. In *International Symposium on Computer Architecture*, June 2003.

[11] M. Marty, J. Bingham, M. Hill, A. Hu, M. Martin, and D. Wood. Improving Multiple-CMP Systems Using Token Coherence. In *International Symposium on High-Performance Computer Architecture*, Feb 2005.

[12] Micron Technology, Inc. *System-Power Calculator*. http://www.micron.com/products/dram/syscalc.html.

[13] A. Moshovos. RegionScout: Exploiting Coarse Grain Sharing in Snoop-Based Coherence. In *International Symposium on Computer Architecture*, June 2005.

[14] A. Moshovos, G. Memik, B. Falsafi, and A. Choudhary. JETTY: Filtering Snoops for Reduced Energy Consumption in SMP Servers. In *International Symposium on High-Perfomance Computer Architecture*, Jan 2001.

[15] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, K. Strauss, S. Sarangi, P. Sack, and P. Montesinos. SESC Simulator, Jan 2005. http://sesc.sourceforge.net.

[16] C. Saldanha and M. Lipasti. Power Efficient Cache Coherence. In *Workshop on Memory Performance Issues*, June 2001.

[17] X. Shen. A Snoop-and-Forward Cache Coherence Protocol for SMP Systems with Ring-based Address Networks. Technical report, IBM T. J. Watson Research Center, June 2004.

[18] P. Shivakumar and N. Jouppi. CACTI 3.0: An Integrated Cache Timing, Power and Area Model. Technical Report 2001/2, Compaq Computer Corporation, Aug 2001.

[19] Silicon Graphics. Silicon Graphics Altrix 3000 Scalable 64-bit Linux Platform. http://www.sgi.com/products/servers/altix/.

[20] Standard Performace Evaluation Corporation (SPEC). http://www.spec.org.

[21] Sun Microsystems. Sun Enterprise 10000 Server Overview. http://www.sun.com/servers/highend/e10000/.

[22] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy. POWER4 System Microarchitecture. In *IBM Journal of Research and Development*, Jan 2002.

[23] Virtutech. Virtutech Simics. http://www.virtutech.com/products.

[24] Z. Vranesic, M. Stumm, D. Lewis, and R. White. Hector: A Hierarchically Structured Shared-Memory Multiprocessor. In *IEEE Computer Magazine*, Jan 1991.

[25] H. S. Wang, X. P. Zhu, L. S. Peh, and S. Malik. Orion: A Power-Performance Simulator for Interconnection Networks. In *International Symposium on Microarchitecture*, Nov 2002.

[26] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *International Symposium on Computer Architecture*, June 1995.