# Multicast Snooping: A New Coherence Method Using a Multicast Address Network

E. Ender Bilir, Ross M. Dickson, Ying Hu, Manoj Plakal, Daniel J. Sorin,
Mark D. Hill, David A. Wood
Computer Sciences Department
University of Wisconsin-Madison
{enderb,dickson,yhu,plakal,sorin,markhill,david}@cs.wisc.edu

## Abstract

*This paper proposes a new coherence method called "multicast snooping" that dynamically adapts between broadcast snooping and a directory protocol. Multicast snooping is unique because processors predict which caches should snoop each coherence transaction by specifying a multicast "mask." Transactions are delivered with an ordered multicast network, such as an Isotach network, which eliminates the need for acknowledgment messages. Processors handle transactions as they would with a snooping protocol, while a simplified directory operates in parallel to check masks and gracefully handle incorrect ones (e.g., previous owner missing). Preliminary performance numbers with mostly SPLASH-2 benchmarks running on 32 processors show that we can limit multicasts to an average of 2-6 destinations (<< 32) and we can deliver 2-5 multicasts per network cycle (>> broadcast snooping's 1 per cycle). While these results do not include timing, they do provide encouragement that multicast snooping can obtain data directly (like broadcast snooping) but apply to larger systems (like directories).*

## 1 Introduction

Large applications, such as simulators and database servers, require cost-effective computation power beyond that of a single microprocessor. Shared-memory multiprocessor servers have emerged as a popular solution, because the system appears like a multi-tasking uniprocessor to many applications. Most shared memory multiprocessors use per-processor cache hierarchies that are kept transparent with a coherence algorithm.

The two classic classes of coherence algorithms are snooping and directories. *Snooping* [14] keeps caches coherent using a totally ordered network to broadcast coherence transactions directly to all processors and memory. Modern implementations of snooping have moved well beyond the initial concept. The Sun Ultra Enterprise 10000 [1], for example, uses four address "buses" interleaved by address. It implements each "bus" with a pipelined broadcast tree constructed from point-to-point links (that behave more like ideal transmission lines to facilitate having multiple bits concurrently in flight), and it has a separate unordered data network (a point-to-point crossbar). Nevertheless, it implements Total Store Order (TSO), SPARC's variant of processor consistency, and it could implement sequential consistency.

In contrast, directory protocols [8, 22] transmit a coherence transaction over an arbitrary point-to-point network to a directory entry (usually at memory) which, in turn, redirects the transaction to a superset of processors caching the block. Due to the unordered network, care must be taken to ensure that concurrent transactions obtain data and update the directory in a manner that appears atomic, despite being implemented with a sequence of messages (e.g., acknowledgment messages from all processors involved). A state-of-the art example of a directory protocol that implements sequential consistency is that of the SGI Origin2000 [20].

Snooping protocols are successful because they obtain data quickly (without indirection) and avoid the overhead of sequencing invalidation and acknowledgment messages. They are limited to relatively small systems, however, because they must broadcast all transactions to all processors and all processors must handle all transactions. In contrast, directory protocols can scale to large systems, but they have higher unloaded latency because of the overheads of directory indirection and message sequencing. Snooping protocols have been more successful in the marketplace because many more small machines are needed than large ones.

In this paper, we investigate a hybrid protocol that obtains data directly (like snooping) when address bandwidth is sufficient, but scales to larger machines by dynamically degrading to directory-style indirection when it is not. We call our proposal *multicast snooping* because it multicasts coherence transactions to selected processors, lowering the address bandwidth required for snooping. With multicast snooping, coherence transactions leave a processor

This work is supported in part by the National Science Foundation with grants MIP-9225097, MIPS-9625558, CCR 9257241, and CDA-9623632, a Wisconsin Romnes Fellowship, and donations from Compaq Computer Corporation, Intel Corporation, and Sun Microsystems.

with a multicast "*mask*" that specifies which processors should snoop this transaction. Masks are generated using prediction and need not be correct (e.g., may fail to include the previous owner or all sharers). A multicast network logically determines a global order for all transactions and delivers transactions to each processor in that order, but not necessarily on the same cycle. Processors process these transactions as they would with broadcast snooping. A mask containing more processors than necessary completes as in broadcast snooping, but it wastes some address bandwidth. A simplified directory in memory checks the mask of each transaction, detecting masks that omit necessary processors and taking corrective action. Table 1 summarizes the differences between broadcast snooping, directories, and multicast snooping.

Section 2 introduces multicast snooping in more detail, delving into the thorny issues of mask prediction, transaction ordering, and forward progress. Section 3 discusses a multicast network sufficient to support multicast snooping. It is similar to the Isotach network proposed by Reynolds, Williams, and Wagner [33]. Sections 4 and 5 give methods and results from a preliminary analysis of sharing patterns, mask prediction, and network throughput. Results for mostly SPLASH-2 benchmarks on 32 processors show that: (1) the mean number of sharers encountered by a coherence transaction is less than 2 (so multicasts could go to far fewer than all 32 processors), (2) a plausible mask predictor can usually include all necessary processors (73-95%) and yet limit multicasts to an average of 2-6 destinations (<< 32), and (3) our initial network can deliver between 2-5 multicasts per network cycle (>> broadcast snooping's 1 per cycle). These results provide encouragement for developing multicast snooping, but they should be considered preliminary since they are not timing simulations and include some methodological approximations.

## 2 Multicast Snooping Coherence

Figure 1 shows the major components of a system that uses multicast snooping. We assume that addresses traverse a totally-ordered multicast address network, such as the one described in Section 3, that data travels on a separate point-to-point data network[1], as in the Sun E10000, and that memory is physically distributed among processors. We illustrate our ideas using a write-invalidate MOSI protocol.

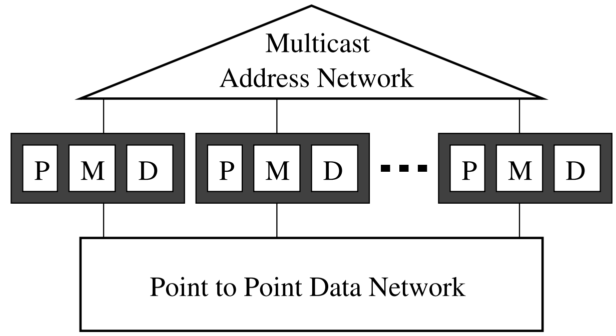


**FIGURE 1. Major components of a multicast snooping system.** P, M and D refer to processor, memory and associated directory.

### 2.1 Background: Snooping and Directories

Consider snooping and directory protocols that implement a write-invalidate MOSI protocol [38] which allows silent replacement of shared blocks. Processors can hold each cache block in one of four states: M (Modified), O (Owned shared), S (Shared), and I (Invalid). Memory has four stable states, corresponding to the states of the processors: M (memory invalid with one processor in state M and others I), O (memory invalid with one processor O and others S or I), S (memory valid with processors S or I), and I (block idle with all processors I). Memory state is implicit (not stored) for snooping and explicit with directories.

A processor must have a block in state M, O, or S to perform a load on it and in state M to perform a store. A processor in state I can use the transaction GETS (get shared) to obtain a block in state S. A processor in O, S, or I can use GETX (get exclusive) to obtain an M block. A proces-

1. The data network is logically separate, but it could be implemented with virtual channels on a unified network.

**TABLE 1. A Comparison of Coherence Methods**

| Coherence Method | Broadcast Snooping | Directories | New: Multicast Snooping |
|---|---|---|---|
| Find previous owner directly (without indirection through memory) | Yes | No | Usually, when prediction is correct (but memory state checked in parallel) |
| Always broadcast? | Yes | No | No (good) |
| Avoid serial invalidates & acks? | Yes | No | Yes (good) |
| Ordered network? | Yes | No | Yes (a challenge) |

sor in state S can silently downgrade a block to I. A processor in state M or O can use transaction PUTX (put exclusive) to request a writeback to transition to I.[1] Coherence protocols must resolve subtle races.

- Example 1: Consider a block B whose initial states at processors P0, P1, and P2 are M, I, and I, respectively. Let P1 and P2 issue GETXs to B at about the same time. With snooping, bus arbitration will order the GETXs (e.g., P1's GETX before P2's GETX), all processors will observe this order, and processors will act accordingly (i.e., P0 will send the block to P1 which will send it to P2). With directories, the serialization happens with an indirection through the directory entry for block B.

- Example 2: Consider a block B whose initial states at processors P0, P1, P2, and P3 are O, S, S, and I, respectively. Let P3 issue a GETX B. Snooping broadcasts P3's GETX to all processors so that P0 responds with B and invalidates while P1 and P2 invalidate. The total order of the bus also ensures no acknowledgment messages are needed to implement a memory consistency model. A directory protocol, in contrast, forwards P3's GETX to only P0, P1, and P2. P0 responds with B, while P1 and P2 respond with acknowledgment messages to indicate when the coherence transaction is complete, a necessity with an unordered network.

### 2.2 A Multicast Snooping Protocol

Multicast snooping operates like broadcast snooping with three major differences. First, coherence transactions are augmented with a predicted "mask" that specifies which processors should receive the transaction and always includes the requesting processor and the block's memory module. We will discuss mask prediction in Section 2.3.

Second, at memory and in parallel with other processors, a simplified directory entry verifies whether the mask is adequate. On receiving a transaction, memory takes action and transitions state immediately (i.e., logically before the next transaction). A GETS whose mask includes the previous owner succeeds. The previous owner, possibly memory, provides the data. A GETX that includes the previous owner and all sharers also succeeds; the previous owner, possibly memory, provides data. A GETX that includes the previous owner but not all sharers partially succeeds with the requesting processor making a transition to state O. When a transaction is not completely successful, memory provides the requesting processor with a negative or partial acknowledgment (nack or semiack, respectively) and a "better" mask. Most coherence transactions cause one response message (a data message or nack). Only GETS to an O block causes two response messages: one from memory and one from the previous owner. Multicast snooping's directory is simpler than a conventional directory because it sends at most one message per transaction and does not have to enter transient states for GETSs and GETXs.

Third, processor actions are somewhat more complex than with broadcast snooping. A processor transitions state immediately on seeing its own or another processor's transaction. Like broadcast snooping with a split-transaction bus, a processor issuing a GETX must buffer (or nack) all foreign transactions to that block until it receives the data and can respond. Multicast snooping is slightly more complex because in a few cases it may receive a nack-type message that nullifies the GETX, causing it to discard the buffered transactions. On receiving a nack, a processor will typically retry the transaction with the "better" mask provided by memory. Forward progress can be ensured by using a broadcast mask (that always succeeds) after *k* retries. Consider again the above examples.

- Example 1´: Consider a block B whose initial states at processors P0, P1, and P2 are M, I, and I, respectively. Let P1 issue GETX B mask=P1,P2 (not P0), which gets ordered before P2's GETX B mask=P0,P1,P2. Thus, P0 sees only P2's GETX, while P1 and P2 see P1's GETX before P2's. With straightforward snooping, P0 would send the block to P2 and P1 would wait forever. With multicast snooping, P1's GETX will be nacked by memory because it failed to include the previous owner P0. Thus, P0 correctly sends B to P2, since P1's GETX is invalidated. On learning from the nack, P1 can retry its transaction with a better mask.

- Example 2´: Consider a block B whose initial states at processors P0, P1, P2, and P3 are O, S, S, and I, respectively. Let P3 issue a GETX B with some mask. If the mask includes the previous owner and all sharers (e.g., mask=P0,P1,P2,P3), the transaction is successful and P3 goes to M. If the mask includes the previous owner but not all sharers (e.g., mask=P0,P1,P3), the transaction is partially successful and P3 goes to O. If the mask omits the previous owner (e.g., mask=P1,P3), the transaction fails and P3 stays I. P3 can then retry unsuccessful and partially successful transactions with a better mask.

Table 2 presents our protocol at a level of detail beyond what is necessary to read the rest of this paper, but it is included for completeness. This baseline protocol omits many optimizations, such as supporting an upgrade transaction to allow a processor to transition from S or O to M

1. To simplify exposition, we will not discuss PUTXs further.

| Requestor | | Memory | | | | | Other Processors in Mask | | | Requestor | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Trans-action | Old State | Old State | Owner in mask? | All in mask? | Send to requestor | New State | Old State | Send to requestor | New State | New State | Success? |
| GETS | I | S,I | yes | x | data_ack | S | | | | S | yes |
| | | M(q),O(q) | yes | x | | O(q) | M,O | data_ack | O | S | yes |
| | | | no | no | nack | same | | | | I | no |
| GETX | O | O(r) | yes | yes | ack | M(r) | S | | I | M | yes |
| | | | | no | semiack | O(r) | S | | I | O | partial |
| | S,I | S,I | yes | yes | data_ack | M(r) | S | | I | M | yes |
| | | S | yes | no | data_semiack | O(r) | S | | I | O | partial |
| | | M(q) | yes | yes | | M(r) | M | data_ack | I | M | yes |
| | | O(q) | yes | yes | ack | M(r) | O | data | I | M | yes |
| | | | | | | | S | | I | | |
| | | | | no | semiack | O(r) | O | data | I | O | partial |
| | | | | | | | S | | I | | |
| | | M(q),O(q) | no | no | nack | same | S | | I | same | no |
| PUTX | M | M(r) | yes | yes | | I | | | | I | yes |
| | O | O(r) | yes | x | | S | | | | I | yes |
| | I | I,S,M(q),O(q) | x | x | | same | | | | same | no |

**TABLE 2. A Baseline Protocol:** This table gives the baseline protocol in more detail. Columns 1 and 2 give the requesting processor's transaction and state when it sees its own transaction. Columns 3-5 give the states a transaction can encounter at memory, while Columns 6-7 give the memory's response. Memory states M and O are augmented with "(r)" if the requestor is (was) the owner and with "(q)" otherwise. An "x" denotes "don't care." Column 8 gives the state that other processors may be in when they see a transaction, while Columns 9-10 give their response. Cases where these processors do nothing are omitted for brevity (observing a GETS in S, observing a PUTX in I, and when omitted from a multicast mask). Finally, Columns 11-12 give the requesting processor's final state and whether the transaction was successful. All other cases are impossible.

without a data message. Another important optimization is to have memory retry unsuccessful transactions directly instead of nacking the requesting processor. This reduces latency in the case of mask misprediction, but it makes ensuring forward progress more complex.

So far, we have argued that multicast snooping can implement coherence. The end of a coherence protocol, however, is to help implement a memory consistency model, such as sequential consistency. Future work involves showing that multicast snooping can implement sequential consistency (or a weaker model) using an extension of Lamport's logical clocks [18] developed at Wisconsin [35, 30, 10].

## 2.3 Mask Prediction and Encoding

A key new challenge for multicast snooping is transaction mask prediction. Masks with too many nodes waste address bandwidth, while masks with too few nodes cause re-tries that add latency when obtaining blocks. Masks can be predicted with information from recent misses to the same block, recent misses to any block, behavior of spatially adjacent blocks, recent misses of the same static load or store instruction, input from software (programmer, compiler, library, or runtime system), or some combination of these. There are several important cases where mask predictors can send a transaction to the minimum number of destinations: memory and the requesting processor. These cases include GETSs for instruction fetches, read-only data and read-mostly data, and GETXs to private data (e.g., stack) or de facto private data (e.g., shared heap with accesses by only one processor). With many multicasts going to few destinations, accurate mask prediction for truly shared data becomes less critical.

We have developed an initial mask predictor, called *Sticky-Spatial(k)*, which performs reasonably well for our benchmarks. Each processor maintains a special direct-mapped table to cache mask prediction information. Each entry is tagged with a block address and contains "sticky"-mask and last-invalidator fields. A GETX transaction for block B predicts a multicast mask which includes the requestor, the directory, and the logical OR of masks from table entries B´-k, B´-(k-1),..., B´, B´+1,..., B´+k (regardless of tags), where B´ is the block address of B modulo table size (hence, "spatial," since we are combining information from *k* spatial neighbors on either side of block B). GETX data

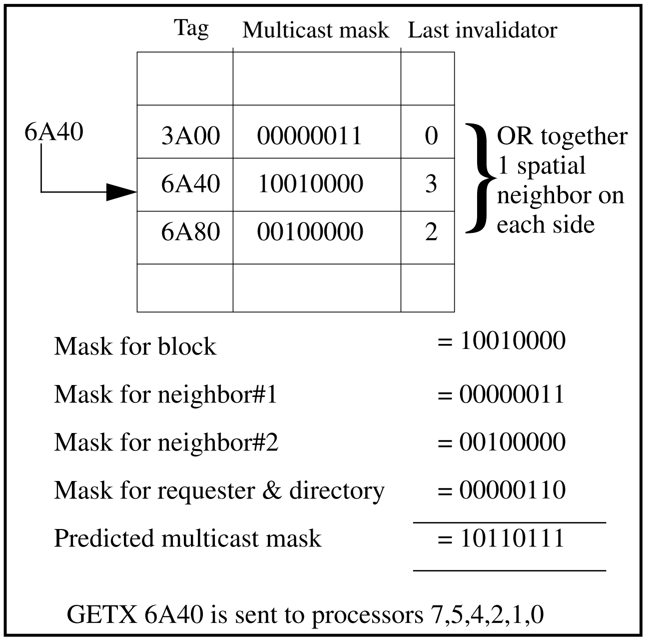


**FIGURE 2. The *StickySpatial(1)* predictor in action.** A GETX for the block with address 6A40 causes the predictor to access its table to find the multicast mask. The predictor also looks up one neighboring entry on each side and ORs these masks, as well as a mask which includes the requester and the directory, to get the final multicast mask. Note that since tags are not checked during prediction, it is possible to combine masks for unrelated blocks (e.g., one of the neighboring entries above corresponds to the unrelated block at address 3A00).

eventually returns with a mask from the previous owner or memory. This mask is logically ORed into entry B´ if the tag of B´ is the block address of B (hence, "sticky" since this will have the effect of recording all processors which have ever had a copy of block B); otherwise, the entry's tag and mask are set to the block address of B and the incoming mask respectively. When a processor is asked to invalidate block B, it sets its last-invalidator field to the requesting processor (regardless of tags). A GETS B transaction predicts a multicast mask which includes the requestor, directory, and last invalidator from entry B´. When the GETS data returns, entry B´ is updated as for a GETX. Figure 2 shows an example of the *StickySpatial(1)* predictor in action.

Efficiently encoding the multicast mask is also an important implementation issue. For this paper we simply assume a full-map directory entry, similar to most directory protocol studies [23]. However, many of the techniques developed for limited directory protocols [16, 27] can be adapted to multicast snooping.

## 3 Multicast Address Networks

A key technology for multicast snooping is a multicast address network.[1] A sufficient condition is that it creates the illusion of a total order of reliable multicasts. That is, multicasts can be *conceptually* numbered in such a way that each destination receives multicasts in strictly increasing order. It is *not* a requirement that a given multicast be delivered to all of its destinations simultaneously. A pipelined broadcast tree, like that used for broadcast snooping in the Sun E10000, meets the above correctness requirements, but falls short of our performance goals. We describe a more suitable network that offers optimization opportunities to our multicast protocol.

### 3.1 Goals and Isotach

Multicast snooping is more effective than broadcast snooping only when the network can combine multicasts and deliver multiple multicasts per network cycle. An ideal multicast network would boast:

- latency and cost as low as for a pipelined broadcast tree
- near-optimal throughput for delivering multicasts
- absence of centralized bottlenecks (e.g., no single "root")
- locality exploitation (e.g., if a multicast's destinations are within a sub-tree, the coherence traffic would not have to leave the sub-tree to be ordered)

Fortunately, Reynolds, Williams, and Wagner [33] have developed a class of networks, called *Isotach* networks, that have more stringent requirements than we have. An Isotach network allows a processor to send a set of heterogeneous variable-sized messages to multiple destinations, and it requires that they arrive at a specific logical time. Our requirements are less stringent than Isotach's in two ways. First, instead of a "set of messages," we have a single fixed-size coherence transaction sending identical information to all destinations. Second, while our processors require that a multicast arrive at all destinations at the same logical time, we do not let the processors specify what that logical time is. Furthermore, we allow coherence transactions to be re-ordered before they are inserted in the total order of multicasts.

### 3.2 An Isotach-like Fat Tree Network

We have developed an indirect multicast address network that meets our requirements and has potential to approach our ideal goals. It is a fat-tree with arbitrary uplinks and Isotach-like down links. It is not an Isotach network

1. Recall that the data responses to address transactions are delivered on a logically separate point-to-point data network.

because it does not meet Isotach's more stringent requirements.

The topology is a k-ary fat-tree network with N roots and P processors at the leaves, illustrated in Figure 3. A coherence transaction to block B travels up the fat-tree to root *r* selected by address (e.g., B mod r). On each network cycle *t* (which we treat as synchronous for simplicity), each root *j* selects a multicast and gives it the logical timestamp *t.j*. If no multicast is available, or if contention in the network prohibits the selected multicast from issuing, null messages are placed on all empty outgoing links in that cycle. Multicasts which issue from a root in the same network cycle *t* are said to belong to pulse *t*. Timestamps are implicitly carried with each multicast message, but can be transmitted using a small $\Delta t$ field (as little as one extra bit) in each message. Null messages add no contention and take up no space in network queues: they are sent on otherwise idle links to ensure ordered delivery by pulse, and their only effect is to update the local time (pulse) of the queue. All real messages also update the local time of the queue.

Interior fat-tree nodes pass on messages from older pulses before messages from more recent pulses. If any incoming network queue is empty, the local pulse of that queue arbitrates with the available multicasts in other queues. On cycles in which network contention prohibits the oldest pulse's multicasts from continuing, null messages are sent with the oldest pulse's pulse identifier. This ensures that the timestamps of a series of messages traversing each fat-tree link have non-decreasing pulse number. Since processors are at the end of a link, they receive multicasts in pulse order, but not necessarily in root order. Multicast snoop order is defined lexicographically as pulse order, then root order. Each destination must therefore sort received messages in each pulse into "by root" order before processing.

Many network issues are yet to be explored. Network implementations should allow asynchrony and must still be

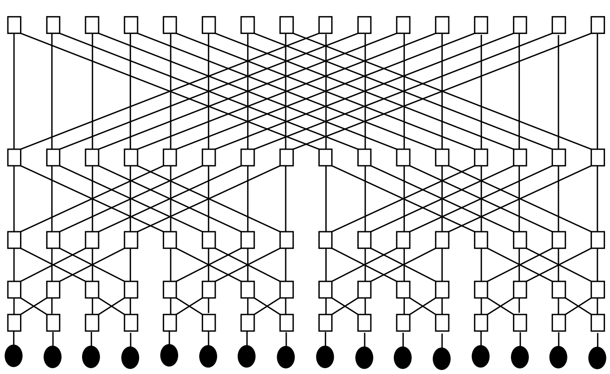

**FIGURE 3. Sample Network Design: 2-ary fat tree with 16 processors and 16 roots.**

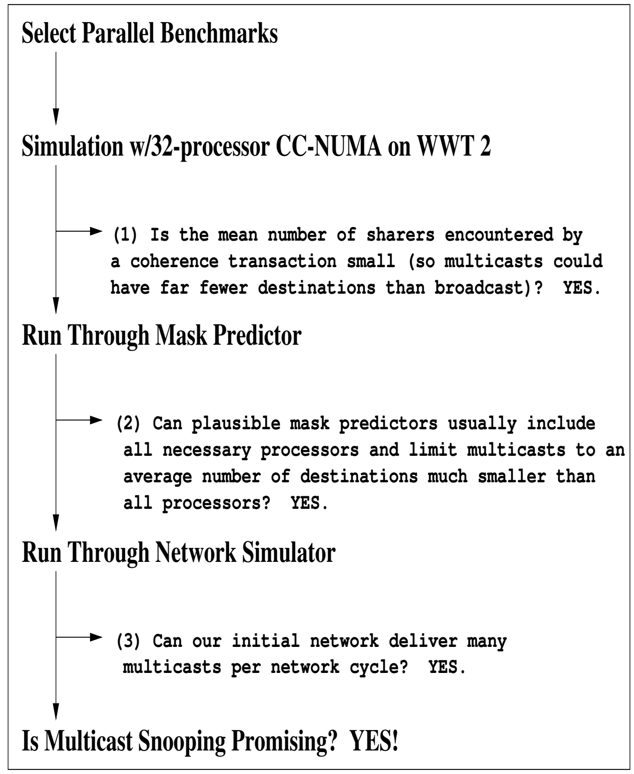


**FIGURE 4. Performance Evaluation Methodology Flowchart**

specified to a level that addresses switching technique, deadlock avoidance, etc. Fault tolerance should be supported to, at least, allow failed switches and links to be avoided after a reboot, as was done for the Thinking Machines CM-5 fat-tree networks [21].

Many other network improvements are yet to be explored. How can locality be exploited so that, for example, a multicast to destinations covered by a sub-tree of the network can be ordered at the sub-tree's root rather than at a network root? In the limit, a multicast from a processor to itself and a co-located shared-memory module should not need to traverse the network at all. Instead, the multicast could simply be inserted in a possibly-shared incoming queue. Finally, what are the consequences of implementing multicast snooping with other indirect and direct topologies, such as a two-dimensional torus?

## 4 Performance Evaluation Methods

This section describes the methods we have used to gather some preliminary evidence supporting multicast snooping. Figure 4 is a flowchart that illustrates the questions we ask in this section.

**Simulation of a 32-processor CC-NUMA system:** In the first part of our evaluation, we ran the benchmarks described in Table 3 on a CC-NUMA simulator to generate traces of coherence transactions in order to (a) answer questions about the mean number of sharers, and (b) have information to feed to mask predictors and network simulators. We used the Wisconsin Wind Tunnel II [28], a parallel, discrete-event, direct-execution simulator of multiprocessor shared-memory machines. The target architecture has 32 processors, and its parameters are shown in Table 4. The parallel benchmarks were written to use an explicitly allocated section of shared memory. Ownership of the pages of this shared memory was distributed in a round-robin manner among the nodes of the simulated machine. We modified the simulator to generate per-directory traces of coherence transaction requests received during the parallel phase of a benchmark (i.e., after initialization).[1] The traces include no relevant timing information, since the CC-NUMA protocol is different from the multicast snooping protocols we wish to study. These traces are an approximation of our baseline MOSI protocol, because they do not include the O state and do include an upgrade transaction. Furthermore, since WWT2 does not model instruction fetches, results are biased against multicast snooping due to omitting the predictable case of sending instruction miss GETSs to memory only.

**Mask Predictor:** In the second step of our methodology, we fed the generated traces into a mask predictor to (a) see if a plausible mask predictor can usually include all necessary processors and limit multicasts to an average number of destinations much smaller than all processors, and (b) generate predicted multicast traces to feed into our multicast network simulator. We predicted with *Sticky-Spatial(1)*, described in Section 2.3, using a 4K-entry table per processor. We assumed a full-map encoding of the directory and masks.

**Multicast Network Simulator:** In the third step of our methodology, we fed the predicted multicast traces into a network simulator that exactly models the abstract network described in Section 3. Results were computed by simulating a binary fat tree with 32 roots, 32 processors, and single element buffers at each link.

1. These include accesses to per-processor private blocks (private data segment and stack). They do NOT include accesses to blocks of shared memory that do not cause a remote coherence transaction request. Including these would improve the relative performance of multicast snooping.

**TABLE 4. WWT II Simulation parameters**

| Parameter | Value |
|---|---|
| # of processors | 32 |
| Type of system | CC-NUMA |
| Coherence mechanism | Directory protocol: full-map, write-invalidate, 3-state MSI |
| Data memory hierarchy | L1 cache, SPARC MBus, Local memory, Remote Block cache |
| L1 data cache | 128KB, direct-mapped, 32-byte blocks, write-back |
| Remote block cache | 512KB, direct-mapped, 32-byte blocks, writeback inclusion with L1 cache for read-write blocks |
| Local memory | 96MB |

| Benchmark | Description of Application | Input Data Set |
|---|---|---|
| *cholesky* | Blocked sparse matrix Cholesky factorization | tk16.O from SPLASH-2 |
| *fft* | Complex 1-D radix-$\sqrt{n}$ 6-step FFT | 64K points |
| *lu* | Blocked dense matrix LU factorization | 512x512 matrices, 16x16 blocks |
| *moldyn* | Simulation of molecular dynamics | 2048 particles, 15 iterations |
| *ocean* | Simulates large-scale ocean movements | 130x130 ocean |
| *radix* | Integer radix sort | 1M integers, radix 1024 |
| *raytrace* | 3-D scene rendering using raytracing | teapot from SPLASH-2 |
| *water-nq* | Quadratic-time simulation of water molecules | 512 molecules |

**TABLE 3. Benchmarks.** Our parallel benchmarks were taken mainly from the SPLASH-2 [43] benchmark suite, with the exception of Moldyn [29] which is a shared-memory implementation of a CHARMM-like [7] molecular dynamics application.

| Benchmark | Avg nodes in multicast | Avg extra nodes predicted | Multicast traffic ratio | Prediction accuracy (%) | Blocks found at home (%) |
|---|---|---|---|---|---|
| cholesky | 3.4 | 1.2 | 1.7 | 94 | 92 |
| fft | 3.2 | 0.3 | 1.4 | 73 | 57 |
| lu | 2.4 | 0.3 | 1.2 | 95 | 93 |
| moldyn | 5.4 | 2.9 | 2.4 | 88 | 56 |
| ocean | 3.4 | 0.8 | 1.3 | 95 | 45 |
| radix | 3.0 | 0.5 | 1.4 | 84 | 80 |
| raytrace | 5.6 | 3.4 | 2.9 | 86 | 75 |
| water-nq | 3.8 | 1.5 | 1.9 | 88 | 85 |

**TABLE 5. Multicast mask prediction statistics (to 2 significant digits).**

## 5 Performance Evaluation Results

**Sharing patterns.** Multicast snooping will work best if the mean number of sharers encountered by a coherence transaction is small (so multicasts can go to far fewer than all processors). Results in Figure 5 confirm results from the published literature [15, 2] that show this is the case (for these benchmarks)[1]. In particular at most 1.3% of transactions required more than two invalidations (for ocean).

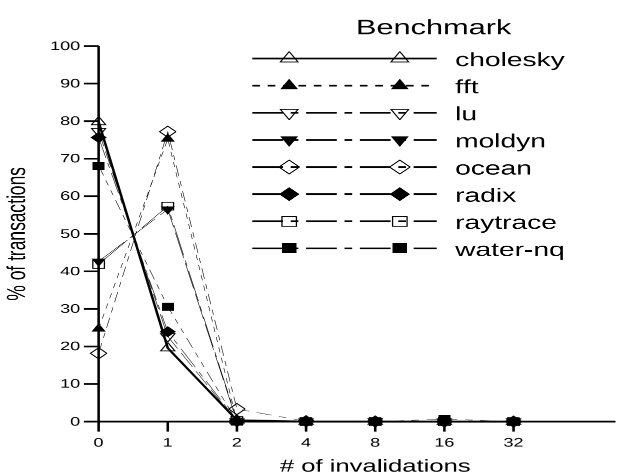


**FIGURE 5. Number of invalidations sent after a GETX/Upgrade or a GETS for an Exclusive**

**Multicast Mask Prediction.** Effective multicast snooping requires implementable mask predictors that usually include all necessary processors and limit multicasts to an average number of destinations much smaller than all processors. Table 5 presents results for a viable predictor: *Sticky-Spatial*(1) with 4K-entry table size (results for a 1K-entry table are similar). Columns 2-4 address how much extra traffic is generated. Column 2 is the average number of nodes per successful multicast[2]; column 3 is the average number of nodes in a predicted multicast that would not have been included in a perfect multicast; column 4 is the ratio of the total number of nodes included in all predicted multicasts (including retries) to the total number of nodes included in all perfect multicasts. Results show that a practical predictor can limit multicasts to 2-6 processors (far fewer than broadcast snooping's 32 processors) and generate traffic within a factor of three of optimal. Thus, *Sticky-Spatial*(1) is a reasonable predictor, but there is room for more improvement.

Table 5's columns 5 and 6 compare multicast snooping to directories. For example, 73% of masks predicted for all the coherence transactions of the fft application include all necessary processors (and possibly more), while 57% of all transactions found the block at the directory. The difference between the two columns indicates the percentage of trans-

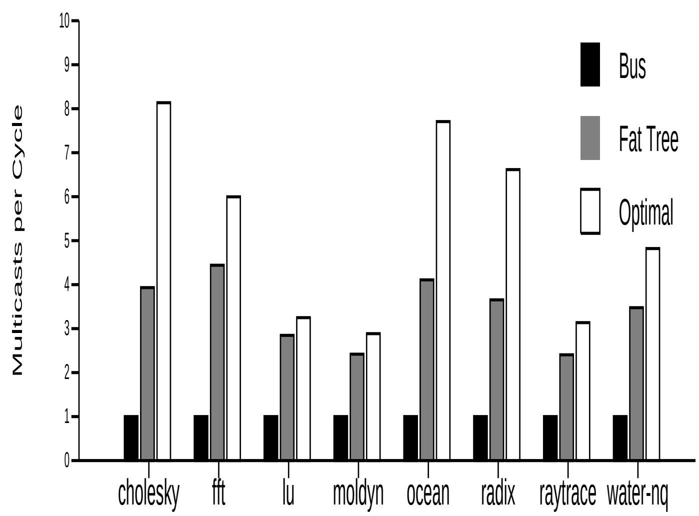


**FIGURE 6. Multicast Address Network Throughput**

1. Our results are not exactly the same as others' due to specific system assumptions (e.g., cache size, associativity, and coherence protocol).

2. If a multicast to *i* destinations fails and is re-issued to *j* destinations, this counts as *i+j* destinations for one successful multicast.

actions for which directories will have to contact a third (owner) node and incur the latency of an extra network hop. As shown, all the entries in Column 5 are larger than the corresponding entries in Column 6. Thus, multicast snooping will find blocks directly (in two hops) more often than directories will (for our benchmarks). This difference is small for the SPLASH-2 kernels, but it is significant for two applications, moldyn and ocean.

**Multicast Network Throughput.** Given a trace of the multicasts from the above section, we now ask whether our multicast address network can deliver much more than one broadcast per network cycle. Figure 6 shows results for a bus (black and always one), our network (gray), and an optimal network (white).[1] Our network generally achieves at least half of optimal throughput. Future work will show how loaded network latencies compare.

## 6 Related Work

Multicast snooping is a hybrid between broadcast snooping [14] and directory coherence [22, 8]. There have been other hybrid systems which have used snooping buses as part of larger, more scalable designs than simple broadcast snooping. One example is the Sequent Sting [25], in which each node is a snooping SMP connected by an SCI directory system. Encore Gigamax [42] uses a hierarchy of buses, where transactions move up the directory as high as needed to maintain coherence. Corollary Profusion[40] links two SMP buses with logic that defers a transaction that must first execute on the other bus. The Data Diffusion Machine [17] and the KSR-1 [12] are both hybrids in that they use hierarchies of snooping buses/rings to implement a COMA protocol that is neither snooping nor a directory. Scott and Goodman [34] add pruning caches to switches in a multistage interconnection network (MIN) to reduce broadcast invalidations. Multicast snooping differs from all these schemes in that (a) masks are predicted and need not be correct, (b) the multicast set is determined by the issuing processor and does not use state distributed throughout the network, and (c) the directory entry is only to verify the prediction.

Stenstrom [36] proposed a write-update coherence protocol that uses multicasts in a multistage interconnection network and maintains sharing information at the owner's cache (the memory directory only maintains a pointer to the owner). In contrast, multicast snooping is a write-invalidate protocol and allows imperfect masks.

1. Consider a trace of $M_{total}$ multicasts, where $M_{max}$ is number of multicasts to the destination that received the most. For a network that can deliver to each destination at most one multicast per cycle, the optimal throughput is $M_{total} / M_{max}$.

Multicasting has been used to support communication constructs in numerous programming environments, including the ISIS [6] and Orca [4] projects. ISIS uses a software scheme to support multicast communication to a process group. The Orca distributed shared memory system is built upon an underlying software multicast mechanism that is part of the Panda virtual machine. ISIS and Panda both provide reliable multicasting and, as with our multicast network, they both ensure that all nodes see communications in the same order. Multicast snooping differs from these projects in that it relies on hardware to efficiently perform reliable multicasting.

Hardware multicast has been studied for both direct [26] and indirect networks [39]. Research has included switch design [37], flow control [5] and deadlock avoidance [24]. Multicast has been proposed for efficient support of synchronization variables [3]. Isotach networks provide totally ordered multicasts and groups of operations which are atomic in logical time [33]. Isotach networks were originally proposed to allow pipelined implementations of sequential consistency without caches and powerful synchronization without locks.

The implementation of our multicast address network uses techniques similar to those used in distributed simulation. In particular, null messages are used to place a lower bound on timestamps of future messages sent on a network link, and this is similar to the use of null messages in conservative parallel discrete-event simulation [9,13] and ghost messages in PRAM emulation [31].

Cache coherence protocols have also been designed to exploit ordering properties of interconnection networks. Landin et al. showed that a class of race-free networks eliminates the need to send acknowledgment messages in a directory protocol [19]. The Delta Cache protocols [41, 11, 32] exploit the strong ordering properties of Isotach networks to provide sequential consistency as well as powerful synchronization operations. However, unlike multicast snooping, Delta Cache protocols require the processor to check timestamps before completing each L1 cache access.

## 7 Conclusions and Future Work

This paper proposes a new coherence method called *multicast snooping*, which behaves like snooping for small systems and gracefully and dynamically degrades to directory-like indirection for large systems. Multicast snooping is unique because processors predict which caches should snoop each coherence transaction by specifying a multicast "*mask*." Transactions are delivered with an ordered multicast network that eliminates the need for acknowledgment messages. Processors handle transactions as they would with snooping, while a simplified directory operates in parallel to check masks and gracefully handle incorrect ones (e.g., previous owner missing).

The results are preliminary, because they include some methodological approximations, do not simulate timing, and are limited to one system size and small benchmarks. Nevertheless, they provide encouragement that multicast snooping can support larger systems than conventional snooping. If the limit is the number of incoming transactions a processor can process, then multicast snooping systems can be 2-5 times larger. Compared to directory systems, multicast snooping appears promising because it more frequently finds data directly (by sometimes avoiding indirection for data that is not at home) and eliminates the need to generate, sequence, and wait for explicit acknowledgment messages.

Future work will involve developing a timing simulator to allow us to study multicast snooping systems in greater detail. We would also like to examine the performance of other applications, such as databases, on this simulator. In addition, there are several other issues that are outside the scope of this particular paper. It would be interesting to examine other configurations, such as clusters of SMPs, and other network architectures, including direct networks. Studying other network possibilities also opens up the possibility of relaxing some of our ordering and synchronization requirements. Finally, fault tolerance should be supported to, at least, allow failed switches and links to be avoided after a reboot.

## 8 Acknowledgments

We would like to thank the following people for their comments on this work and/or paper: Pei Cao, Anne Condon, Robert Cypher, Paul Close, Jose Duato, Charles Fischer, Erik Hagersten, Joel Emer, Anders Landin, Milo Martin, Shubu Mukherjee, Gil Neiger, Paul Reynolds, Yannis Schoinas, Jeff Thomas, and Craig Williams.

## 9 References

[1] The Ultra Enterprise 10000 Server. http://www.sun.com/servers/datacenter/whitepapers/E10000.ps.

[2] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 280–289, Honolulu, Hawaii, May 30–June 2, 1988.

[3] John B. Andrews, Carl J. Beckmann, and David K. Poulsen. Notification and Multicast Networks for Synchronization and Coherence. *Journal of Parallel and Distributed Computing*, 15(8):332–350, February 1992.

[4] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. Orca: A language for Parallel Programming of Distributed Systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[5] Raoul A. F. Bhoedjang, Tim Ruhl, and Henri E. Bal. Efficient Multicast On Myrinet Using Link-Level Flow Control. In *Proc. International Conference on Parallel Processing*, pages 381–390, August 1998.

[6] Kenneth P. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12):37–53, December 1993.

[7] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D. J. States, S. Swaminathan, and M. Karplus. CHARMM: A Program For Macromolecular Energy, Minimization,and Dynamics Calculation. *Journal of Computational Chemistry*, 4(187), 1983.

[8] L. M. Censier and P. Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, 27(12):1112–1118, December 1978.

[9] K. M. Chandy and Jayadev Misra. "Distributed simulation: A case study in design and verification of distributed programs ". *IEEE Transactions on Software Engineering*, SE-5(5):440–452, September 1979.

[10] Anne E. Condon, Mark D. Hill, Manoj Plakal, and Daniel J. Sorin. Using Lamport Clocks to Reason About Relaxed Memory Models. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, Orlando, Florida, January 1999.

[11] Bronis R. de Supinski. *Logical Time Coherence Maintenance*. PhD thesis, University of Virginia, May 1998.

[12] S. Frank, H. Burkhardt III, and J. Rothnie. The KSR1: Bridging the Gap Between Shared Memory and MPPs. In *Proc. COMPCON 1993*, pages 285–295, Spring 1993.

[13] Richard M. Fujimoto. Parallel Discrete Event Simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

[14] James Goodman. Using Cache Memories to Reduce Processor-Memory Traffic. In *Proceedings of the International Symposium on Computer Architecture*, Trondheim, Norway, June 1983.

[15] Anoop Gupta and Wolf-Dietrich Weber. Cache Invalidation Patterns in Shared-Memory Multiprocessors. *IEEE Transactions on Computers*, 41(7):794–810, July 1992.

[16] Anoop Gupta, Wolf-Dietrich Weber, and Todd Mowry. Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes. In *Proceedings of the 1990 International Conference on Parallel Processing (Vol. I Architecture)*, pages 312–321, 1990.

[17] Erik Hagersten, Anders Landin, and Seif Haridi. DDM–A Cache-Only Memory Architecture. *IEEE Computer*, 25(9):44–54, September 1992.

[18] Leslie Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[19] Anders Landin, Erik Hagersten, and Seif Haridi. Race-Free Interconnection Networks and Multiprocessor Consistency. In *Proceedings of the International Symposium on Computer Architecture*, June 1991.

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

[21] Charles E. Leiserson et al. The Network Architecture of the Connection Machine CM-5. In *Proceedings of the Fifth ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, July 1993.

[22] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, June 1990.

[23] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[24] Xiaola Lin and Lionel M. Ni. Deadlock-Free Multicast Wormhole Routing in Multicomputer Networks. In *Proc. 18th Int'l Symp. on Computer Architecture*, pages 116–125, May 1991.

[25] T. Lovett and R. Clapp. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. In *Proc. 23rd Int'l Symp. on Computer Architecture*, pages 308–317, May 1996.

[26] Prasant Mohapatra and Vara Varavithya. A Hardware Multicast Routing Algorithm for Two-Dimensional Meshes. In *Proc. Eighth IEEE Symposium on Parallel and Distributed Processing*, pages 198–205, October 1996.

[27] Shubhendu S. Mukherjee and Mark D. Hill. An Evaluation of Directory Protocols for Medium-Scale Shared-Memory Multiprocessors. In *Proceedings of the 1994 International Conference on Supercomputing*, pages 64–74, Manchester, England, July 1994.

[28] Shubhendu S. Mukherjee, Steven K. Reinhardt, Babak Falsafi, Mike Litzkow, Steve Huss-Lederman, Mark D. Hill, James R. Larus, and David A. Wood. Wisconsin Wind Tunnel II: A Fast and Portable Parallel Architecture Simulator. In *Workshop on Performance Analysis and Its Impact on Design (PAID)*, June 1997.

[29] Shubhendu S. Mukherjee, Shamik D. Sharma, Mark D. Hill, James R. Larus, Anne Rogers, and Joel Saltz. Efficient Support for Irregular Applications on Distributed-Memory Machines. In *Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 68–79, July 1995.

[30] Manoj Plakal, Daniel J. Sorin, Anne E. Condon, and Mark D. Hill. Lamport Clocks: Verifying a Directory Cache-Coherence Protocol. In *Proceedings of the 10th Annual ACM Symposium on Parallel Architectures and Algorithms*, pages 67–76, June 1998.

[31] Abhiram G. Ranade. How to Emulate Shared Memory. *Journal of Computer and System Sciences*, 42(3):307–326, 1991.

[32] Paul Reynolds and Craig Williams. Personal communication, October 1998.

[33] Paul F. Reynolds, Jr., Craig Williams, and Raymond R. Wagner, Jr. Isotach Networks. *IEEE Transactions on Parallel and Distributed Systems*, 8(4):337–348, April 1997.

[34] Steven L. Scott and James R. Goodman. Performance of Pruning-Cache Directories for Large-Scale Multiprocessors. *IEEE Trans. on Parallel and Distributed Systems*, 4(5):520–534, May 1993.

[35] Daniel J. Sorin, Manoj Plakal, Mark D. Hill, and Anne E. Condon. Lamport Clocks: Reasoning About Shared-Memory Correctness. Technical Report CS-TR-1367, University of Wisconsin-Madison, March 1998.

[36] Per Stenstrom. A Cache Consistency Protocol for Multiprocessors with Multistage Networks. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 407–415, 1989.

[37] Craig B. Stunkel, Rajeev Sivaram, and Dhabaleswar K. Panda. Implementing Multidestination Worms in Switch-Based Parallel Systems: Architectural Alternatives and Their Impact. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.

[38] Paul Sweazey and Alan Jay Smith. A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 414–423, June 1986.

[39] Vara Varavithya and Prasant Mohapatra. Asynchronous Tree-Based Multicasting in Wormhole-Switched MINs. In *Proceedings of the 26th International Conference on Parallel Processing*, August 1997.

[40] George White and Pete Vogt. Profusion (tm): A Buffered, Cache Coherent Crossbar Switch. In *IEEE Hot Interconnects*, pages 87–96, August 1997.

[41] Craig Williams. Concurrency Control in Asynchronous Computations. Ph.d. thesis, University of Virginia, Computer Sciences Department, January 1993.

[42] A.W. Wilson, Jr. Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors. In *Proceedings of the 14th International Symposium on Computer Architecture*, pages 244–253, June 1987.

[43] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Shingh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, June 22–24, 1995.