Scaling SMP Machines Through Hierarchical Snooping

George Kola and Michael Marty

Computer Sciences Department
University of Wisconsin-Madison

CS 757 Final Project
5/14/2002

ABSTRACT

We examine an approach to scaling SMP nodes by using a hierarchical bus with transaction filtering to reduce bus traffic. An analysis of messages in a snoop-based cache coherence protocol reveals the types of transactions that can be filtered from the bus hierarchy. We implemented our filtering mechanism in Simics/Ruby, a full-system simulator, and show that bus traffic can be reduced when running commercial and scientific workloads. We also explain why memory locality is necessary to achieve effective filtering in a distributed memory system. Hierarchical snooping also introduces complications to achieving sequentially consistent and deadlock-free execution.

1 Introduction

Small-scale multiprocessor built using snooping-bus schemes are extremely cost effective and popular. In fact, many mass-produced microprocessors, such as chips based on the Intel Pentium Pro, offer built-in support for a cache coherent memory system. Thus small SMP machines can be built with little extra cost beyond that of the actual microprocessors and are often found in today's desktop machines.

However, due to the broadcast nature of a snooping protocol, SMP machines have scalability issues. To build larger shared-memory multiprocessor machines, computer architects have devised schemes using a directory-protocol [1]. Such machines are typically large and expensive as a custom interconnection network must be used to support the point-to-point messaging involved with a directory.

Hierarchical snooping may allow machines to be built leveraging the cost advantages of a simple interconnection bus and the price/performance advantages of mass-produced processing nodes. Such an approach may become more important with the advent of chip-multiprocessing [2]. Using a costly

interconnection network may not be an option when building a high-performance desktop machine using off-the-shelf CMP nodes.

In this paper, we examine hierarchical snooping in closer detail. In Section 2, we discuss hierarchical snooping in the most general sense. Coherence monitoring and filtering is dissected in Section 3. Our strategy for simulation is presented in Section 4. Section 5 presents our data and observations. Memory consistency and correctness issues are discussed in Section 6. Other related work is highlighted in Section 7, and finally we conclude in Section 8.

2 Hierarchical Snooping

A large multiprocessor can be constructed by logically connecting the buses of multiple SMP nodes in a 2-level hierarchy. Each leaf node, consisting of multiple processors on a common bus, is connected to a higher-level interconnection bus. The processor caches within a leaf node are kept coherent by a traditional snooping protocol. To maintain cache coherence across several nodes, any transaction that appears on the bus of a leaf node must also appear on all other buses. It is highly likely that bus saturation would occur due to the volume of transactions. It is also conceivable that a substantial portion of transactions appearing on the bus do not need to propagate to every processor in the system. Only those transactions that affect the cache coherence state of any processor cache in a node are necessary to appear on the bus of that node.

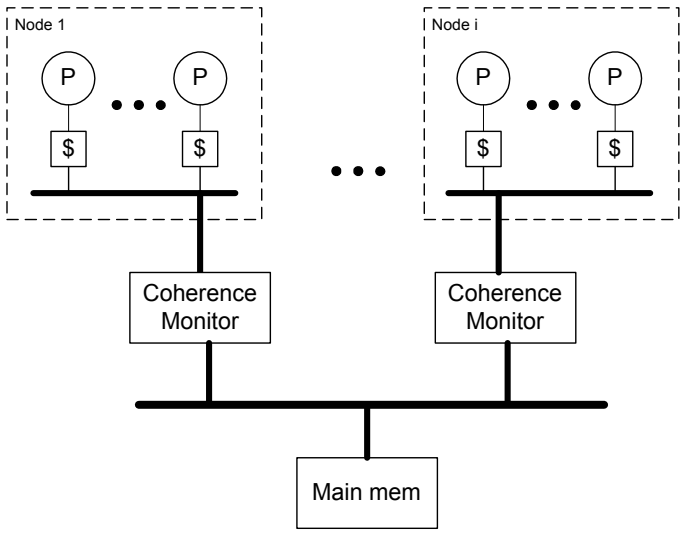

Figure 1- Hierarchy of bus-based nodes with coherence monitors

We hypothesize that a similar multiprocessor can be constructed with a mechanism to prevent bus saturation. Figure 1 shows the same hierarchy with a "coherence monitor" used to connect each node bus to the higher-level bus. The purpose of a coherence monitor is to forward only the necessary transactions in both directions thus reducing the bandwidth demands of all buses in the system. Each message appearing on both the node bus and the top-level bus is examined and forwarded based on the state accumulated by the coherence monitors. A coherence monitor is effectively a filter and both terms will be used interchangeably throughout this paper.

Figure 2 shows a similar hierarchy with distributed memory; each logical node has a local memory instead of a single centralized memory. Therefore, each leaf in the hierarchy is a complete multiprocessor and can conceptually be either an SMP or CMP node. We will focus on distributed memories because our target application is the construction of multiprocessor machines using commodity nodes. Distributed memory also offers the potential advantages of exploiting locality within a node. This is discussed further in section 5.3.2.

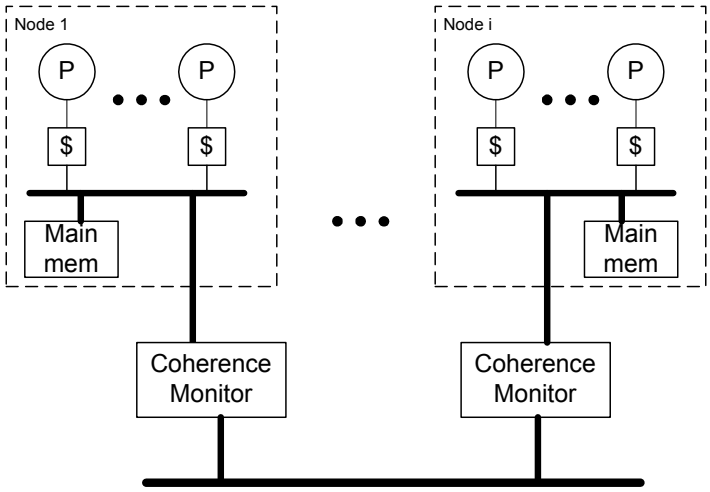

Figure 2- Hierarchy of bus-based nodes with memory distributed

3 Coherence Monitors

A coherence monitor must examine every message and determine whether or not to forward the message to the next level in the bus hierarchy. Incorrectly removing a message could lead to deadlock or incoherent caches, therefore, message filtering must be conservative. We now examine the types of messages that can be filtered in a typical snoop-based protocol.

For messages appearing on the bus of a leaf node, those that do not require data from remote memories and do not change the state of any remote cache can be filtered. Table 1 broadly classifies the types of messages that appear on a node bus for a typical 4-state MOSI invalidation protocol (this protocol will be assumed throughout the paper). Any read to a memory block mapped to some remote memory must be propagated to the next level in the bus hierarchy. Likewise, any writeback to a remote memory must also be forwarded. For a read transaction to a local memory block, the message can be filtered if there are no remotely modified/owned copies of that block as the local memory will have a fresh copy of the block. A read-exclusive transaction to a local block can be filtered if and only if there are no remote copies of that block.

Table 1- Types of transactions appearing on a local bus of a node.

Bus Transaction	**Memory Block Location**	**State of Any Remote Cache**	**Filterable**
read exclusive	Remote	any	no
read	Remote	any	no
read	Local	shared or uncached	yes
read exclusive	Local	uncached	yes
read exclusive	Local	shared, owned, or modified	no
write back	Local	any	yes
write back	Remote	any	no

To filter incoming messages from the top-level bus (those that originate from a remote node), the coherence monitor must consider the memory block in question and whether a local processor has that

block cached. Table 2 shows transactions that may appear on the bus and those that can be filtered. Any transaction reading and writing data to memory must be propagated to the node containing the location of the block. Therefore, any writebacks observed to an outside memory block can be filtered. For bus read and read-exclusive transactions for remote blocks, the message may be dropped if no processor within the local node has the block cached. A bus-read transaction to a remote block may also be dropped if no local processor has the block in owned or modified state.

Table 2 - Types of transactions appearing on the top-level bus of a node.

Bus Transaction	**Memory Block Location**	**State of Any Local Cache**	**Filterable**
read exclusive	local	any	no
read	local	any	no
read	remote	shared or uncached	yes
read exclusive	remote	uncached	yes
read exclusive	remote	shared, owned, or modified	no
write back	local	any	no
write back	Remote	any	yes

To filter messages as described, a coherence monitor must acquire state about local and remote caches. Because of the broadcast nature of a snooping protocol, all the necessary state is gathered by snooping the buses. On every message appearing on a bus, the coherence monitor must update its state and determine whether to propagate the message.

To filter messages originating for a local processor, the coherence monitor must maintain state for every local memory block cached by some remote processor. Unlike a directory protocol, the particular sharer does not need to be known as the messages are not routed in a point-to-point interconnect. Therefore, only two bits of state are needed for each memory block-- one to denote that a block may be remotely cached in the shared state, and one to denote that a block is remotely modified or owned. The shared bit is set if a bus-read message is observed on the top-level bus for the local memory block. Likewise, a bus-read-exclusive message will set the modify/owned bit. If a local processor issues a bus-

read-exclusive message for a local memory block with either of the bits set, they will be cleared after the message is forwarded. An observed writeback to a local memory block will also clear any bits that are set for that block. A problem does occur when a remote cache replaces a local block in the shared state. The coherence monitor will be unaware of the replacement and will assume that the block is still cached remotely. The only consequence of this scenario is the propagation of an unnecessary message which does not compromise correctness. We will reexamine this problem in section 5.3.3.

Filtering messages from the top-level bus presents a larger challenge as the coherence monitor must consider the status of the local caches which may have *any* block in the memory system cached. One solution is to maintain two bits of state for every block in the entire memory system. A bit is used to denote that a local processor has the block in the shared state with the other bit denoting a modified or owned block. Another approach is to maintain enough state only for the maximum number of blocks that can be cached by all the processors in that node. This approach results in a substantial reduction in state required because the amount a node can cache is far less than the total amount of memory in a machine. However an associative lookup would be required on each transaction appearing on the bus. We choose the former approach for simplicity reasons. To set the state bits, the coherence monitor examines each request from a local processor. The modify/owned bit is set on a bus-read-exclusive transaction and the share bit is set on a bus-read transaction. The bits for a block are cleared when a remote processor makes either a bus-read-exclusive request or a writeback to that block.

4 Simulation Overview

To evaluate the effectiveness of coherence monitoring and filtering, we used the Simics full-system multiprocessor simulator [3]. The Sun Microsystems' SPARC v9 platform architecture is simulated at the functional level. This allows us to evaluate our implementation using non-trivial benchmark applications running on an unmodified Solaris operating system. Simics is complemented with a custom add-on module called Ruby-- a detailed memory hierarchy simulator developed by the Wisconsin Multifacet group. Ruby provides a rich framework for evaluating a multitude of memory system designs including cache coherence protocols and interconnection networks. We extended Ruby's existing implementation of a 2-level broadcast snooping protocol to include coherence monitoring and filtering. A

16-processor configuration is partitioned into four nodes to mimic a hierarchy of bus-based nodes. Each processor has a memory associated with it. Memory is interleaved at the block-level. Figure 3 shows a high-level view of the configuration.

.

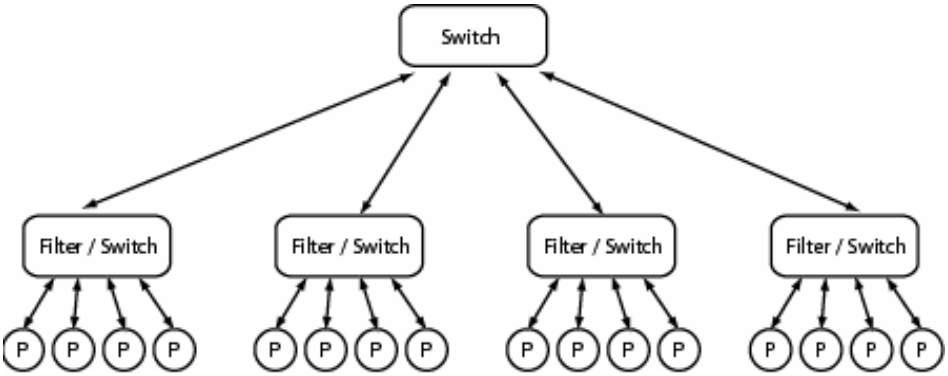

Figure 3- Ruby implements a 2-level hierarchy of switches for its MOSI broadcast protocol. We augmented the first-level switches with filter logic.

The following benchmark applications were used to evaluate our filtering mechanism:

1. On-Line Transaction Processing (OLTP): DB2 with a TPC-C like workload
2. Static Web Content Serving: Apache with SURGE
3. Java Server Workload: SPECjbb
4. Scientific Application: Barnes-Hut

Our simulation goal was to evaluate the effectiveness of filtering in terms of the amount of messages actually filtered. In other words, we did not set out to measure raw performance improvement with time-based metrics. We also treat each first-level switch as a "node bus" for the sake of discussion.

5 Results

5.1 Message Relevance

We first collected statistics about the relevance of messages appearing on the bus of each node with no filtering in place. A message was deemed relevant if the address of the message maps into the node's local address range, or if any cache in the node contains a copy of a remote block. We did not attempt to distinguish between bus-read and bus-read-exclusive transactions. Figure 4 shows the results for one node in the system. Results were similar for the other three nodes therefore we omit this redundant data. We refer to "Outside Messages" as those that originate from a processor not in the local node. An "Irrelevant Outside Message" is a message that requests a remote block not present in any of the local caches.

Figure 4 shows that a large percentage of messages appearing on the bus originate from remote processors. This observation correlates well with Ruby's memory interleaving strategy as 75% of all memory accesses would be remote in the case of a total distributed memory access pattern. The percentage of irrelevant messages requesting remote data is less than 40% for all four benchmarks. These are the only messages that can safely be filtered and represents the potential of traffic reduction.

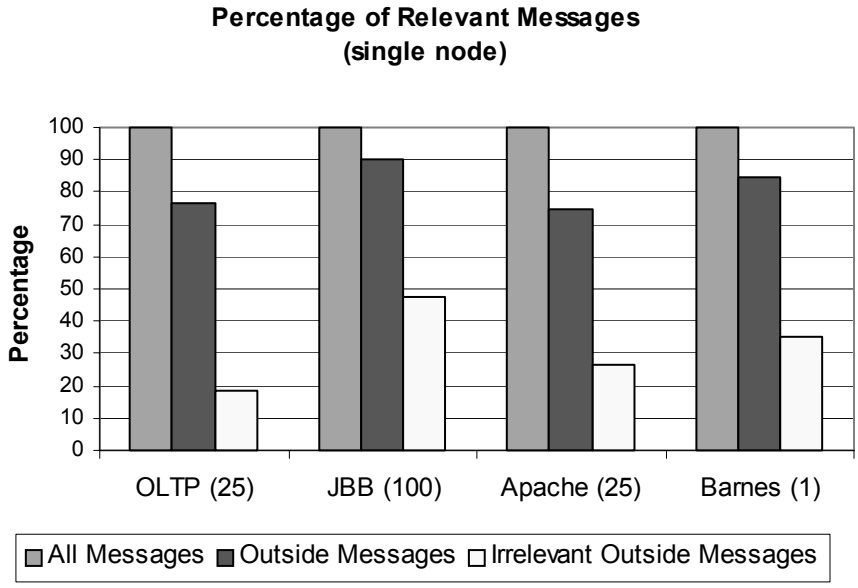

Figure 4- Breakdown of the messages appearing on the bus of a single node

5.2 Filtering

We then added the filter logic to Ruby's first-level switches. We first implemented logic to filter only outgoing messages based the state collected by snooping messages received from other nodes. By reducing the amount of outbound messages, the traffic present on the top-level switch is reduced. Next, incoming messages from the top-level switch were filtered based on the state collected by snooping messages as they are routed. Reducing the amount of messages propagating to the processors would lesson the traffic present on a local bus if a strict bus-based hierarchy were employed.

Figures 5-8 show the reduction in message traffic with filtering in place. The graphs represent the number of messages present in the first-level switch. Therefore, to match our logical descriptions of a bus hierarchy, we can represent the local bus traffic by the left-most set of bars. The reduction in bus traffic on the top-level bus can be inferred from the amount of outsides messages reduced by outgoing filtering as shown by the middle set of bars. It is worthwhile to emphasize that incoming filtering does not reduce top-level bus traffic.

For OLTP, the total traffic removed from the local bus is about 10.5% for outgoing filtering which also represents the reduction in messages seen on the top-level bus. For incoming and outgoing filtering, 20% of messages on the local bus are removed.

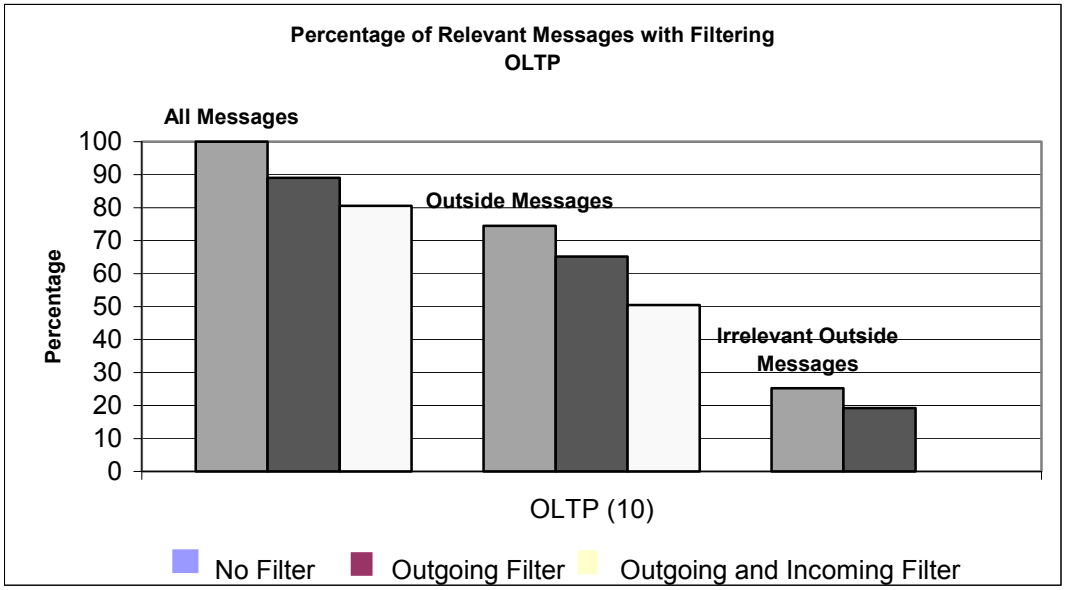

Figure 5

JBB benefits the most from filtering as shown in Figure 6. Filtering removes approximately 50% of the traffic from the local bus. However with only outgoing filtering enabled, traffic on the top-level bus is reduced by about 15%.

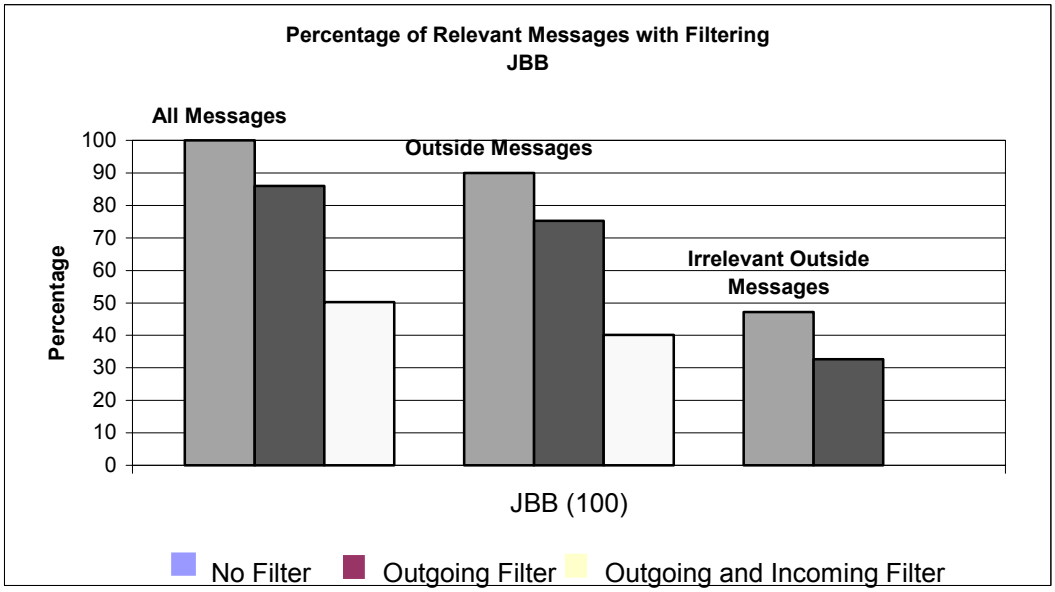

Figure 6

With Apache, we did not observe much reduction in top-level bus traffic, and local bus traffic was only reduced by full filtering.

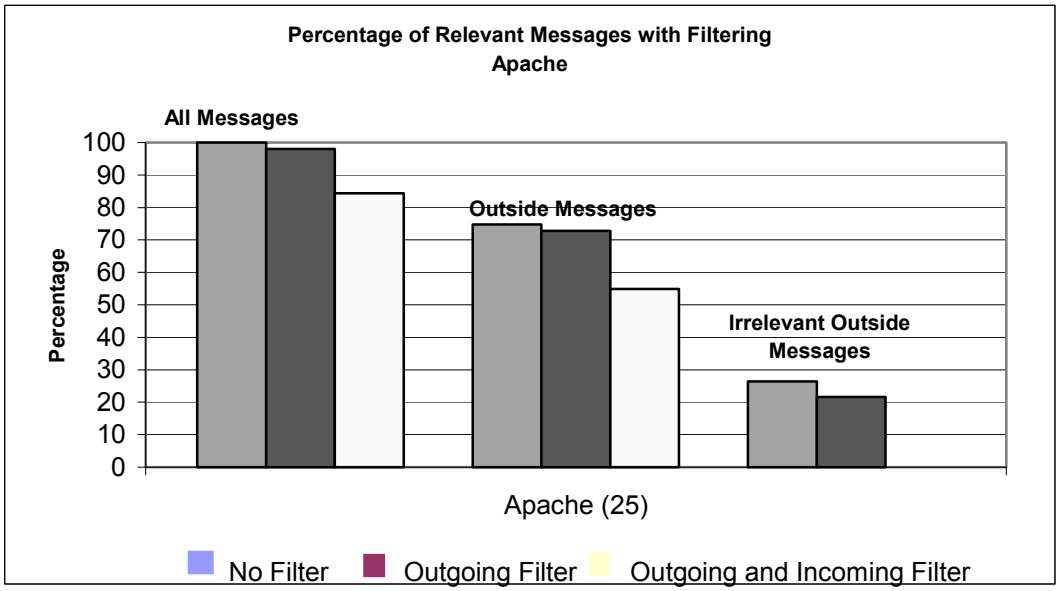

Figure 7

For Barnes-Hut, outgoing filtering reduces the number of messages on the top-level bus by approximately 12%. Incoming filtering results in a 48% reduction in traffic on the local bus.

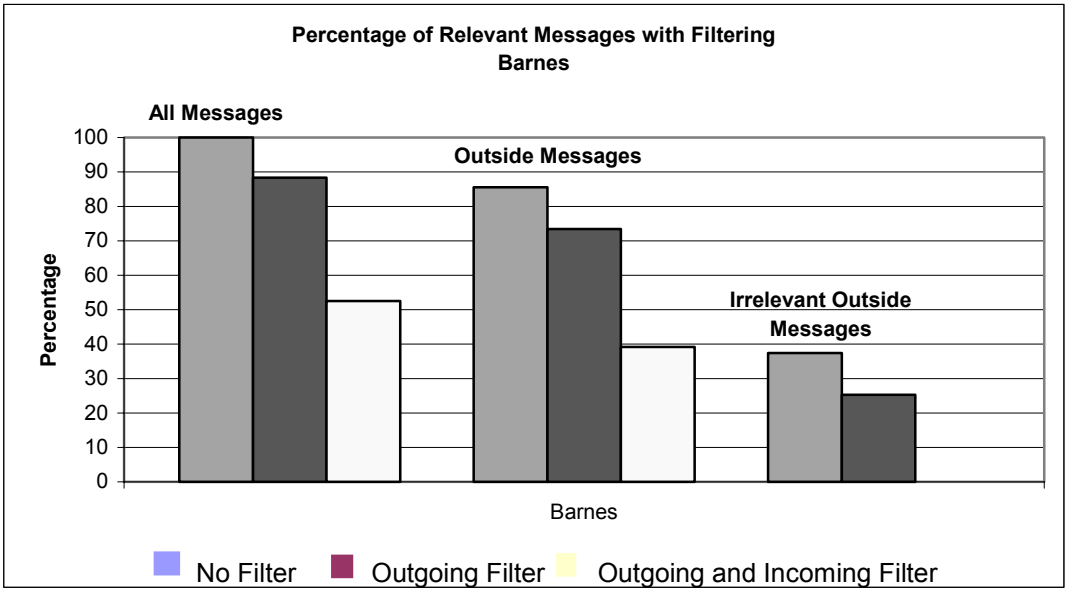

Figure 8

5.3 Other Observations

5.3.1 Types of Filtered Messages

The figures below show the reasons, corresponding to Tables 1 and 2, for filtering messages. For outgoing messages, we see that a substantial number of messages are filtered when there is a remote sharer. However for incoming messages, most filtering occurs due to no local copies of the block in question.

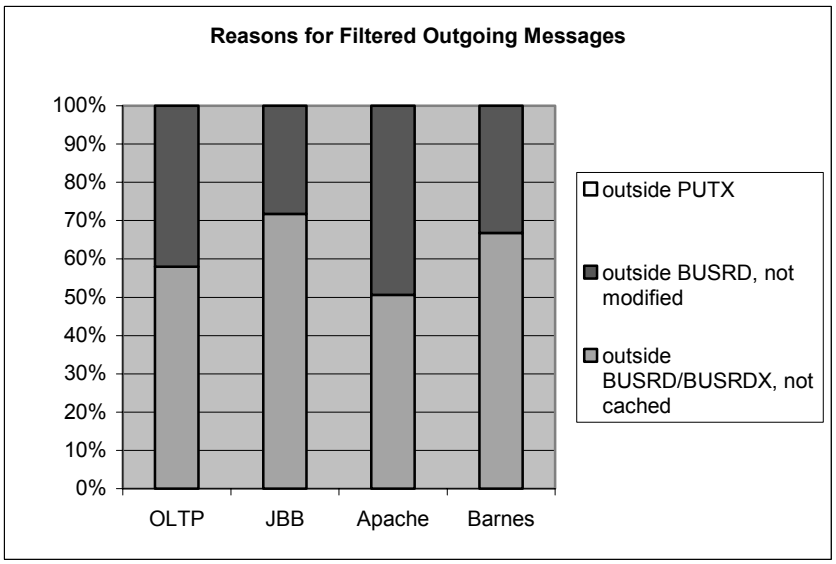

Figure 9

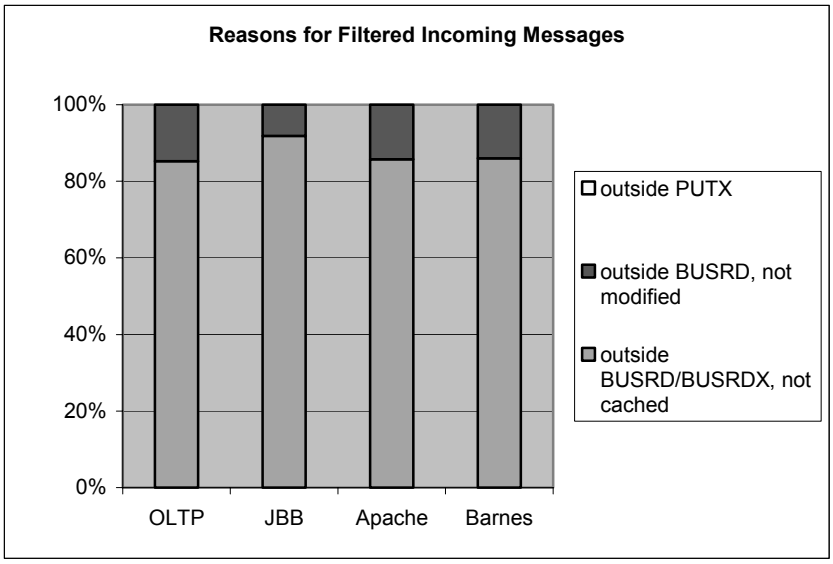

Figure 10

5.3.2 Memory Locality

We believe that Ruby's memory interleaving severely, combined with no OS support for local memory allocation, degraded the effectiveness of coherence filtering. Even unshared accesses to program stack and data areas result in communication to remote nodes. To implement effective filtering, all non-shared data should be allocated in local memory. Therefore, a system should not interleave memory at the block-level. Operating system support also must be provided to allocate memory at local nodes whenever possible.

As a simple test of this hypothesis, we modified Ruby such that any node can access the memory of a remote node as if it were local. Preliminary results show that a substantial portion of messages can be removed if memory locality is present. For example, for OLTP we saw that local bus traffic was reduced by 67.9% as opposed to our previous figure of 10.5%. We saw similar results for JBB, Apache, and Barnes-Hut.

5.3.3 Silent Replacements

For each of our benchmarks, we examined the state of the coherence filters at different intervals. One problem that we haven't solved is "silent replacement". That is, the coherence monitor is unaware of a processor replacing a block in the shared state. We believe that the number of coherence filter bits set may eventually exceed the amount of caching that is physically possible. Indeed we observed an increasing trend of bits set at every interval. However, our benchmarks did not stress the caches as few writebacks were observed. As future work, our benchmarks should be modified to cause more replacements to occur in order to determine the extent of the silent replacement problem.

6 Memory Consistency and Correctness Issues

Memory consistency has a large impact on the programmability of a shared-memory multiprocessor [4]. Many systems strive to implement sequential consistency which is considered the most straightforward model for programmers to reason with. Detecting write completion and ensuring write atomicity[1] are two major components of preserving the sufficient conditions for sequentially consistent execution. These are easily accomplished in a bus-based machine due to the simplistic interconnect. Write completion can be detected as soon as the transaction appears on the bus, and the centralized path through which all transactions pass makes write atomicity effortless.

A hierarchy of buses complicates the situation and the components of sequentially consistent execution must be reexamined. An early detection of a write can no longer be detected as soon as the transaction appears on the local bus. The transaction may traverse up and down the hierarchy. A solution to this problem is to provide indication to the issuing processor when the transaction has appeared on the entire hierarchy of buses. It may even be possible to optimize this solution by requiring indication only when the transaction reaches the top-level bus. In this case, a competing transaction from a different leaf bus may need to be NACKED.

Write atomicity is no longer straightforward because the processors are distributed across different buses. Processors on the local bus may see the transactions before other processors on remote nodes. A problem arises if a transaction on the top-level bus gets NACKed in one of the node buses. Care should be

[1] Write atomicity ensures that a write reaches all processors at the same logical time.

taken so that this does not occur. This can be handled by giving higher priority to traffic from the top-level bus compared to local node traffic. Since NACKs introduce the problem of starvation, a suitable mechanism should be in place to handle this potential problem as well.

In Ruby's implementation of a MOSI broadcast protocol, a two-level hierarchy of switches forms the interconnect. It ensures a total order on network messages and implements sequential consistency. The memory system also has a bit for every block in the system to designate whether or not the block is fresh. This allows the memory to decide whether data should be sent on a given bus transaction. Therefore, the shared/owned line present in many machines, such as the Sun E6000, is eliminated. In our implementation of coherence filtering, only messages that do not change any coherence state are filtered. Therefore, it is immaterial whether these messages appear on the bus. Thus our coherence filtering does not affect Ruby's memory consistency.

7 Related Work

The Encore Corporation built one of the earliest machines that used a method of hierarchical snooping. The Gigamax system [5] consisted of up to eight nodes each being a regular bus-based multiprocessor. The nodes were connected together to form a two-level hierarchy of buses. State monitors were used to filter transactions, however a single logical entity was not used to snoop both the local and global bus. The local state monitor was also supplemented with a remote access cache to reduce the latency of remote memory accesses.

Profusion [6] couples two Pentium Pro buses through a buffered cross-bar switch. Compared to our approach, this approach is not scalable and is limited to two Pentium Pro buses with a maximum of four processors per bus. Profusion has a method of coherence filtering which is similar to our incoming message filter. Since it is designed for just a single case of joining two buses, it allows for a local bus transaction to look at the remote tags (in the other coherence filters) and decide whether or not to forward a transaction. The Profusion design also decouples the memory from the internal Pentium Pro bus and adds L3 cache between the processors and system memory bus. The coherence monitors also filter the I/O transactions (transactions between memory and I/O devices). This improves the I/O bandwidth as I/O transactions do not have to compete with processor bus transactions.

The Sun WildFire system [7] joins two to four Sun Enterprise E6500/E5500/E4500/E3500 machines into a larger cache-coherent machine. Each of the individual machines is an SMP. When joined together they become a CC-NUMA machine with extremely large nodes. The WildFire system also supports S-COMA to achieve memory locality within a node. The Solaris operating system uses integrated hardware counters and a variant of the Reactive-NUMA algorithm to determine which pages to switch from CC-NUMA to S-COMA. The WildFire system also has modified the Solaris operating system to implement hierarchical affinity scheduling. Their results show that operating system support is essential for good performance in a hierarchical snooping machine.

A more recent use of message filtering is implemented in the Intel 870 chipset [8]. Two classes of shared-memory multiprocessor systems are supported by the 870 architecture: a single-bus processor scalable from two to four processors, and a distributed system scalable from four to more than sixteen processors. For larger systems of the latter type, a scalability port switch (SPS) is used to connect nodes of the former type. An integrated "snoop filter" tracks the state of all cache lines in processor and I/O hub caches. It tracks both local and remote transactions and maintains the state of approximately 200,000 cache lines. Each entry contains an address tag, a presence vector (1-bit per node), the cache consistency protocol state, and error correcting code bits. Indeed Intel's snoop filter is very similar to our description of coherence monitors. For filtering incoming remote messages, an associative lookup is used to reduce the amount of state necessary. The internal interconnect is constructed of a crossbar and a network of buses. Therefore, the presence vector can be used to route only those messages required to maintain cache consistency. This design reduces strain on the simplistic interconnection bus that we proposed.

Intel's snoop filter also maintains the memory consistency model of the local nodes. It contains a programmable protocol engine that handles global ordering and other conditions to maintain consistency. To further reduce bus traffic caused by remote nodes, the 870 chipset provides a hot-page mechanism to allow the operating system to migrate pages to local memories.

8 Conclusion

Building machines out of existing low-cost SMP/CMP nodes is a cost effective way to achieve higher performance. Hierarchical snooping, with filtering, is a method to build these machines using commodity parts and a low-cost interconnect. Many coherence messages can be safely removed from certain buses in the hierarchy without affecting coherence state. We have implemented filtering logic using a full-system simulator with real-world benchmarks. Our results show that filtering has limited effectiveness with memory distributed across the entire system. Memory locality is crucial to reducing the negative traffic effects of a broadcast protocol. A bus-based hierarchical system should be constructed with coarse-grained memory interleaving and operating system support to achieve optimal performance.

REFERENCES

[1] A. Agarwal, R. Simoni, M.Horowitz, and J.Hennessy. An Evaluation of Directory Schemes for Cache Coherence. In *Proceeding of the 15th Annual International Symposium on Computer Architecture*, pages 280-289, 1988.

[2] L.A. Barroso et al. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture,* pages 282-293, June 2000.

[3] P. S. Magnusson et al. SimICS/sun4m: A Virtual Workstation. In *Proceedings of Usenix Annual Technical Conference,* June 1998.

[4] Sarita V. Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer,* 29(12):66-76, December 1996

[5] David E. Culler, Jaswinder Pal Sing, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann Publishers; ISBN: 1558603433; 1st edition, August 1998.

[6] G.White and P. Vogt. Profusion (tm):A Buffered, Cache Coherent Crossbar Switch. In *IEEE Hot Interconnects*, pages 87-96, 1997

[7] E. Hagersten and M. Koster. WildFire: A Scalable Path for SMPs. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture (HPCA-5),* pages 171-181, Orlando, Florida, January 1999.

[8] Briggs et. Al. Intel 870: a building block for cost-effective, scalable servers. IEEE Micro , Volume: 22 Issue: 2 , Mar/Apr 2002.