Appears in the proceedings of the
8th Annual International Symposium on High-Performance Computer Architecture (HPCA-8)
Cambridge, MA, February 2-6, 2002

Bandwidth Adaptive Snooping

Milo M. K. Martin, Daniel J. Sorin, Mark D. Hill, and David A. Wood
Computer Sciences Department
University of Wisconsin-Madison
{milo, sorin, markhill, david}@cs.wisc.edu
http://www.cs.wisc.edu/multifacet/

Abstract

This paper advocates that cache coherence protocols use a bandwidth adaptive approach to adjust to varied system configurations (e.g., number of processors) and workload behaviors. We propose Bandwidth Adaptive Snooping Hybrid (BASH)*, a hybrid protocol that ranges from behaving like snooping (by broadcasting requests) when excess bandwidth is available to behaving like a directory protocol (by unicasting requests) when bandwidth is limited.* BASH *adapts dynamically by probabilistically deciding to broadcast or unicast on a per request basis using a local estimate of recent interconnection network utilization. Simulations of a microbenchmark and commercial and scientific workloads show that* BASH *robustly performs as well or better than the best of snooping and directory protocols as available bandwidth is varied. By mixing broadcasts and unicasts,* BASH *outperforms both snooping and directory protocols in the mid-range where a static choice of either is inefficient.*

1 Introduction

Snooping and directory protocols are the two dominant classes of cache coherence protocols for hardware shared memory multiprocessors. In a snooping system, a processor broadcasts a request for a block to all nodes in the system to find the owner (which could be memory) directly. In a directory protocol, a processor unicasts a request to the home directory for the block, the directory forwards the request to the owner (trivial when the directory is the owner), and the owner responds to the requestor. Thus, snooping protocols can achieve lower latencies than directory protocols on sharing misses (a.k.a., cache-to-cache transfers or dirty misses) by avoiding the indirections incurred by directories.

By broadcasting to avoid indirections for sharing misses, snooping can outperform directories when bandwidth is plentiful, but directories outperform snooping when bandwidth is limited. In today's commercial workloads, sharing misses comprise a significant fraction of level two cache misses and correspondingly impact performance [3, 18].

This work is supported in part by the National Science Foundation with grants EIA-9971256, CDA-9623632, and CCR-0105721, an IBM Graduate Fellowship (Martin), an Intel Graduate Fellowship (Sorin), two Wisconsin Romnes Fellowships (Hill and Wood), and donations from Compaq Computer Corporation, IBM, Intel, and Sun Microsystems.

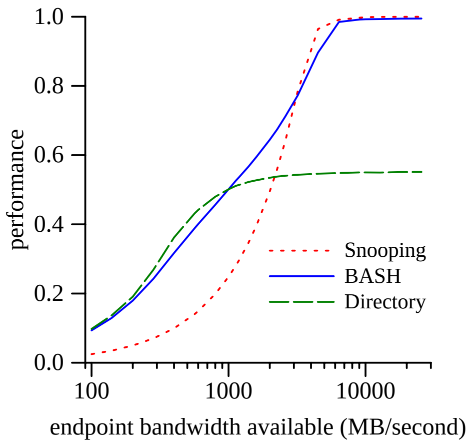


Figure 1. Performance vs. available bandwidth for a microbenchmark on 64 processors

Martin et al. [23] showed that snooping can outperform directories on a medium size (16 processor) system running commercial workloads, at the cost of additional bandwidth. In Figure 1, we plot performance versus available interconnection network bandwidth for a simple locking microbenchmark on 64 processors (described in Section 4). This graph reinforces the intuition that the relative performances of snooping and directories depend on available bandwidth. The point at which increasing bandwidth does not improve performance occurs at a bandwidth much greater (by a factor of 5) for snooping than for directories.

Designing a single protocol to provide high performance for many system configurations and workloads is difficult. Hennessy writes [14], "[W]e don't have a coherency scheme that does well under all these situations: from small to large processor counts, different levels of [software] optimization, and differing cache sizes." We advocate an adaptive approach to address this challenge.

There are two reasons why an adaptive scheme is desirable. First, due to the trend towards integrating the coherence protocol logic and the processor on the same die [7, 12, 30], a single protocol must suffice for multiple hardware configurations (processor counts, cache sizes, and interconnection networks). If the microprocessor is to be used in a scalable system, the protocol must also be scalable. For example, since Alpha 21364 [12] systems can scale to hundreds of processors, a directory protocol is currently the only option. However, many scalable system designs are used for smaller systems that could be better served by a snooping protocol. Even the vast majority of scalable systems sold are systems of moderate size. For example, a recent essay [24] estimated that, of the 30,000 Origin 200/2000 [20] systems shipped, less than 10 systems contained 256 or more processors (~0.03%), and less than 250 of the systems had 128 processors or more (~1%).

Second, statically choosing between a directory protocol and a snooping protocol is not desirable due to the varying behaviors of different workloads and the time-varying behavior within a workload. Our results show that for a given system size and configuration, some workloads perform better with snooping while other workloads are better served by a directory protocol. Further, a given workload's demand on system bandwidth varies dynamically over time. For example, different phases of behavior for a multiprocessor database workload have been observed with periods on the order of minutes [25]. During a phase of high cache miss rate, a broadcast request in a snooping protocol could further congest the system.

For these two reasons, an adaptive hybrid protocol that provides robust performance is preferable to a static choice of either snooping or directories. Our contribution is an adaptive mechanism (Section 2) and a hybrid protocol (Section 3) that leverages this mechanism to perform like snooping (by broadcasting requests) if bandwidth is plentiful and perform like a directory protocol (by unicasting requests) if bandwidth is limited. Our protocol, *bandwidth adaptive snooping hybrid (BASH)*, adapts dynamically to the available bandwidth to provide robust performance. Using a microbenchmark (shown in Figure 1 and further explored in Section 4) and commercial workloads (Section 5), we show that the performance of *BASH* tracks that of the directory protocol in the limited bandwidth case and tracks that of snooping in the plentiful bandwidth case. Moreover, in the mid-range where the performances of snooping and directories are similar, *BASH* outperforms both protocols.

2 A Bandwidth Adaptive Mechanism

In this section, we describe the mechanism each processor uses to decide dynamically whether a request should be broadcast or unicast.

2.1 Goal and Approach

BASH's goal is to minimize average miss latency. Given infinite bandwidth, broadcasting all requests would achieve this goal by avoiding all indirections for sharing misses. However, the finite bandwidth of interconnection networks can lead to congestion and queuing delays that outweigh the benefit of avoiding indirection. Nevertheless, mean queuing delay only dominates when the interconnect is highly utilized. Figure 2 illustrates this trade-off with a simple queuing

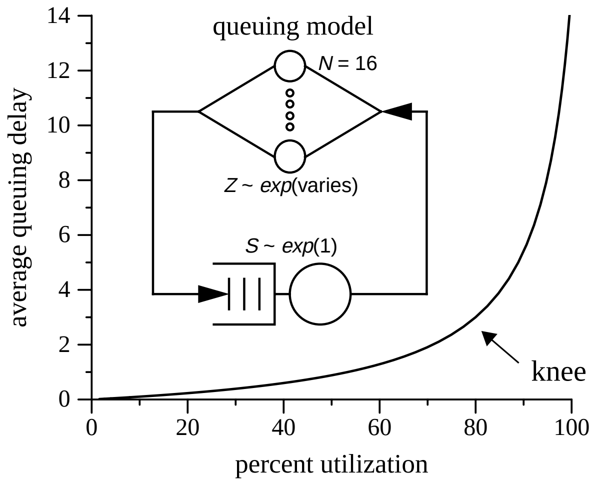


Figure 2. Average queuing delay vs. utilization for a sim-

network. Above the "knee" in the curve, increasing utilization dramatically increases response time.

The mechanism we propose for *BASH* uses feedback to keep the interconnect utilization below this critical level and thus mitigate queuing delays. Our mechanism uses a processor-local estimate of interconnect utilization to keep utilization below a pre-specified threshold by dynamically adjusting the probability of broadcasting. Feedback control theory suggests that the mechanism should adapt to changes in interconnect congestion, but not so quickly that it overshoots and leads to oscillation [19]. As described in Section 2.2, our mechanism avoids oscillation by adapting relatively slowly and using a probabilistic mechanism to decide whether or not to broadcast. In initial experiments, we tried a simpler mechanism that switched between always and never broadcasting, and we observed unstable behavior due to oscillation between these two extremes.

2.2 Implementation

Our bandwidth adaptive implementation uses a simple mechanism to estimate the interconnect utilization and adjust the rate of broadcast. The mechanism consists of three parts: (1) estimating interconnect utilization, (2) adjusting the probability of broadcasting, and (3) determining whether or not to broadcast a specific request.

First, a processor uses the utilization of its link to the interconnection network as a local estimate of global interconnect utilization. While this local information does not capture certain global effects, it is easy to obtain and correlates strongly with global interconnect utilization due, in part, to the broadcast nature of the requests that are most likely to cause contention. Each processor uses a simple, signed, saturating *utilization counter* to calculate if the link utilization is above or below a static threshold. Figure 3 illustrates the counter's operation assuming a target link utilization of 75%. For each cycle, the mechanism increments the counter by one if the link is utilized, and decrements it by

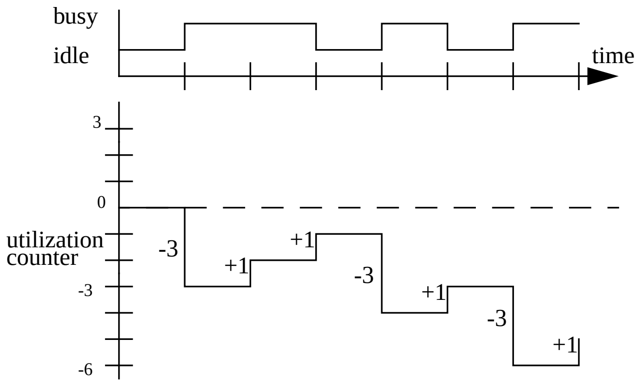


Figure 3. Example operation of the utilization counter

three otherwise. When the counter is sampled, a positive value means that the link was used more than the threshold, and a negative value means that the link was used less than the threshold. The counter is reset to zero after each sample. Since the link in Figure 3 was used 4 out of the previous 7 cycles (57%), the counter would be, as expected, negative ($4\times1 + -3\times3 = -5$).

Second, an unsigned, saturating *policy counter* averages the utilization information and determines the fraction of requests that should be broadcast. Our mechanism samples the utilization counter every *n* cycles (the sampling interval), and it increments/decrements the policy counter by one if the utilization was greater/less than the threshold. Thus, a larger value of the policy counter corresponds to a lower probability of broadcast.

Finally, a given request is unicast or broadcast with a probability proportional to the policy counter. For example, an 8-bit policy counter with the value of 100 implies that a request should be unicast with probability of 100/255 or 39%. For each out-going request, the processor compares the policy counter to a randomly generated integer the same size as the policy counter. The processor unicasts if the policy counter is smaller than the random number, and it broadcasts otherwise. Pseudo-random numbers can be generated easily by a linear feedback shift register [11]. Our mechanism generates random numbers and performs the comparison to the policy counter off the critical path, allowing the mechanism to have negligible impact on miss latency.

Through experimentation, we selected a utilization threshold of 75%, a sampling interval of 512 cycles, and a policy counter size of 8 bits. A smaller sampling interval and policy counter size would enable the mechanism to respond more rapidly to different workload phases, but they would make the mechanism more susceptible to oscillation. With these parameters, our adaptive mechanism can change from 100% unicast to 0% unicast (or vice versa) in 512×255 = ~130,000 cycles in which the measured utilization is above/below the threshold. Since the uncontended round-trip latency for an L2 cache misses is around 125 cycles (for our target system), the mechanism can adapt over its entire range in ~1000 cache misses.

3 A Bandwidth Adaptive Snooping Protocol

BASH, our bandwidth adaptive snooping hybrid protocol, incorporates features of both snooping and directory protocols. While there are different ways to combine these two types of protocols to synthesize a hybrid, we choose to form *BASH* from an aggressive snooping protocol and a recently-published directory protocol. These two protocols have some (surprisingly) common features that we exploit in creating our hybrid protocol.

First, in Sections 3.1 and 3.2, we describe the two protocols used as the foundation of our hybrid. They will also serve as the base cases against which we will compare *BASH* for its evaluation in Sections 4 and 5. Second, in Section 3.3, we describe our synthesis of these two protocols. A key issue in this synthesis is reconciling the differences between the methods used by snooping and directories to enforce ordering between racing transactions. For each protocol described in this section, Figure 4 illustrates the operation of two typical protocol transactions, so as to highlight the similarities and differences between the protocols. Third, in Section 3.4, we discuss several issues relating to *BASH*, including livelock/deadlock, scalability, complexity, and verification.

All three protocols are write-invalidate, use the MOSI states [29], allow processors to silently downgrade from S to I, support several transactions (e.g., get an S copy, get an M copy, writeback an M or O copy), and interact with the processors to support a consistency model. Our results assume sequential consistency.

3.1 A Snooping Protocol

Traditional snooping protocols rely on a totally ordered delivery of coherence requests to (1) enable processors and memories to agree on the next owner and (2) obviate the need for explicitly acknowledging invalidations of shared blocks. Since racing transactions are totally ordered, a snooping cache controller can make a strictly local decision on each transaction and infer that other nodes will make compatible decisions. We base *BASH* on an aggressive MOSI snooping protocol (which we refer to as *Snooping*) that is loosely based on the Sun UE10000 [6].

We assume that our snooping protocol uses separate virtual networks for requests and responses. The request network must enforce a total order, but it need not ensure synchronous broadcast [23]. Modern snooping systems use address-interleaved hierarchical switches to achieve high throughput for ordered broadcasts [6]. The response network has no ordering requirements and can use any unordered switched interconnection network.

A processor broadcasts its requests to all other nodes on the request network, and all processors snoop all requests. The owner (potentially memory) sends data directly to the requestor over the response network. A requestor must also snoop its own request, which serves as a *marker* to indicate its place in the total order.

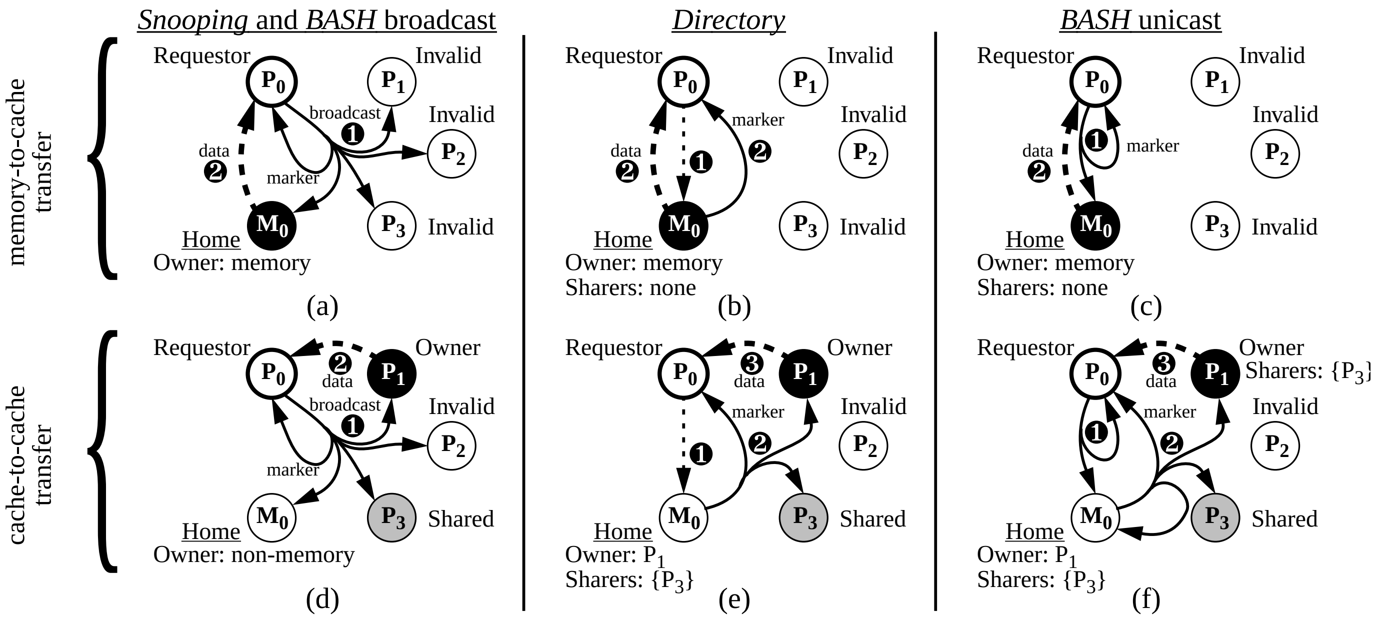


Figure 4. Example operation for *Snooping*, *Directory*, and *BASH* with four processors (P_0-P_3) and memory module (M_0). (a)-(c) illustrate a request by P_0 for exclusive access to a block that is satisfied by memory (a memory-to-cache transfer). (d)-(f) illustrate a similar request by P_0 where P_1 is the owner and P_3 is a sharer (a cache-to-cache transfer with an invalidation). Totally ordered messages are drawn with solid lines and unordered messages are drawn with dashed lines.

A memory controller behaves much like it does in a traditional snooping protocol, except for keeping one bit of state per block to indicate if it is the owner, similar to what was done in the Synapse N+1 [9]. This bit of state eliminates the need for a global owned snoop response. If memory is the owner, it responds with data.

3.2 A Directory Protocol

Traditional directory protocols rely on unordered or, at most, point-to-point ordered interconnection networks. While the lack of ordering facilitates building scalable interconnects, it requires the protocols to use explicit acknowledgment messages and transient states to enforce order between racing transactions.

A recent directory protocol, implemented in the AlphaServer GS320 [10], uses a totally ordered multicast interconnect to optimize the protocol and eliminate explicit acknowledgments. Like *Snooping*, the GS320 uses a marker on the ordered network to indicate a request's place in the total order. Below we describe a protocol (which we refer to as *Directory*) modeled after the GS320.

Directory uses three virtual networks: an unordered request network, a totally ordered network for requests forwarded by the directory to processors, and an unordered network for responses from processors and directories. The totally ordered forwarded request network supports multicasting and, as in *Snooping*, eliminates the need for acknowledgment messages. In the GS320, this totally ordered interconnect is implemented as an 8-way crossbar that connects to 4-processor nodes, supporting up to 32 processors.

Processors unicast all requests to the directory at memory. The memory controller maintains a directory with state about each block for which it is the home, including the owner and a superset of the sharers. Like all directory protocols, the memory controller responds directly when it has sufficient permissions and forwards the request when it does not. On a direct response, the memory controller sends the data on the unordered response network and a marker message on the forwarded request network. The latter indicates the request's place in the total order. Forwarded requests are sent via the totally ordered multicast network to the owner, sharers, and requestor. The marker sent to the requestor allows it to infer where the forwarded request occurs in the total order. The owner and sharers observing the same total order obviates the need for explicit acknowledgement messages.[1] Racing request messages are ordered at the directory, and they are either processed locally or forwarded on the ordered multicast network.

When a processor responds to a request that is forwarded to it by the directory, it does not need to send an explicit acknowledgment, since the forwarded request network is totally ordered. Processors also monitor the forwarded request network for the marker messages that indicate their request's place in the total order.

3.3 An Adaptive Hybrid Protocol

In this section, we describe how our hybrid protocol integrates *Snooping* and *Directory*, and we then discuss a number of issues that arise from the integration. The integration is possible because both protocols use a totally ordered net-

1. Along with the total order, the GS320 protocol guarantees that all forwarded requests can be processed at the target node, which is also necessary to eliminate explicit acknowledgments.

work to eliminate the need for explicit acknowledgments and preserve order between racing transactions. However, we must resolve a key discrepancy in how these protocols order racing transactions. In *Snooping*, racing request messages are ordered entirely by the request network. In *Directory*, the order of racing messages is determined by the order in which they are processed by the directory controller; the ordered multicast network simply preserves this order for forwarded requests. Resolving this difference requires subtle, but relatively simple, hardware. Our adaptive protocol handles these races similarly to Multicast Snooping [4, 28], and *BASH* can be considered a special case of this more general protocol.

Our hybrid protocol uses two virtual networks. Requests use a totally ordered multicast request network, but no restrictions are placed upon its topology or synchrony. As in *Snooping*, the total order of requests—necessary for correct coherence protocol behavior and enforcing a memory consistency model—is determined by their ordering on the request network. Responses travel on an unordered point-to-point data network.

From the requestor's point of view, *BASH* behaves similarly to *Snooping*, except that the cache controller must choose whether to broadcast or unicast each request. Our policy for deciding between broadcast and unicast was explained in Section 2.2. Writeback requests are always unicast. Since the request network is the ordering point, *a BASH* "unicast" request is actually a dualcast sent to both the home node and back to the requestor. Similar to *Snooping*, the return of the request acts as the marker and informs the requestor of the transaction's place in the total order. Processors respond to incoming requests as in *Snooping*, except processors must detect and ignore retried requests as discussed below.

Like *Directory*, *BASH*'s memory controller maintains the owner and a superset of the sharers for each block for which it is the home. The memory controller's basic operation is to compare the owner/sharer information from the directory against the set of nodes that received the request message to determine if the request was sent to a sufficient set of nodes.[2] If the request was sent to the owner and all necessary sharers, the memory controller updates the directory state and responds with data as necessary. For broadcast requests, a *BASH* memory controller behaves as in *Snooping*, with the addition of updating the directory state as needed. For unicast requests that find data at the home, the memory controller behaves as in *Directory*, immediately updating the state and responding with data. Unlike *Directory*, *BASH* need not send a marker message, since this was already sent with the original request.

When a processor issues a unicast for a block that is owned in a third node, the memory controller behaves as in *Directory*, with two important differences. First, it does not update the directory state, because the request is not yet satisfied. Second, instead of forwarding the request on a separate forwarded request network, it *retries* the request as a multicast on the totally ordered request network. The multicast set for the retried request includes the memory controller in addition to the owner, sharers, and requestor. Assuming no racing transactions, the owner will satisfy the retried request.

More complex cases occur when broadcasts, unicasts, and multicast retries race for the same block. The memory controller has a "window of vulnerability" between when the original and retried requests are ordered on the request network. If a broadcast request for the same block is ordered during that window, the retried request's multicast set may be insufficient, forcing the request to be retried again. Since any non-broadcast request may require retries, there exist livelock and deadlock issues, discussed in Section 3.4.

3.4 Discussion

With *BASH*, as for any coherence protocol, one must address the issues of livelock/deadlock, scalability, complexity, and verification.

Livelock and deadlock. Retrying requests presents the twin problems of livelock and deadlock. Livelock could occur, for example, if a non-broadcast is competing with broadcasts for a heavily contended block; no matter how many times the memory controller retries a non-broadcast request, there is no guarantee that it will ever succeed. *BASH* avoids livelock by broadcasting—which is guaranteed to succeed—on its third retry.

Most multiprocessor systems avoid interconnection network deadlocks (in part) by accessing virtual networks in a strict order to avoid cyclic dependences. By retrying requests on the same virtual network, *BASH* introduces a circular dependence—and thus potential deadlock—because it may use the request network multiple times to process a single request. Rather than avoiding deadlock, *BASH* detects a potential deadlock and resolves it by sending a negative acknowledgment (nack) to the original requestor. Specifically, if the memory controller cannot allocate a network buffer for the retry, it sends a nack to the requestor on the data response network. The requestor can then reissue its request as a broadcast, which is guaranteed to succeed.

Scalability. *BASH* is more scalable than snooping protocols because it does not require all requests to be broadcast, yet it is less scalable than directory protocols that do not rely on a totally ordered interconnect [20, 21]. Fortunately, hierarchical switches can be used to make high-bandwidth totally ordered interconnects. Removing the broadcast-always behavior of snooping may allow the design of a well-balanced system of significantly larger size than would be possible with broadcast snooping. Examples of real systems with a large number of processors and an ordered intercon-

2. If a processor is the owner, it also tracks the sharer set and determines if the request was sufficient, so as to make a decision consistent with that of the memory controller.

nect include the AlphaServer GS320 [10], Sun's UE10000 [6], and Fujitsu's PRIMEPOWER 2000 [15], and these systems support 32, 64, and 128 processors, respectively. In addition, Martin et al. [23] recently proposed an approach for an ordered interconnect with no central bottleneck. This approach allows for more general, and perhaps more scalable, interconnect topologies that still maintain a total order.

Complexity. As a hybrid of two protocols, *BASH* is more complex than either protocol on which it is based, and the difficulty of verification is directly related to the complexity. However, complexity does not grow as much as one might expect because of the strong similarities between the underlying protocols. For example, *BASH* processors react identically to requests, regardless of whether they are unicasts, multicasts, or broadcasts. In fact, a broadcast in this system appears as though the directory simply specified an overly generous set of sharers to invalidate.

As a rough measure of the complexity of each protocol, Table 1 displays the numbers of states (both stable and transient), events, and state transitions for each controller. Compared to *Snooping* and *Directory*, we find *BASH* has a comparable number of states, but about 50% more events and double the number of transitions. While not all state/event combinations are equally difficult to verify, and the numbers of states and events depend somewhat on how one chooses to express a protocol, implementing *BASH* should be less difficult than including both a snooping and directory protocol in the same system.

Verification. To gain confidence in the correctness of *BASH*, we have used both random testing and formal methods. All three protocols—*Snooping*, *Directory*, and *BASH*—were tested using a stand-alone random tester. This tester uses false sharing, random action/check (store/load) pairs [33], and widely variable message latencies to force each protocol through a myriad of corner cases. We ran the tester through millions of coherence operations and uncovered numerous subtle race conditions. In the end, our tool reported full coverage for all state transitions with no detected errors.

In our experience, random testing is excellent at finding protocol errors even for complex protocols, but it is little help for finding deadlock, livelock, and memory consistency model errors. We have explored more formal methods, including model checking and Lamport clocks [26], to address these issues. A technical report [28] describes our experience verifying an enhanced version of the Multicast Snooping protocol [4].[3] Since *BASH* is based upon this enhanced protocol, this proof carries over directly to *BASH*.

3. The original Multicast Snooping protocol described in Bilir et al. [4] must nack insufficient requests. Sorin et al. [28] describe the important optimization of retries at the directory. This optimization allows an insufficient unicast in *BASH* to have the same latency as a request that must be forwarded by the directory.

Table 1. States, events, and transitions for *BASH*, *Snooping*, and *Directory*

Protocol	Total			Cache			Mem/Dir		
	States	Events	Trans.	States	Events	Trans.	States	Events	Trans.
BASH	21	23	114	17	14	94	4	9	20
Snooping	19	13	68	17	9	61	2	4	7
Directory	21	13	75	17	9	61	4	4	14

4 Microbenchmark Performance Evaluation

Before presenting performance results for commercial workloads using full-system simulation, we present results for a simple locking microbenchmark. The microbenchmark is easy to understand and allows us to explore the effects of system scaling and workload intensity. We start by describing our microbenchmark and simulation methods. We then explore the performance of *BASH* over a range of available bandwidths, utilization thresholds, system sizes, and workload intensities.

4.1 Microbenchmark

In the microbenchmark, each processor acquires and releases locks that are generally uncontended. After the release of one lock, a processor immediately attempts to acquire another lock. Each processor can have at most one outstanding request. Since we choose the number of locks to be approximately the number of lines per cache, the microbenchmark incurs sharing misses almost exclusively. While this is near the worst-case performance scenario for directory protocols, smaller fractions of sharing misses do not qualitatively change our conclusions, as shown in Section 5.

4.2 Simulation Methods

Before discussing our microbenchmark results, we describe our memory system simulator and timing assumptions. Our memory hierarchy simulator captures timing races and all state transitions (including transient states) of the coherence protocols in cache and memory controllers. We consider integrated processor/memory nodes connected via a single link to an interconnection network. Since *BASH*, *Snooping*, and *Directory* all require a total order of requests, but do not require a specific interconnection network topology, we abstract the details of the interconnect design by modeling a fixed latency crossbar with limited bandwidth and contention at the endpoints. All request, forwarded request, and retried request messages are 8 bytes, and data responses are 72 bytes (64 byte data block with an 8 byte header).

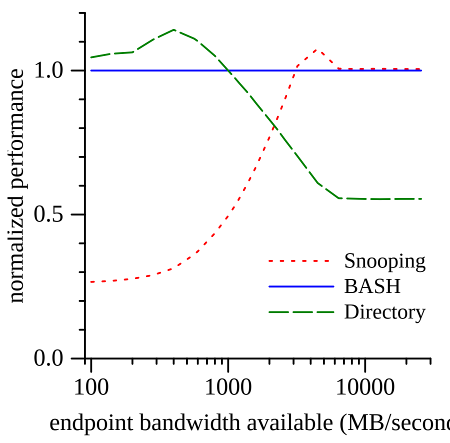


Figure 5. Normalized performance vs. available bandwidth for a micro-benchmark on 64 processors

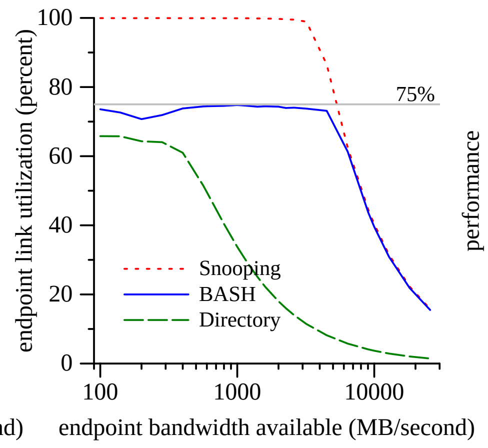


Figure 6. Utilization vs. available bandwidth for a microbenchmark on 64 processors

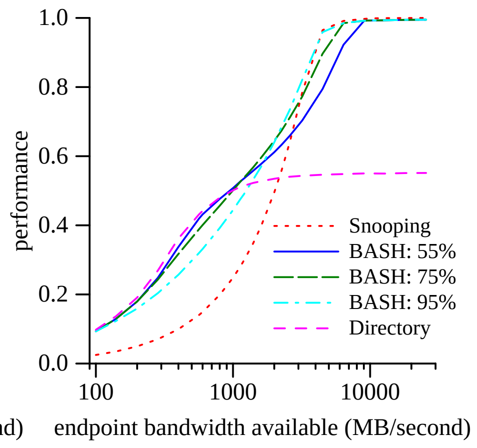


Figure 7. Sensitivity to utilization threshold value for a microbench-mark on 64 processors

To approximate the published latencies of systems like the Alpha 21364 [12], we selected 50 ns for each interconnection network traversal (which includes wire propagation, synchronization, and routing) and 80 ns for memory DRAM access time. When a protocol request arrives at a processor or memory, it takes 25 ns or 80 ns, respectively, to provide data to the interconnect. These assumed latencies result in a 180 ns latency to obtain a block from memory in all three protocols, a 125 ns latency for a cache-to-cache transfer for both a *Snooping* and a broadcast *BASH* request, and a 255 ns latency for a cache-to-cache transfer for a *Directory* and a unicast *BASH* request.

For *Snooping* and successful *BASH* requests, the cache-to-cache transfer latency is smaller than the memory latency (~70% of memory latency: 125 ns vs. 180 ns). We assume that this scenario is carefully optimized, as is the case for the IBM NorthStar (RS64-II) SMPs [5], where a cache-to-cache transfer latency is ~55% of main memory latency [17]. The cache-to-cache transfer latency for *Directory* requests and for *BASH* requests that need to be retried is significantly higher than a fetch from memory, due to the indirection through the directory (memory controller). An indirected request incurs the latencies of a DRAM directory access, supplying the data from the cache, and three interconnect traversals. An SRAM directory or directory cache would mitigate the latency of accessing directory state. However, due to a third traversal of the interconnect, a *Directory* cache-to-cache transfer would still have a greater latency than that of a broadcast request.

4.3 Microbenchmark Results

We compare the microbenchmark performance (in units of normalized lock acquires per nanosecond) of *BASH* against *Snooping* and *Directory*. Figure 5 presents the same data as shown in Figure 1, except in Figure 5 performance is normalized to that of *BASH*. The graph shows that, for a 64-processor system, *BASH* performs like *Directory* in the limited bandwidth case and like *Snooping* in the plentiful bandwidth case. At extremely low available bandwidths, *Directory* outperforms both other protocols; *BASH* is ~10% worse than *Directory* due to additional marker messages (shown in Figure 4(f)). In the middle range (near where the *Snooping* and *Directory* curves intersect), *BASH* outperforms both protocols by up to 25%. As the available bandwidth increases, *Snooping* outperforms *BASH*, because *BASH* conservatively reduces its rate of broadcast. As bandwidth becomes even more plentiful, *BASH* always broadcasts requests, and thus the performances of *BASH* and *Snooping* converge.

Interconnection network utilization. To further explain these performance results, we plot interconnection network endpoint utilization versus available bandwidth in Figure 6. *Snooping* uses large amounts of bandwidth and thus over-utilizes the network in the case of limited bandwidth, while *Directory* under-utilizes the network when bandwidth is plentiful. *BASH* achieves the desired 75% utilization (denoted by the horizontal line) until bandwidth is so plentiful that even by always broadcasting it does not reach 75% utilization. Figure 5 shows that this is also the point at which the performances of *BASH* and *Snooping* converge.

Utilization threshold selection. In Figure 7, we plot performance versus available bandwidth for three threshold values, and we observe that performance is not overly sensitive to the exact threshold value selected. Even for thresholds as high as 95% or as low as 55%, the qualitative performance of *BASH* remains similar. While we choose 75% for our experiments, we do not claim that 75% is the optimal threshold for this or any other workload. In practice, it has achieved good performance.

Adaptation to system size. To explore the potential benefits of *BASH* over a range of system sizes, we plot performance *per processor* versus available bandwidth for a range of pro-

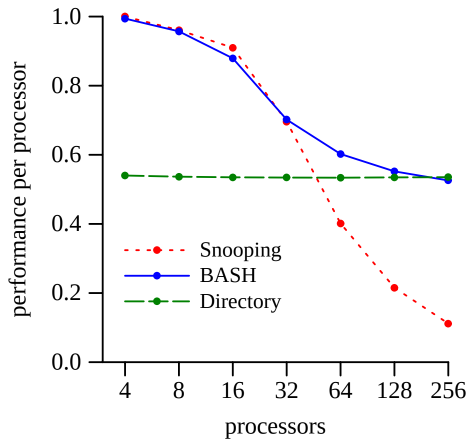


Figure 8. Impact of system size for a microbenchmark with 1600 MB/second endpoint bandwidth per processor

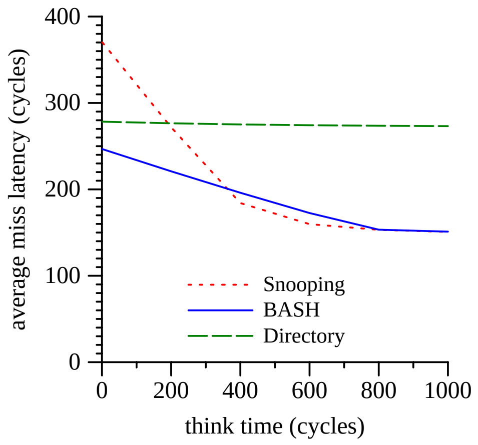


Figure 9. Average miss latency vs. think time for a microbenchmark on 64 processors with 1600 MB/second endpoint bandwidth per processor

cessor counts in Figure 8. Endpoint link bandwidth per processor is fixed at 1600 MB/second, and the processor count is plotted logarithmically on the x-axis. We observe that the line for *Directory* is nearly flat, signifying near-perfect scalability. *BASH* performs as well as *Snooping* for small systems and as well as *Directory* for large systems. In the midrange, *BASH* outperforms both other protocols. For this specific design point, *Directory* far outperforms *Snooping* for processor counts above 64. Higher link bandwidth would help *Snooping*, but the figure illustrates why directory protocols are attractive for large-scale systems and why an adaptive scheme is desirable in general.

Adaptation to workload intensity. To explore the impact of workload behavior on performance, we adjust the intensity of the microbenchmark's memory traffic. The memory traffic's intensity is adjusted by adding a *think time* (i.e., the time between when a processor releases one lock and acquires another). Figure 9 plots average miss latency (lower is better) versus think time. Increasing think time corresponds to decreasing workload intensity. Our results show that, for a fixed system configuration (64 processors and 1600 MB/second endpoint link bandwidth), the choice between snooping and directories depends on workload intensity (i.e., think time or miss rate).

5 Workload Performance Evaluation

While microbenchmarks can provide insight into behavior and allow exploration of the design space, *BASH*'s performance on commercial workloads matters most. This section describes our benchmarks, target system assumptions, and simulation techniques for evaluating the performance of bandwidth adaptive snooping. We compare *BASH* against *Snooping* and *Directory* using full-system simulation of a 16-processor SPARC system running four commercial workloads and one scientific benchmark. Unfortunately, due to the complexities of full-system simulation and commercial workload setup and tuning, we are currently unable to obtain results with more than 16 processors. To approximate *BASH*'s performance on larger systems, we also present simulation results for a 16 processor system in which the bandwidth cost of broadcasting is four times greater than normal.

5.1 Benchmarks

Table 2 describes our benchmarks. We concentrate on commercial workloads, such as database and web servers, but we also include one scientific application for comparison. We run all of the commercial workloads for a warm-up period to bring the system to a steady state before measurement. To simplify the simulations of our commercial workloads, the client does not model think time between requests, and the client and server are collocated on the same simulated machine.

5.2 Target System Assumptions

We evaluate 16-node SPARC systems running unmodified Solaris 8. Each node contains a processor core, level one caches, a unified level two cache (4 MB, 4-way set associative, 64-byte blocks), a cache controller, and a memory controller for part of the globally shared memory (2 GB total). We assume that a processor and level one caches would complete four billion instructions per second if the memory system beyond the level one caches was perfect. This could be accomplished, for example, with a 2 GHz processor that has a perfect-L2-cache IPC (instructions per cycle) of 2.

5.3 Simulation Methods

We simulate our target systems with the Simics full-system multiprocessor simulator [22], and we extend Simics with a memory hierarchy simulator (described in Section 4.2) to compute execution times. Simics is a system-level architectural simulator developed by Virtutech AB that is capable of booting unmodified commercial operating systems and running unmodified applications. We use Simics/sun4u, which simulates Sun Microsystems' SPARC v9 platform architecture (e.g., used for Sun E6000s). Simics

Table 2. Benchmark descriptions

On-Line Transaction Processing (OLTP): DB2 with a TPC-C-like workload. The TPC-C benchmark models the database activity of a wholesale supplier, with many concurrent users performing read/write transactions against the database. Our OLTP workload is based on the TPC-C v3.0 benchmark using IBM's DB2 v7.2 EEE database management system and an IBM benchmark kit to build the database and model users. Our experiments use a 1 GB 10-warehouse database stored on five raw disks and an additional dedicated database log disk. There are 128 simulated users (8 per processor). The database was warmed up for 10,000 transactions before taking measurements, and our results are based on runs of 1000 transactions.
Static Web Content Serving: Apache with SURGE. Web servers such as Apache have become an important enterprise server application. We use Apache 1.3.19 for SPARC/Solaris 8 configured to use pthread locks and minimal logging as the web server, and SURGE [2] to generate web requests. Our experiments used a repository of 2000 files (totalling ~50 MB). There are 160 simulated users (10 per processor). The system was warmed up for ~80,000 transactions, and our results are based on runs of 2,500 requests.
Java Server Workload: SPECjbb. SPECjbb2000 is a server-side java benchmark that models a 3-tier system, focusing on the "middleware" server business logic and object manipulation. We used Sun's HotSpot 1.4.0 Server JVM. The benchmark includes driver threads to generate transactions. Our experiments used 24 threads and 24 warehouses (with a data size of approximately ~500MB). The system was warmed up for 100,000 transactions, and our results are based on runs of 100,000 transactions.
Dynamic Web Content Serving: Slashcode. Our Slashcode benchmark is based on an open-source dynamic web message posting system used by slashdot.org. We use Slashcode 2.0, Apache 1.3.20, and Apache's `mod_perl` 1.25 module for the web server, and MySQL 3.23.39 as the database engine. A multithreaded user emulation program is used to simulate user browsing and posting behavior. The database is a snapshot of the slashcode.org site, and it contains ~3000 messages. There are 48 simulated users (3 per processor). The system was warmed up for 240 transactions before taking measurements, and our results are based on runs of 50 transactions.
Scientific application: Barnes-Hut from SPLASH-2. We selected one application from the SPLASH-2 benchmark suite [32]: *barnes-hut* with 64K bodies. The benchmark was compiled with Sun's WorkShop C compiler and uses the PARMACS shared-memory macros used by Artiaga et al. [1]. The macro library was modified to enable user-level synchronization through test-and-set locks rather than POSIX-thread library calls. We began measurement at the start of the parallel phase to avoid measuring thread forking.

is a functional simulator only, and it assumes that each instruction takes one simulated cycle to execute (although I/O may take longer), but it provides an interface to support our detailed memory hierarchy simulation. We use Simics to generate blocking requests to a unified single level cache. We use this simple processor model to enable tractable simulation times for full-system, multiprocessor simulation of commercial workloads.

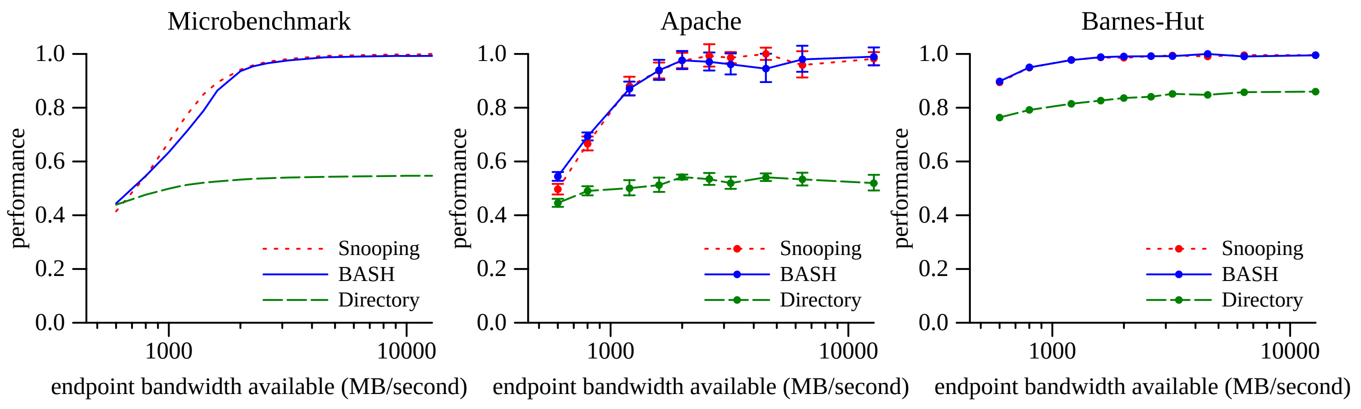
Since full-system simulation captures kernel behavior and inter-processor timing, small changes in system timing can lead to significant variations in execution time. For example, we find that our operating system intensive workloads (OLTP, Slashcode, and Apache) exhibit more variation than workloads that are less operating system intensive (SPECjbb and Barnes-Hut). To overcome observed instabilities, we calculate the arithmetic mean and standard deviation of multiple simulations to estimate experimental uncertainty. We plot the mean and, if the coefficient of variation is greater than 1%, error bars at plus/minus one standard deviation for all data points. To gather multiple data points, we perturb our otherwise deterministic simulations by adding a small random delay to each request.

5.4 Results

We now present results for the workloads described in Table 2. Figure 10 illustrates the performances of the protocols over a range of bandwidths for 16 processors. For each benchmark, we plot performance—normalized to that of *Snooping* with unbounded bandwidth—as a function of available interconnect endpoint bandwidth. We also include the results of our microbenchmark to allow for direct comparison. Our results show that, for a 16 processor system and a range of bandwidths, *Snooping* and *BASH* perform similarly, and both outperfom *Directory*. The macrobenchmark results look qualitatively similar to the microbenchmark, but the performance difference between *Snooping* and *Directory* is smaller for some of the benchmarks. This is due to a lower cache miss rate (Barnes and Slashcode) or a smaller fraction of sharing misses (SPECjbb).

To approximate *BASH*'s performance on a larger system, we increase the cost of broadcast by quadrupling the interconnect bandwidth used by any broadcast request. Figure 11 presents these results and shows that, for a range of bandwidths, *BASH* performs as well or better than both *Snooping* and *Directory*. We did not perform any macrobenchmark simulations with less than 600 MB/second endpoint bandwidth due to excessive simulation times. However, we expect the performance of *BASH* to closely track that of *Directory*, as was the case for the microbenchmark on 64 processors (shown in Figure 1).

While these results show that *BASH* can adapt to system configuration, one of *BASH*'s strengths is adaptation to varying behaviors between workloads. In Figure 12, we plot the 1600 MB/second data excerpted from Figure 11 normalized to the performance of *BASH*. For this configuration, *Snooping* outperforms *Directory* for Barnes-Hut and OLTP, but the reverse is true for SPECjbb. The performances of Slashcode and Apache are similar for *Snooping* and *Directory*. For this configuration, *BASH* matches or exceeds the performances of both other protocols for all five workloads.



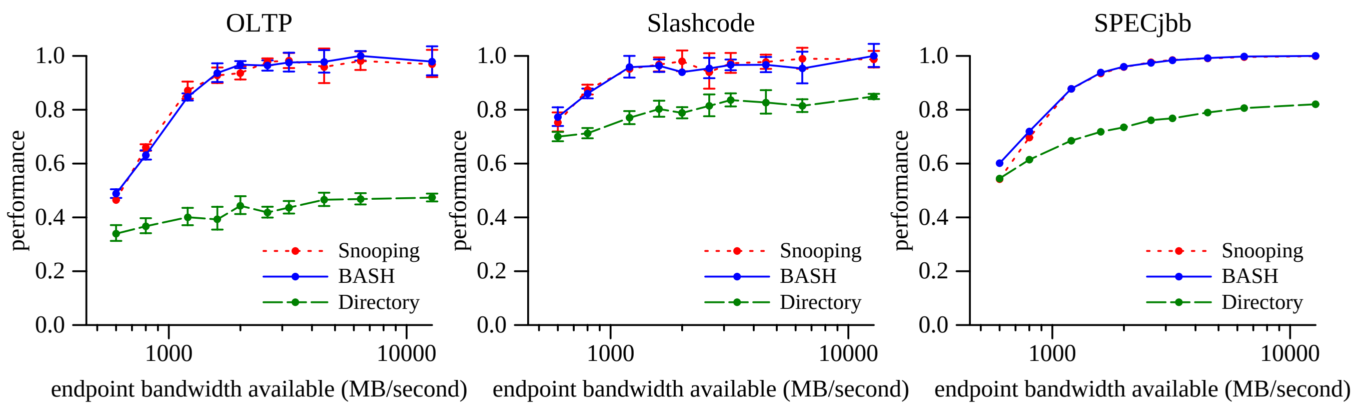
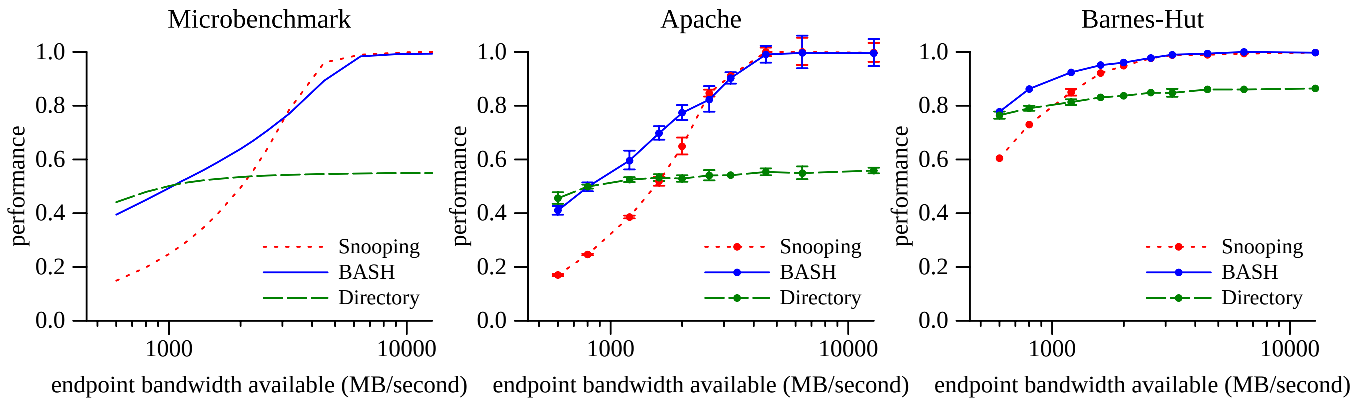


Figure 10. Performance vs. available bandwidth for 16 processors



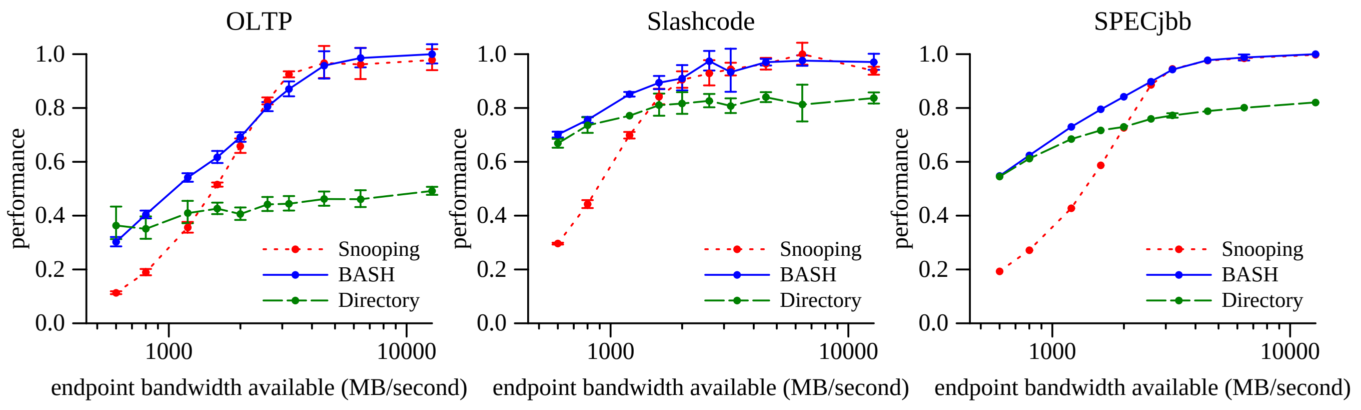


Figure 11. Performance vs. available bandwidth for 16 processors with 4x broadcast cost

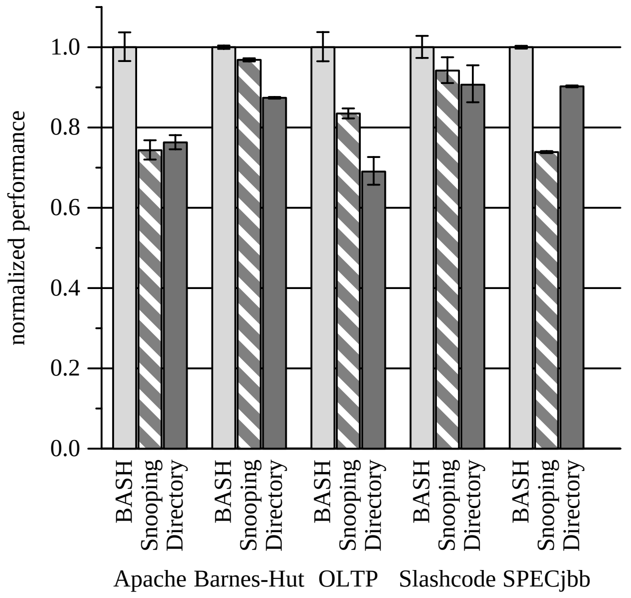


Figure 12. Adapting to workload intensity

Though increasing the relative cost of broadcasting does not capture other effects of increasing system sizes such as lock contention and changes to sharing patterns, we believe these results show that bandwidth adaptive coherence in general, and *BASH* in particular, enables robust system performance for a wide range of system configurations and workloads.

6 Related Work

Prior related research falls roughly into the two categories of protocols and networks.

Protocols. There is a great deal of research in protocols that adapt towards sharing patterns rather than network usage. Multicast Snooping protocols [4, 28] allow processors to multicast requests to those nodes that are suspected to need to observe the request, and the multicast mask is predicted based on observed sharing patterns. This differs from bandwidth adaptive snooping in that *BASH* only predicts two types of masks (unicast or broadcast) and chooses its multicast mask based on available bandwidth. Competitive snoopy caching adapts between an invalidate and an update protocol [16] to limit the overhead to within a factor of two of optimal. Additional research has pursued the idea of adaptive protocols, but none of which we are aware consider interconnection network utilization. Another class of adaptive protocols includes the COMA [9, 13] and R-NUMA [8] protocols. These protocols migrate data to where it is used, adaptively reducing communication and network traffic.

Networks. Scott and Sohi [27] proposed using network feedback to adaptively avoid tree saturation in multistage interconnection networks. They use feedback control theory to adjust network usage to avoid performance degradation due to hot spots in memory access patterns. This work differs from bandwidth adaptive snooping in that it seeks to throttle requests from being issued, whereas *BASH* chooses between issuing a unicast or a broadcast.

Thottethodi et al. [31] have developed a scheme for using global network information to throttle requests before they can congest the network. At the cost of an additional network sideband for communication of contention effects, this scheme can adapt more accurately than a scheme that relies solely on local information. This work complements bandwidth adaptive snooping in that it could be used to estimate network utilization, and future work may adapt these techniques to improve our detection of network congestion. Other research has also explored techniques for estimating interconnect traffic (refer to the related work in [31]).

7 Conclusions and Future Work

We have developed a hybrid shared memory cache coherence protocol, and we have demonstrated the benefits provided by adaptivity. Moreover, the trend towards integration of the coherence protocol logic and the processor on the same die suggests a unified adaptive design. An adaptive approach allows a single highly integrated microprocessor design to be used in many system configurations (e.g., number of processors). Also, adaptivity allows the system to adjust to various workloads, including future workloads whose behaviors are unknown at hardware design time.

One area of future work is the exploration of additional mechanisms for deciding whether to unicast or broadcast. Particularly in the middle range of bandwidth where a decision based on available bandwidth is less obvious, it might be preferable to predict based on sharing patterns. There are many instances where the decision would be easy, such as the choice to unicast requests for misses due to instruction fetches. Moreover, integrating bandwidth adaptivity with multicast snooping [4]—rather than simply unicasting or broadcasting—might be worthwhile. Additionally, more sophisticated adaptive mechanisms, perhaps using global estimates of interconnection network utilization [31], could be employed.

Acknowledgments

We thank Virtutech AB for their support of Simics and the Condor group and Remzi Arpaci-Dusseau for providing additional computing resources. Alaa Alameldeen, Pacia Harper, and Min Xu contributed to the setup and tuning of our workloads. We also thank Adam Butts, Kourosh Gharachorloo, Anders Landin, Alvin Lebeck, Ravi Rajwar, Amir Roth, Craig Zilles and the Wisconsin Computer Architecture Affiliates for their comments on this work.

References

[1] E. Artiaga, N. Navarro, X. Martorell, and Y. Becerra. Implementing PARMACS Macros for Shared Memory Multiprocessor Environments. Technical report, Polytechnic University of Catalunya, Department of Computer Architecture Technical Report UPC-DAC-1997-07, Jan. 1997.

[2] P. Barford and M. Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *Proceedings of the 1998 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 151–160, June 1998.

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

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

[5] J. Borkenhagen and S. Storino. 4th Generation 64-bit PowerPC-Compatible Commercial Processor Design. IBM Server Group Whitepaper, Jan. 1999.

[6] A. Charlesworth. Starfire: Extending the SMP Envelope. *IEEE Micro*, 18(1):39–49, Jan/Feb 1998.

[7] A. Charlesworth. The Sun Fireplane Interconnect. In *Proceedings of SC2001*, Nov. 2001.

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

[9] S. J. Frank. Tightly Coupled Multiprocessor System Speeds Memory-access Times. *Electronics*, 57(1):164–169, Jan. 1984.

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

[11] S. W. Golumb. *Shift Register Sequences*. Aegean Park Press, revised edition, 1982.

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

[13] E. Hagersten, A. Landin, and S. Haridi. DDM–A Cache-Only Memory Architecture. *IEEE Computer*, 25(9):44–54, Sept. 1992.

[14] J. Hennessy. The Future of Systems Research. *IEEE Computer*, 32(8):27–33, Aug. 1999.

[15] N. Izuta, T. Watabe, T. Shimizu, and T. Ichihashi. Overview of PRIMEPOWER 2000/1000/800 Hardware. *Fujitsu Scientific & Technical Journal*, 36(2):121–127, Dec. 2000.

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

[17] S. Kunkel. Personal Communication, Apr. 2000.

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

[19] B. C. Kuo. *Automatic Control Systems*. Prentice Hall, seventh edition, 1995.

[20] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 241–251, June 1997.

[21] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63–79, Mar. 1992.

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

[23] M. M. K. Martin, D. J. Sorin, A. Ailamaki, A. Alameldeen, R. M. Dickson, C. J. Mauer, K. E. Moore, M. Plakal, M. D. Hill, and D. A. Wood. Timestamp Snooping: An Approach for Extending SMPs. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 25–36, Nov. 2000.

[24] J. R. Mashey. NUMAflex Modular Design Approach: A Revolution in Evolution. Posted on comp.arch news group, Aug. 2000.

[25] A. Nanda, K.-K. Mak, K. Sugavanam, R. K. Sahoo, V. Soundararajan, and T. B. Smith. MemorIES: A Programmable, Real-Time Hardware Emulation Tool for Multiprocessor Server Design. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.

[26] M. Plakal, D. J. Sorin, A. E. Condon, and M. D. Hill. Lamport Clocks: Verifying a Directory Cache-Coherence Protocol. In *Proceedings of the Tenth ACM Symposium on Parallel Algorithms and Architectures*, pages 67–76, June 1998.

[27] S. Scott and G. Sohi. The Use of Feedback in Multiprocessors and its Application to Tree Saturation Control. *IEEE Transactions on Parallel and Distributed Systems*, 1(4):385–398, Oct. 1990.

[28] D. J. Sorin, M. Plakal, M. D. Hill, A. E. Condon, M. M. Martin, and D. A. Wood. Specifying and Verifying a Broadcast and a Multicast Snooping Cache Coherence Protocol. Technical Report 1412, Computer Sciences Department, University of Wisconsin–Madison, Mar. 2000.

[29] P. Sweazey and A. J. 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.

[30] J. M. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. POWER4 System Microarchitecture. IBM Server Group Whitepaper, Oct. 2001.

[31] M. Thottethodi, A. R. Lebeck, and S. S. Mukherjee. Self-Tuned Congestion Control for Multiprocessor Networks. In *Proceedings of the Seventh IEEE Symposium on High-Performance Computer Architecture*, Jan. 2001.

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

[33] D. A. Wood, G. A. Gibson, and R. H. Katz. Verifying a Multiprocessor Cache Controller Using Random Test Generation. *IEEE Design and Test of Computers*, Aug. 1990.