Appears in the proceedings of the
Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX),
Cambridge, Massachusetts, November 13-15, 2000.

Timestamp Snooping: An Approach for Extending SMPs

Milo M. K. Martin, Daniel J. Sorin, Anastassia Ailamaki, Alaa R. Alameldeen, Ross M. Dickson,
Carl J. Mauer, Kevin E. Moore, Manoj Plakal, Mark D. Hill, David A. Wood
Computer Sciences Department
University of Wisconsin-Madison
`http://www.cs.wisc.edu/multifacet`

Abstract

Symmetric multiprocessor (SMP) servers provide superior performance for the commercial workloads that dominate the Internet. Our simulation results show that over one-third of cache misses by these applications result in cache-to-cache transfers, where the data is found in another processor's cache rather than in memory. SMPs are optimized for this case by using snooping protocols that broadcast address transactions to all processors. Conversely, directory-based shared-memory systems must indirectly locate the owner and sharers through a directory, resulting in larger average miss latencies.

This paper proposes timestamp snooping, a technique that allows SMPs to i) utilize high-speed switched interconnection networks and ii) exploit physical locality by delivering address transactions to processors and memories without regard to order. Traditional snooping requires physical ordering of transactions. Timestamp snooping works by processing address transactions in a logical order. Logical time is maintained by adding a few bits per address transaction and having network switches perform a handshake to ensure on-time delivery. Processors and memories then reorder transactions based on their timestamps to establish a total order.

We evaluate timestamp snooping with commercial workloads on a 16-processor SPARC system using the Simics full-system simulator. We simulate both an indirect (butterfly) and a direct (torus) network design. For OLTP, DSS, web serving, web searching, and one scientific application, timestamp snooping with the butterfly network runs 6-28% faster than directories, at a cost of 13-43% more link traffic. Similarly, with the torus network, timestamp snooping runs 6-29% faster for 17-37% more link traffic. Thus, timestamp snooping is worth considering when buying more interconnect bandwidth is easier than reducing interconnect latency.

1 Introduction

This paper is concerned with the design of shared-memory multiprocessors for commercial workloads. These systems provide an important computational infrastructure for the Internet. These workloads have two key characteristics. First, there is diminished incentive to dramatically increase the number of processors in a multiprocessor. While some scientific workloads can use 1000s of processors, most multiprocessors for commercial workloads—which are the bulk of total multiprocessor sales—are smaller systems: 2-4 processors first, 4-8 next, then 16-32, and finally 64 and up. Service providers who need even more throughput often create clusters, since they also desire availability and know that little commercial software runs on multiprocessors with 100s of processors. As Gregory Papadopoulos says [34], "*Multiprocessors can't just scale up. They have to scale down, too.*" Many computer vendors have responded to this market reality by optimizing for small systems first. Intel, for example, concentrates on four processor "quad" systems. Eight processor systems integrate two quads together [43]. Still larger systems cluster multiple quads into a distributed shared-memory system [29].

The second key characteristic of commercial workloads is that data sharing is common, i.e., they frequently suffer cache misses that must obtain data from other processors' caches. These misses have been called *cache-to-cache transfers*, *dirty misses*, and (for directory protocols) *three-hop transactions*. For an online transaction processing (OLTP) commercial workload, Barroso et al. [8] report that cache-to-cache transfers account for 55-62% of misses and state that tradeoffs that penalize cache-to-cache transfers "will have a severe impact on OLTP performance." Moreover, since cache-to-cache transfers are due to true sharing, false sharing, and migratory data, they cannot be reduced by increasing the cache sizes. Instead, because larger caches have fewer capacity and conflicts misses, the fraction of cache-to-cache transfers and their impact on performance actually increases [24].

A key way in which many computer vendors optimize cache-to-cache transfers is by supporting *snooping* cache coherence. Systems using snooping—called *symmetric multiprocessors (SMPs)*—broadcast each coherence transaction on a "bus" that implicitly places transactions into a total order and synchronously delivers them to memories and caches. Low-latency broadcast ensures that the owner

This work is supported in part by the National Science Foundation with grants MIP-9625558, EIA-9971256, and CDA-9623632, two Wisconsin Romnes Fellowships, and donations from Compaq Computer Corporation, Intel Corporation, and Sun Microsystems. Milo Martin is supported by an IBM Graduate Fellowship. Anastassia Ailamaki is supported by the Anthony C. Klug NCR Fellowship in Databases.

ASPLOS 2000 Cambridge, MA Nov. 12-15 , 2000

of a block can respond to a coherence request as quickly as possible. Recently, some vendors have begun to further optimize cache-to-cache transfers by carefully tuning the implementation path for sourcing data from one cache to another. The IBM NorthStar (RS64-II) system [11], for example, has a cache-to-cache transfer latency (43 cycles) that is 55% of main memory latency (78 cycles) [23].

More than a decade ago, Agarwal et al. [2] predicted that limited bus bandwidth would lead to the demise of SMPs. They argued that directory-based coherence protocols provide better scalability by sending memory transactions over a point-to-point network to a directory (usually at memory) that redirects the request either to memory (trivial) or to other processors. Yet, to date, directories have failed to dominate SMPs for two reasons. First, as noted above, scalability is much less important for most commercial servers. Second, directories achieve their scalability by adding a level of indirection to handle cache-to-cache transfers, significantly increasing average miss latency. For example, a cache-to-cache transfer takes 1036 ns on the SGI Origin 2000 [26] compared to 742 ns on the Sun UE10000 [12], despite a comparable memory access time [21].

Instead of fading away, SMPs have continued to dominate the multiprocessor market, largely due to a wealth of techniques for building high-bandwidth, low-latency "buses" that deliver transactions in an implicit, physical total order. These techniques include split-transaction buses, physically separate interconnects for address requests and data responses [38], bus-bridging with filtering [43], multiple request buses interleaved by address [12], multiple hierarchical buses [11], and moving from physical shared wire buses to logical buses implemented as bit-sliced pipelined broadcast trees with point-to-point links [12]. To match this bus bandwidth, SMP designers have increased snoop bandwidth by duplicating cache tags, using multiple banks, and exploiting the higher bandwidth of on-chip cache controllers and tags.

However, two new issues threaten the future of SMPs. The first issue is the increasing cost (and opportunity cost) of building address networks that perform *ordered broadcasts*, i.e., deliver snooping coherence transactions to all processors in a total order. Increasing levels of integration are encouraging some designers to directly connect processors, eliminating the cost and delay of glue chips. The Compaq 21364 [19], for example, can be connected in a torus, which does not lend itself to delivering broadcasts in a total order. Designs that use hierarchy also have trouble exploiting near neighbor communication, because delivering transactions in order requires all transactions to pass through the root(s) of the hierarchy. An example of a directly connected system with significant natural hierarchy is IBM's Power4 [13], which contains two processors per chip, four chips per multi-chip module (MCM), and multiple MCMs per board, with a memory controller for each chip.

The second issue threatening the future of SMPs is that delivering each transaction to all caches and memories at the same time—*synchronous broadcast*—is a poor match for emerging interconnection options. At an architectural level, designers may wish to conserve resources by having coherence transactions and data dynamically share the same physical network links. Maintaining synchronous broadcast requires that switches carefully coordinate their routing decisions, thereby limiting feasible topologies, routing schemes, and buffering strategies. At an implementation level, most current SMPs use back-pressure flow-control to throttle the address network when end-point contention causes queues to fill up. With hierarchical designs, longer relative latencies, and high-bandwidth source synchronous links [20], maintaining synchronous broadcast may require a global, rather than simply local, flow control mechanism.

Will problems with ordered and synchronous broadcast doom SMPs? We think not, because snooping protocols depend on processing coherence transactions in a total order, but they need not require that the network deliver the transactions in that order or at the same time. To this end we propose *timestamp snooping*, which operates as follows. A processor node generates a coherence transaction and submits it to the network. The network assigns the transaction a logical timestamp and then broadcasts it to all processor and memory nodes without regard for order. Finally, processor and memory nodes process coherence transactions in the same logical order, but not necessarily at the same physical time. While other write-through coherence proposals have used logical timestamps [10, 44], timestamp snooping applies them to ubiquitous writeback MOESI protocols (see Section 6).

Qualitatively, the benefit of timestamp snooping is that it has the potential to outperform traditional SMPs by exploiting the natural hierarchy of high-bandwidth switched networks and delivering address transactions to each node via the shortest path. Timestamp snooping can outperform moderate-scale directory systems by eliminating three-hop cache-to-cache transfers. Of course, directories will always provide superior performance for large-scale systems because of the greater bandwidth requirements of snooping.

There are two primary challenges to implementing timestamp snooping. First, the network must be augmented to maintain logical time and deliver transactions by their logical time deadlines. Section 2 provides an overview of the technique and presents one possible implementation based

on token passing. Second, the coherence protocol must be extended to eliminate the need for synchronous broadcasts. Section 3 explains how to accomplish this by adding a state bit in memory to indicate whether memory owns a block (as in Synapse [15]).

Section 4 discusses our evaluation methodology that includes full system simulation of a 16-processor SPARC system running commercial workloads. Section 5 presents quantitative results that show that timestamp snooping is faster than (6-29%) two directory protocols at a cost of using more interconnection network bandwidth (13-43%). Finally, Section 6 discusses related work and Section 7 concludes.

This paper make two contributions. First, it presents timestamp snooping as a design for implementing MOESI snooping on switched networks. Second, it shows how, relative to directories, timestamp snooping reduces execution time by using more interconnection network bandwidth.

2 Timestamp Snooping Networks

In this section, we describe how to build an interconnection network that delivers address transactions with an explicit logical order, rather than delivering them in an implicit physical order. The network is the sole method of communication between cache and memory controllers. This network proposal is independent of topology and routing policy, and it adds relatively little hardware to existing network switches.

Our snooping system uses a broadcast address network and a logically separate data network. The data network must reliably deliver data messages to a single destination, but it can do so without regard for order. It can—and we assume it will—be implemented in the same physical network as the address network. By providing separate virtual networks for address requests and data responses, deadlocks are easily avoided [14]. Since the data network design is conceptually straightforward, we will not discuss it further.

2.1 Basic Idea

Our address network creates snooping's total order by adding a logical timestamp to each transaction. Transactions are delivered as quickly as possible without regard to order. Processors and memories receive transactions out of order, put them back into the global order using the logical timestamps, and process transactions only after receiving a guarantee that no earlier transaction will arrive. This approach relies on two types of logical times:

- *Ordering time (OT)* is the logical ordering time of an address transaction. The OTs of all transactions define a total order. Snooping protocols will use this total order for processing transactions.
- *Guarantee time (GT)* for a network switch or processor interface is a logical time that is guaranteed to be less than the OTs of any transactions that may later be received. A processor or memory can safely process a transaction whose OT $\leq$ GT, because no logically earlier transactions may be in-flight.

Processing proceeds in four steps:

- **Assign OT:** When a processor injects a transaction into the network, the processor assigns an OT to the transaction. This OT is at least the source's GT plus the logical time to get from the source to the furthest destination (e.g., the number of hops between them).
- **Broadcast:** The network may use any deadlock-free routing algorithm to broadcast transactions to all nodes in any order.
- **Compute GTs:** Network switches exchange information to coordinate when it is safe to update their GTs. Switches may stall GT updates to ensure that in-flight transactions are delivered to the next routing hop with enough time remaining to reach their final destination at or before their OT.
- **Destination Operation:** Processors and memories receive transactions out of OT order, put them back into order, and (logically) process a transaction only when no earlier transaction could still arrive (i.e., when a transaction's OT is less than or equal to the GT of the end-point).

To the first order, this network will have the same performance as an unordered broadcast network, because the GT calculation only affects whether the end-point can process a transaction, not whether the network can deliver it.

2.2 One Network Implementation

This section describes *one specific way* to implement the abstract ordered network of the last section. In this implementation, OTs and GTs are maintained implicitly. A transaction carries only an explicit *slack* term, which is the extra amount of logical time that the transaction can use to get to its destination (in addition to the required logical time to get from its current location to its destination). Slack is initially set by the source, and it is modified by switches while in-transit, as explained below. From the slack, a switch can determine the OT. Similarly, GTs are implicitly maintained using token passing, as in earlier ordered network proposals [32, 37].

Each processor/memory node connects to one network switch. Switches are connected to each other in some topology, and our scheme can be applied to many network topologies. Each node and switch begin operation with one (or more) tokens on each input port.

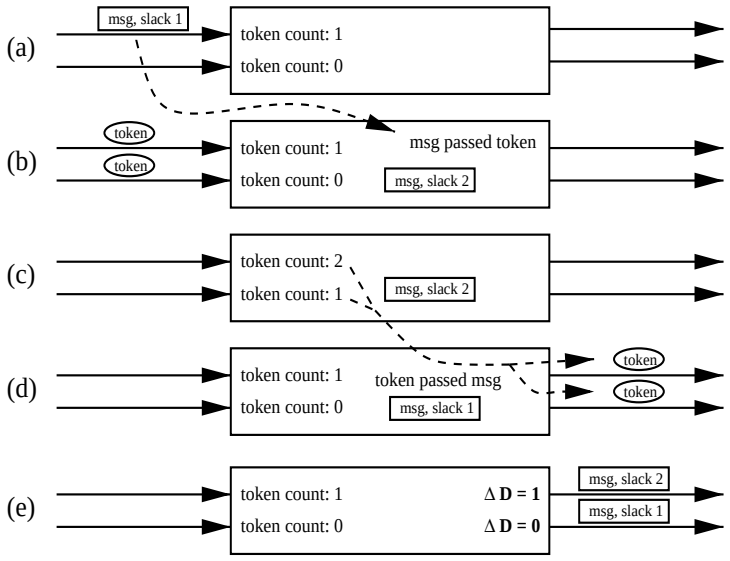

Figure 1. Token Passing Example

This figure illustrates a token passing example with a simplified 2x2 switch, where (physical) time flows from top to bottom. Initially, in (a), the switch has an empty buffer and an incoming message, *msg*, with slack equal to one. In (b), the switch handles this message, and we assume that contention forces this message to be buffered. As the message moves past the token counter, the switch buffers the message and increments its slack to equal 2 (ΔGT=1). In (c), the switch processes the incoming tokens by incrementing the token counters. Then, in (d), it can issue a token on each output. When the token is issued on the outputs, it moves past the message in the buffer and causes its slack to decrease to one (ΔGT=-1). Finally, in (e), we assume that the contention is removed and the switch can issue the message on its output links, with the slack modified on each link to reflect the change in distance to the furthest destination on each link (note that ΔD=1 on the top link).

Source Node Operation. When a node generates a transaction, it includes a non-negative *slack S* in its payload. This action implicitly sets the transaction's OT to:

$$OT_{transaction} = GT_{source} + D_{max} + S$$

where GT_{source} is the GT of the transaction's source, D_{max} is the network pipeline delay (e.g., network hops) to the furthest destination, and S is the slack. Setting S to a small positive value allows GTs to advance during moderate network contention without unduly delaying destination processing. The source node then passes the transaction to the adjacent switch. A node implicitly maintains its GT in the same manner as will be discussed for switches.

Switch Operation. Switches exchange tokens to maintain their GTs and to ensure that all transactions are delivered at or before their OTs. Each switch logically maintains a token counter per input port and contains a logically centralized transaction buffer. Intuitively, the GT of a switch is the number of tokens it has propagated. A switch may propagate a token whenever:

- It has received a token from each input (i.e., the token counters of all incoming links are non-zero), and
- All buffered transactions have non-zero slack.

When a switch propagates a token it:

- Sends a token on each of its outputs,
- Decrements the slack of all buffered transactions, and
- Decrements the token counter for each input.

When a switch receives a transaction, it must forward it to one or more output ports. We assume a statically balanced broadcast routing algorithm using minimum distance spanning trees implemented with a table lookup on transaction source ID.

There are three circumstances under which a switch modifies the slack of an in-flight transaction to ensure that a transaction's OT is invariant as it is delivered to all destinations. In all cases, it is calculated with the recurrence:

$$S_{new} = S_{old} + \Delta GT + \Delta D$$

where S_{old} is the previous slack, ΔGT is the difference in GT, and ΔD is the magnitude of the *decrease* in maximum pipeline depth for a branch of the broadcast (to allow for unbalanced broadcast trees). All switch operations must further satisfy the invariant that $S_{new} \geq 0$.

First, when a transaction enters a switch, ΔGT equals the number of tokens it moves past (i.e., the value of the token counter on its input port). In effect, moving past a token makes a transaction earlier in logical time; thus slack must be increased to hold OT invariant. Second, when a switch propagates a token that moves past a buffered transaction, the transaction effectively becomes later (closer to its OT) in logical time, requiring that its slack be decremented ($\Delta GT = -1$). Note that the invariant of having $S_{new} \geq 0$ prohibits tokens from moving past zero-slack transactions. Third, when a switch sends a transaction out, a ΔD is obtained for each outgoing branch in the same lookup that selects output ports. ΔD is zero for the branch that continues on the longest path, and it is greater for shorter paths. Figure 1 provides an example illustrating how switches exchange tokens and transactions, and it covers the three cases of the recurrence. Note that, in practice, a token can be encoded in one or two bits piggybacked on a transaction or a null message.

The switch is standard [14, Chapter 2] except for the token passing logic, which operates in parallel with normal message routing. While the token passing logic may delay GTs

from advancing, it never delays transaction delivery. The additional hardware for token passing includes i) a read-only table to obtain ΔD (easily combined with the routing table), and ii) logic that detects zero-slack transactions and decrements the slack of buffered transactions when a token is sent to each output. Including zero-detect logic and a small decrementor (e.g., 4-6 bits) on a modest number of buffers (e.g., 64) is neither difficult nor expensive. Finally, the arbitration logic gives precedence to zero-slack transactions, to speed token passing.

Destination Operation. Processors and memories will frequently receive transactions with positive slack, implying $OT_{transaction} > GT_{destination}$. To recreate the total order, each endpoint must sort the transactions and delay processing until it is sure it will not receive a transaction with an earlier OT. Moreover, all endpoints must, in the same way, fairly order transactions that have the same OT. This is easily done by breaking ties with a function of source ID numbers.

We implement destination processing with an augmented priority queue.[1] When a transaction arrives, it is inserted into the priority queue with its current slack (and source ID tie-breaker). When a destination gets a token from its adjacent switch, it processes all slack-zero transactions, decrements the slack of still enqueued transactions, and sends a token to its switch.

Buffering. To avoid deadlock, network switches and endpoints must provide enough buffering to ensure that *early* transactions (S>0) cannot block *on-time* transactions (S=0). A sufficient solution adds no special buffering to switches but requires endpoints to provide worst-case buffering (e.g., 128 address buffers per endpoint in a system that allows 8 outstanding transactions from each of 16 processors). We are investigating more parsimonious solutions.

3 Timestamp Snooping Protocols

Conventional write-invalidate snooping protocols maintain a subset of the MOESI stable states—M (Modified), O (Owned), E (Exclusive), S (Shared), and I (Invalid) [40]—in response to transactions delivered in order (ordered broadcast) and at the same time (synchronous broadcast). Many implementations require processors in M or O to assert an *owned* signal that is logically OR-ed to inform memory not to respond. Similarly, processors in S or E can assert a *shared* signal to prevent a processor seeking an S copy from entering the E state.

Timestamp snooping protocols can also support any subset of the MOESI states in response to transactions that have been delivered quickly but restored to a logical order. Implementing owned and shared signals, however, is difficult since transactions can arrive at different times. For this reason, we recommend eliminating the *owned* signal with the old Synapse scheme of adding one bit per block to indicate if memory is the owner [15]. This can be implemented with minimal memory overhead (0.2% for 64-byte blocks) or without additional memory bits by changing the error correcting code [7, 18, 27, 33]. This change will increase memory controller occupancy, however, by turning some memory reads into read-modify-writes. We can eliminate the *shared* signal by adding a second bit in memory or by forgoing the E state optimization.

At least two groups have formally shown that snooping depends on the order but not on the time transactions are processed [1, 39]. Thus, timestamp snooping correctly implements coherence and allows processors to implement any memory consistency model. Sorin et al. [39] present a detailed specification of a broadcast snooping protocol that could operate with a timestamp snooping network.

Finally, the timestamp snooping network permits two optimizations not possible with conventional snooping. These involve ‘peeking’ at a transaction that has been delivered but not yet safely put in the logical order. First, memory and cache controllers can prefetch data from DRAM and SRAM, respectively, provided they do not send the data until the logical order is confirmed. Second, processors can process other processors’ early transactions to blocks currently in stable states S, I, or not present. The first optimization hides timestamp snooping’s worst-case broadcast delay, while the second can significantly reduce average queueing delay and buffer utilization.

4 Performance Evaluation Methods

This section describes our benchmarks, target system assumptions, and simulation techniques for evaluating timestamp snooping. We concentrate on comparing timestamp snooping versus a directory protocol using full system simulation of a 16-processor SPARC system running commercial workloads. We assume two example network topologies based on point-to-point links and no requirement for global synchrony. We do not compare against a fully-synchronous SMP, because, as we argued in the introduction, it is not clear how to build such a machine at the high bandwidths required for future systems.

1. Recall that a priority queue is a data structure that permits items to be inserted, can respond with the value of the minimum item, and can remove the minimum item. Priority queues are widely used in routers and can be implemented with constant time operations using linear space [28, 31].

Table 1. Benchmark Descriptions

Online Transaction Processing (OLTP): DB2 with a TPC-C-like workload. The OLTP workload is based on the TPC-C benchmark [41] on IBM's DB2 v6.1 database management system. TPC-C portrays the activity of a wholesale supplier, with many concurrent users executing read/write transactions against the database. The TPC-C implementation in our experiments uses the IBM benchmark kit to build the database and model users who execute transactions with no keying or think time. We modified the driving scripts to execute a set number of transactions per user. Our experiments use a memory-resident 400 MB, 4-warehouse database and executes 10 transactions for each of 16 concurrent users.
Decision Support Systems (DSS): DB2 with TPC-H-like workload. The DSS workload is modeled by a single run of query 12 from the TPC-H benchmark [42] on IBM's DB2 v6.1. We used the IBM TPC-H benchmark kit to create a memory-resident 100 MB database. TPC-H consists of a suite of business-oriented ad-hoc queries, and each query involves a complex execution plan, which is implemented by a rich collection of collaborating operators. The configuration exploits parallelism amongst operators by enabling intra-query parallelism.
Web server: Apache with SURGE. Web servers, such as Apache [4], have become an important enterprise server application. We used Apache 1.3.9 for SPARC/Solaris 7, and it was driven by SURGE, the Scalable URL Request Generator [6], which analytically generates traffic representative of real-world requests. The SURGE client used 12 threads with zero think time to generate 652 HTTP/1.1 requests for documents chosen from a corpus of 8000 text files (totalling 160MB).
Web search engine: Altavista. Search engines, such as Altavista, are a significant server workload due to their increased deployment on enterprise intranets. We used an evaluation copy of Altavista 2.3A for Solaris [3] to build a 500MB index of nearly 160,000 web pages from over 6000 Internet web servers. The Altavista server uses two threads (each in a separate process) to handle a single query. Our query generation client used 12 request generation threads with zero think time to execute 50 search requests from a pre-constructed request trace. Each request returned an average of 4000 web pages (2.5% of our index size).
Scientific workloads: SPLASH-2. Since scientific workloads are not the focus of this paper, we selected only one application from the SPLASH-2 benchmark suite [45]: *barnes-hut* with 16K bodies. The benchmark was compiled with the POSIX-threads version of the PARMACS shared-memory macros used by Artiaga et al. [5], and we began measurement at the start of the parallel phase to avoid measuring thread forking. The macro library was modified to use user-level synchronization through test-and-set locks rather than POSIX-thread library calls.

4.1 Benchmarks

Table 1 describes the benchmarks we use. We concentrate on commercial applications, such as database workloads, web servers, and search engines, but we also include one scientific application for comparison purposes. All of the workloads were run once for warm-up and then again for measurement. To simplify the simulations of the client-server benchmarks (OLTP, web serving, and web searching), we ran the client and the server on the same machine.

4.2 Target System Assumptions

We evaluate 16-node SPARC systems running an unmodified copy of Solaris 7. Each node contains a processor core, level one caches, a unified level two cache (4 Mbyte, 4-way, 64-byte blocks), a cache controller, and a memory controller for part of the globally shared memory (1 GByte total). We assume that a processor and level one caches would complete four billion instructions per second with a perfect memory system beyond the level one caches. This could be accomplished, for example, with a 1 GHz processor having an IPC of 4 (*instructions per cycle* with a perfect memory system), or a 2 GHz processor with an IPC of 2.

Protocols. We implement timestamp snooping (*TS-Snoop*) and two directory protocols (*DirClassic* and *DirOpt*). All are MSI protocols, 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 copy), and interact with processors to support sequential consistency. TS-Snoop uses one-bit per block in memory. Controllers prefetch from memory or cache as soon as transactions arrive (optimization 1 in Section 3), but controllers do not perform early invalidations (optimization 2).

Both directory protocols use a full bit vector for sharers. DirClassic is modeled after the protocol used in the commercially-deployed SGI Origin 2000 [26]. It assumes unordered virtual networks, and it sometimes nacks (negatively acknowledges) transactions. Recent directory research has sought to reduce or eliminate nacks [7, 18]. To this end, we developed DirOpt, which uses point-to-point ordering on one virtual network to avoid nacks and avoid all blocking at cache and memory controllers.

Networks. We consider integrated processor/memory controller nodes connected in two *example*[2] interconnection topologies that are illustrated in Figure 2: *four indirect radix-4 butterflies* and a *direct 4x4 2D torus.* These network topologies are examples that are not necessarily optimal, but they were selected because they are similar to

2. Recall that our scheme is topology independent.

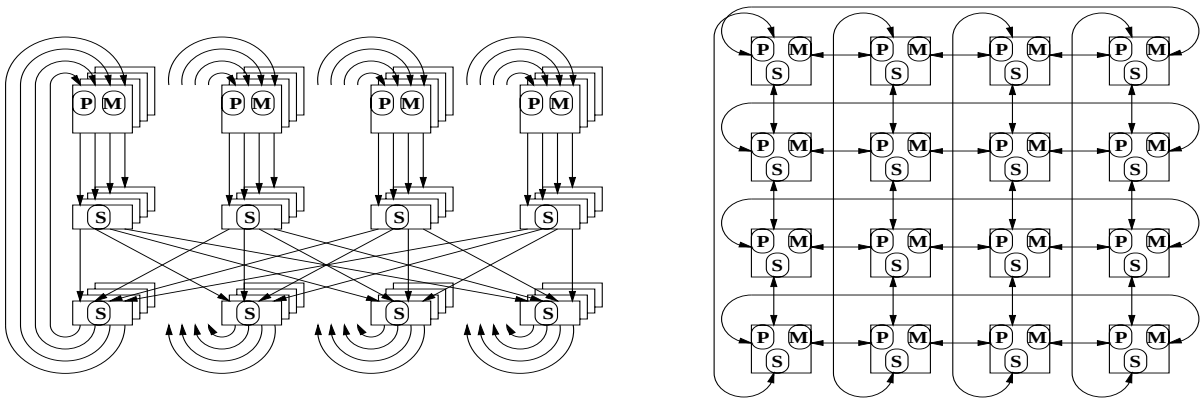

Figure 2. 16-Processor System with Four Radix-4 Butterflies (left) and 4 x 4 Bidirectional Torus (right)

current or future systems interconnects. We selected the butterfly because SMP proposals have used indirect switching nodes to create trees or fat trees to support fast broadcast [9, 12]. A butterfly is like a fat tree that uses separate switches for the "up" and "down" paths, enabling a higher radix implementation (for a fixed number of switch pins) at a cost of not having some shorter paths [14]. We assume four butterflies, selected round-robin, so that processor/memory controller nodes have four outgoing and four incoming point-to-point links (the same as with the torus described next). A 16 processor radix-4 butterfly delivers a message using 3 links and broadcasts a transaction with 3-link latency using 21 links (1+4+16).

We selected a 2D bidirectional torus because it is the proposed network topology of the Compaq Alpha 21364 [19]. Like the Alpha 21364, our network switch is integrated onto the processor die, avoiding any additional switch chips. A torus delivers messages using a mean of 2 links and broadcasts transactions using 15 links with a mean arrival latency of 2 links and worst-case latency of 4 links.

TS-Snoop uses two virtual networks: one for address transactions (that applies the techniques of Section 2) and the other for an unordered data network. The directory protocols use three virtual networks: an unordered request network, a network for requests forwarded by the directory to processors, and an unordered network for responses from processors and directories. The forwarded request virtual network is unordered for DirClassic and point-to-point ordered for DirOpt. All virtual networks use virtual cut through routing, which requires only one virtual channel per virtual network.

System Timing Assumptions. Table 2 gives selected timing assumptions. To approximate the published latencies of the Compaq Alpha 21364 [19], we selected 15 ns for each switch transversal (which includes wire propagation, synchronization, and routing) and 80 ns memory access time.

Table 2. Unloaded Network Timing Assumptions

Description	Value
Assumed latency	
Enter/exit a network (D_{ovh})	4 ns
Switch to switch (D_{switch})	15 ns
Access directory and memory (D_{mem})	80 ns
Access cache (from network) (D_{cache})	25 ns
Computed for indirect radix-4 butterfly	
One way latency ($D_{net} = D_{ovh} + 3*D_{switch}$)	49 ns
Block from memory ($D_{net} + D_{mem} + D_{net}$)	178 ns
Block from cache with timestamp snooping ($D_{net} + D_{cache} + D_{net}$)	123 ns
Block from cache with directory "3 hops" ($D_{net} + D_{mem} + D_{net} + D_{cache} + D_{net}$)	252 ns
Computed for direct 4x4 torus (means)	
One way latency ($D_{net} = D_{ovh} + [0,4]*D_{switch}$ with mean $D_{ovh} + 2*D_{switch}$)	34 ns
Block from memory ($D_{net} + D_{mem} + D_{net}$)	148 ns
Block from cache with timestamp snooping ($D_{net} + D_{cache} + D_{net}$)	93 ns
Block from cache with directory "3 hops" ($D_{net} + D_{mem} + D_{net} + D_{cache} + D_{net}$)	207 ns

When a protocol message arrives at a cache or memory it takes 25 ns or 80 ns, respectively, to provide data to the network. With timestamp snooping, cache or memory accesses may not complete until the protocol message is ordered (see Section 3).

Notice that, for snooping, the cache-to-cache transfer latency is smaller than memory latency (e.g., 70% of memory latency on the butterfly: 123 ns vs.178 ns). We assume

that this case is carefully optimized as is the case for the IBM NorthStar (RS64-II) SMPs [11] where a cache-to-cache transfer latency is 55% of main memory latency [23]. The cache-to-cache transfer latency for the directory protocols is significantly higher than a fetch from memory, due to indirection through memory and thus incurring the latency of the directory access, supplying the data from the cache, and three network messages. Combining these two effects, timestamp snooping has a cache-to-cache miss latency that is roughly half that of the directory protocols with either network.

4.3 Simulation Methods

We simulated our target systems with the Simics full-system multiprocessor functional simulator [30], and we extended Simics with a memory hierarchy simulator to compute execution times.

Simics. Simics is a system-level architectural simulator developed by Virtutech AB that is capable of booting unmodified commercial operating systems and running arbitrary unmodified applications. We are using Simics/sun4u, which can simulate Sun Microsystems's SPARC v9 platform architecture (e.g., used for Sun E6000s) in sufficient detail to boot an unmodified copy of Sun Solaris 7. Simics 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 detailed memory hierarchy simulation. Using full system simulation allows us to avoid trace-based simulations that fail to capture the timing-sensitive and data dependent nature of cache coherence operations.

Processor Model. We use Simics to approximate a processor core and level one caches that execute 4 billion instructions per second and generate blocking requests to the level two data cache. We use this simple processor model to enable tractable simulation times for full-system simulation of commercial workloads. We elected to obtain approximate numbers for the mostly commercial workloads rather than more precise numbers for small scientific workload benchmarks. This simplification is also supported by the existence of current high performance SMPs that use statically scheduled processors [11, 27] and thus limit the number of parallel misses.

Memory System. We have implemented a memory hierarchy simulator that supports all of our cache coherence protocols. It captures all state transitions (including transient states) of our coherence protocols in level two caches and memory. We accurately model unloaded network latencies, timestamp snooping ordering delays, but we do not model network contention. Thus, our results will be meaningful only when network bandwidth is sufficient to keep network contention low. We validated our memory system simulator by comparing simulation results against measured results from the hardware counters on a Sun E6000 and by simulating microbenchmarks with known results.

Table 3. Benchmark Characteristics

Benchmark	Total Data Touched	Total Misses	3-Hop Misses
DB2/TPC-C	47.1 MB	5.3 M	43%
DB2/TPC-H Q12	8.7 MB	1.7 M	60%
Apache/SURGE	13.3 MB	2.3 M	40%
Altavista	15.3 MB	2.4 M	40%
Barnes	4.0 MB	1.0 M	43%

Stability of Results. Multiprocessor performance is sensitive to subtle timing issues. Since full-system simulation captures kernel behavior and inter-processor timing, minor changes in timing can lead to significant variation in run time. To overcome observed instabilities, we performed redundant simulations perturbed by injecting small random delays in all message responses. Among a set of perturbed simulations, most results were near the minimum, while a small minority of outliers were the victims of unfortunate timing races. Thus, when presenting the results in Section 5, we report the minimum run time from a set of runs whose only difference is the perturbation.

In the DSS workload, we observed low-activity startup and end transients with highly variable lengths. When measuring total runtime, these transients often obscured the length of the active segment of the query. To compensate for this effect, we report the measurements from only the contiguous active segment of the query simulation.

5 Performance Evaluation

This section evaluates timestamp snooping against directories, with two example networks of 16 processors, by examining benchmark characteristics, execution times, and network traffic.

Benchmark Characteristics. Table 3 displays results that partially characterize the execution of our benchmarks. Column 1 gives the benchmark name. Column 2 gives the total data memory touched. The relatively small size of data touched makes our 4MB level two caches behave more like infinite caches where cache-to-cache misses dominate capacity and conflict misses. This situation may be representative of future systems where designers use larger caches and other techniques to minimize capacity and conflict misses. Columns 3 and 4 give the total number of misses on all processors and the percent of misses that are cache-to-cache transfers, respectively, from an average of the runs described next.

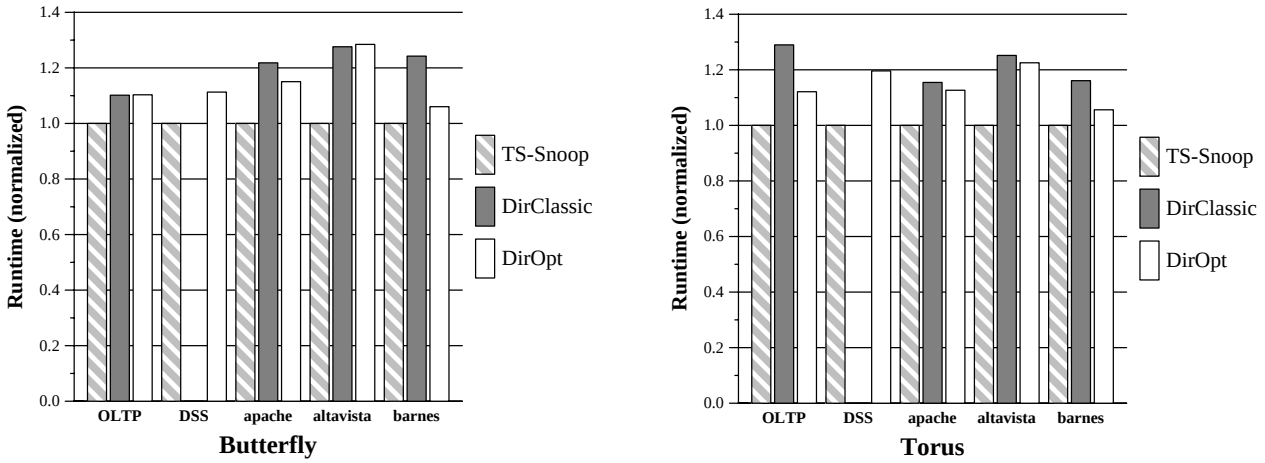

Figure 3. Normalized Runtime with Butterfly (left) and Torus (right)

Execution Times. Figure 3 gives benchmark runtimes (smaller is better) with the butterfly (left) and torus (right)[3] for timestamp snooping and directories, normalized to the runtime of TS-Snoop. Timestamp snooping substantially improves runtime over the range of workloads. On the butterfly, TS-Snoop runs 10-28% and 6-28% faster than DirClassic and DirOpt, respectively.[4] On the torus, TS-Snoop is 15-29% and 6-23% faster than DirClassic and DirOpt, respectively. We omit DSS results with DirClassic in Figures 3 and 4, because runtimes were more than twice as long as those of the other two protocols, due, in part, to a large number of nacks.

This improvement occurs because of the high fraction of cache-to-cache misses (see Table 3 column 4) and the fact that TS-Snoop handles uncontended cache-to-cache misses in about half the time of a directory. TS-Snoop is faster for cache-to-cache misses because it avoids both a directory lookup and a third hop through the network that are both incurred by a directory protocol. Recall that the unloaded network latency assumptions in Table 2 for the butterfly found that cache-to-cache transfers take 123 ns for timestamp snooping and 252 ns for directories, while both took 178 ns for data found in memory.

Network Traffic. While timestamp snooping reduces the latency for obtaining data from other caches, this reduction comes at a cost of higher bandwidth. The extra bandwidth required for timestamp snooping can cause additional latency due to increased contention (not modeled here). Figure 4 illustrates the total traffic per butterfly link (left) and torus link (right) of all three protocols, normalized to that of TS-Snoop. The figure assumes 72-byte data messages (including a 64-byte data block) and 8-byte non-data messages (including the necessary bits of a 44-bit physical address). On the butterfly, TS-Snoop uses 13-43% more link bandwidth than the directory protocols (or alternatively, directory protocols use 12-30% less bandwidth than TS-Snoop). Results are similar on the torus, with TS-Snoop using 17-37% more bandwidth.[5]

The intuition for these numbers follows from a back-of-the-envelope calculation. On the butterfly, a timestamp snooping transaction sends an address packet over 21 links and receives a data packet over three links, for a total bandwidth of 384 bytes (21×8 + 3×72). Directory protocols, at a minimum, send an address packet over three links and receive a data packet over three links, for a total of 240 bytes (3×8 + 3×72). Thus, if all protocols took the same number of misses, the extra bandwidth used by timestamp snooping cannot exceed 60% (or alternatively, directories use at least 63% the bandwidth of timestamp snooping). The actual bandwidth difference is smaller, however, because (a) all MSI protocols send two data messages when a processor in state M gets a request for an S block, and (b) directory protocols send additional non-data messages, such as invalidations, acknowledgments, negative acknowledgments, and forwarded requests.

Furthermore, both butterfly and torus link bandwidth results are sensitive to block size and system size, which are currently 64 bytes and 16 processors, respectively. Doubling the block size on a 16-node butterfly, for example, reduces the upper limit on the extra bandwidth per miss of timestamp snooping to 33%. Increasing the number

3. Comparing runtimes between the two networks is problematic since the networks use different numbers of switches and wires.

4. "X is n% faster than Y" means that $Time_Y/Time_X - 1 = n\%$.

5. The data bandwidth used varies modestly with protocol, since protocols do not take the same number of misses. When spinning on a lock, for example, getting a nack rather than data may save data bandwidth by generating a re-request without a miss.

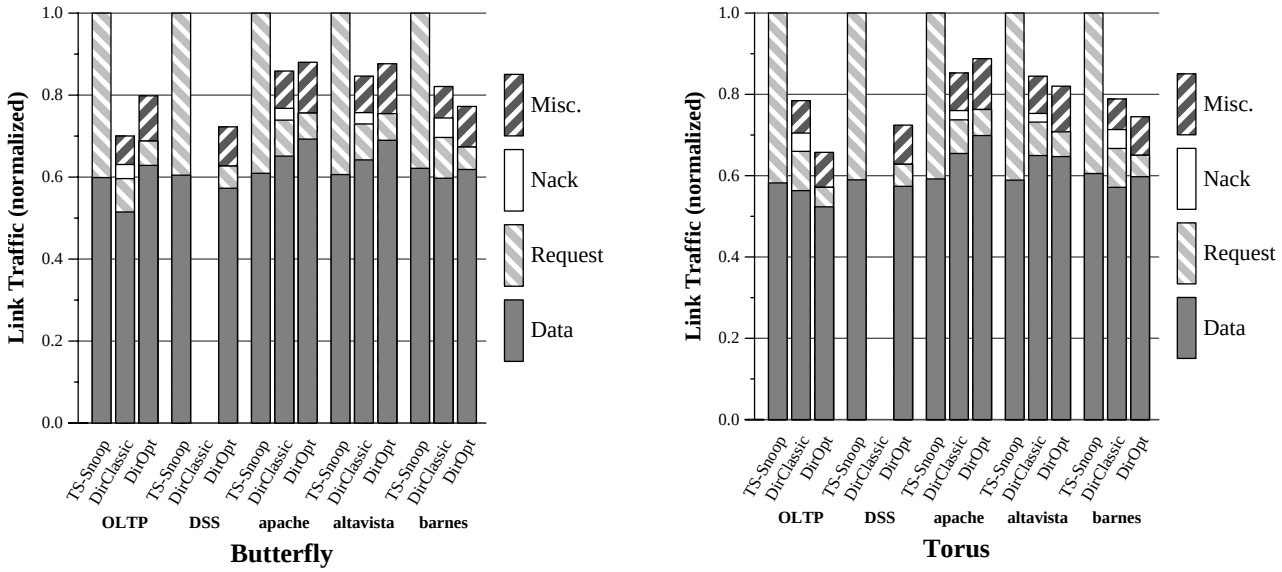

Figure 4. Normalized Link Traffic with Butterfly (left) and Torus (right)
Miscellaneous directory messages are for forwarding, invalidations, and acknowledgments.

of processors increases the cost of broadcasting each transaction. Thus, at larger number of processors, directory protocols or hierarchical coherence schemes become increasingly attractive. Conversely, reducing system size to 8 or 4 processors reduces the bandwidth requirements of timestamp snooping.

6 Related Work

Coherence Protocols. The *gray zone* [10] and *delta coherence* protocols [44] are the prior work most closely related to timestamp snooping. All three proposals implement coherent shared memory using messages that are (implicitly) timestamped over an interconnection network that recursively calculates the age of the oldest outstanding message. Both gray zone and delta coherence protocols are essentially write-through protocols because they send reads, writes, and read-modify-writes that *bind* remotely (e.g., writes can remotely update values). Delta protocols implement sequential consistency using an isotach network [37] to ensure that messages arrive exactly "on time." Gray zone implements weak consistency by allowing messages to arrive early, processing only non-synchronization messages early, but processing synchronizing read-modify-writes "on time." Gray zone also discards and retries messages that arrive late, which seems complex, especially for writes that update multiple copies.

Timestamp snooping, on the other hand, adapts the above ideas to work with writeback MOESI coherence protocols that are (nearly) universally used in commercial systems. For example, with MOESI, a non-binding coherence transaction allows a cache to enter state M and then perform many binding writes locally. Binding reads, writes and read-modify-writes do not occur remotely. MOESI coherence allows timestamp snooping to operate with processors that do or do not preserve the order of memory references to implement either sequential consistency or weak consistency with early transaction delivery and selective early transaction processing. There are also more technical differences (e.g., gray zone argues for a global physical clock and delta coherence protocols sometimes need to delay when a read hit takes effect).

Several other coherence protocols require explicit network order but, like timestamp snooping, do not rely on when transactions are delivered. Landin et al. [25] discuss many protocol options for the restricted case of acyclic network topologies with ordered links and FIFO buffers. Bilir et al. [9] describe a multicast snooping protocol whose network delivers transactions in order but at different times. Pong et al. [35] describe a broadcast snooping protocol with an ordered broadcast implemented on a central memory controller chip that is separated from four processors by FIFO queues. The directory protocol of Compaq's recent AlphaServer GS320 [18] uses a crossbar to provide a total order of forwarded requests and invalidations, thereby eliminating the need to acknowledge invalidations. Unlike these protocols, timestamp snooping uses logical timestamps to create a total order.

Simulation. Considerable work has been done using logical ordering times in an implementation to create correct function. Many parallel discrete event simulation (PDES)

algorithms [16] clearly separate the logical (virtual) time of the target system being simulated from the physical time of the host performing the simulation. Conservative PDES algorithms transmit logical times in messages and make recipients delay message processing until it is safe to proceed, while optimistic PDES algorithms, such as Jefferson's Time Warp [22], process messages speculatively and roll back if necessary. Timestamp snooping uses ideas from conservative PDES.

At least two other systems inspired by PDES use logical time to determine how computation is performed, but neither of these machines supports a shared-memory multiprocessor model as the present proposal does. Fujimoto's Time Warp machine [17] applies optimistic PDES to hardware to implement a *sequential* program execution on parallel hardware using speculation. Ranade [36] proposed hardware to mimic a PRAM model using techniques derived from conservative PDES.

Networks. Several other efforts have included logical time in network delivery. Isotach networks [37] deliver a message at precisely the logical time specified when it enters the network. Logical time is updated using a token-passing method that inspired how the timestamp snooping network manages the guarantee time (GT). Ranade's network ensures that memory references arrive at the correct PRAM cycle [36]. Multicast snooping's network [9] ensures that address multicasts arrive in the logical order determined by the network roots. All of these efforts delay messages so that they arrive at exactly the appropriate logical time. The network proposed in this paper seeks higher performance by delivering transactions as soon as possible, but not later than a logical time deadline.

7 Conclusions and Future Work

This paper presents *timestamp snooping* as a design for implementing MOESI snooping on switched networks. Timestamp snooping works by processing address transactions in a logical order determined by logical timestamps implicitly added to address transactions. Relative to two directory protocols, timestamp snooping reduces execution time by 6-29% at a cost of using 13-43% more interconnection network bandwidth on a 16-processor SPARC system running commercial workloads. Thus, as another example of the classic latency-bandwidth tradeoff, timestamp snooping is worth considering when buying more interconnect bandwidth is easier than reducing interconnect latency.

We see several promising avenues of future work. First, we would like to refine the timestamp network design of Section 2 and develop alternatives. Second, we would like to implement multicast snooping [9] on these networks to reduce transaction bandwidth and extend the system size for which snooping is viable. Third, we would like to speculatively send data in response to requests that arrive before their ordering times. Fourth, we would like to explore additional applications of logical time (e.g., to improve multiprocessor availability).

Acknowledgments

We thank Virtutech AB—especially Magnus Christensson, Johan Högberg, Peter Magnusson, Andreas Moestedt, and Bengt Werner—for their critical support of Simics; Gary Valentin, Gopi Attaluri, and Bernard Beaton for their support of IBM DB2; Paul Barford for the SURGE client; and Ernest Artiaga for the PARMACS macros. We thank the Condor group and Remzi Arpaci-Dusseau for providing additional computing resources. We thank the following for their comments on this work and/or paper: Wisconsin Computer Architecture Affiliates, Alan Baum, Anne Condon, Robert Cypher, Charles Fischer, Joel Emer, Steve Kunkel, Yannis Schoinas, Jeff Thomas, and Craig Zilles.

References

[1] Y. Afek, G. Brown, and M. Merritt. Lazy Caching. *ACM Trans. Prog. Lang. Syst.*, 15(1):182–205, Jan. 1993.

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

[3] Altavista Business Solutions. http://doc.altavista.com/business_solutions/bus_solutions.html.

[4] Apache HTTP Server Project. http://www.apache.org/httpd.html.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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*, May 1999.

[10] R. Bisiani, A. Nowatzyk, and M. Ravishankar. Coherent Shared Memory on a Message Passing Machine. In

Proceedings of the 1989 International Conference on Parallel Processing, pages I–133–141. ICPP, August 1989.

[11] J. Borkenhagen and S. Storino. 4th Generation 64-bit PowerPC-Compatible Commercial Processor Design. IBM Whitepaper, January 13, 1999, http://www.rs6000.ibm.com/resource/technology/nstar.pdf.

[12] A. Charlesworth. Extending the SMP Envelope. *IEEE Micro*, pages 39–49, Jan/Feb 1998.

[13] K. Diefendorff. Power4 Focuses on Memory Bandwidth. *Microprocessor Report*, 13(13), Oct. 1999.

[14] J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks*. IEEE Computer Society Press, 1997.

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

[16] R. M. Fujimoto. Parallel Discrete Event Simulation. *Commun. ACM*, 33(10):30–53, Oct. 1990.

[17] R. M. Fujimoto. The Virtual Time Machine. In *Proceedings of the Second ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, June 1990.

[18] 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 (ASPLOS IX)*, Nov. 2000.

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

[20] M. Horowitz, C.-K. K. Yang, and S. Sidiropoulos. High-Speed Electrical Signaling: Overview and Limitations. *IEEE Micro*, 18(1), January/February 1998.

[21] C. Hristea, D. Lenoski, and J. Keen. Measuring Memory Hierarchy Performance of Cache-coherent Multiprocessors Using Micro Benchmarks. In *Proceedings of Supercomputing '97*, Nov. 1997.

[22] D. R. Jefferson. Virtual Time. *ACM Trans. Prog. Lang. Syst.*, 7(3):404–425, July 1985.

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

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

[25] A. Landin, E. Hagersten, and S. Haridi. Race-Free Interconnection Networks and Multiprocessor Consistency. In *Proceedings of the International Symposium on Computer Architecture*, June 1991.

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

[27] G. Lauterbach and T. Horel. UltraSPARC-III: Designing Third Generation 64-Bit Performance. *IEEE Micro*, 19(3), May/June 1999.

[28] C. E. Leiserson. Systolic Priority Queues. In *Caltech Conference on VLSI*, pages 199–214, Jan. 1979.

[29] T. D. Lovett and R. M. Clapp. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 308–317, May 1996.

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

[31] S.-W. Moon, J. Rexford, and K. G. Shin. Scalable Hardware Priority Queue Architectures for High-Speed Packet Switches. In *Proc. IEEE Real-Time Technology and Applications Symposium*, pages 203–212, June 1997.

[32] A. Nowatzyk. Performance Analysis of Hypercube Based Ensemble Machine Architectures. Phd thesis, Carnegie-Mellon, 1989.

[33] A. Nowatzyk, M. Monger, M. Parkin, E. Kelly, M. Borwne, G. Aybay, and D. Lee. S3.mp: A Multiprocessor in a Matchbox. In *Proc. PASA*, 1993.

[34] G. M. Papadopoulos. SC99 State-of-the-Field Address, 1999.

[35] F. Pong, M. Dubois, and K. Lee. Design and Performance of SMPs with Asynchronous Caches. Technical Report HPL-1999-149, HP Labs, Nov. 1999.

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

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

[38] A. Singhal, D. Broniarczyk, F. Cerauskis, J. Price, L. Yaun, C. Cheng, D. Doblar, S. Fosth, N. Agarwal, K. Harvery, E. Hagersten, and B. Liencres. Gigaplane: A High Performance Bus of Large SMPs. In *IEEE Hot Interconnects*, pages 41–52, Aug. 1996.

[39] 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.

[40] 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.

[41] Transaction Processing Performance Council. TPC Benchmark C, Draft Specification, Revision 4.0.q, Aug. 1999.

[42] Transaction Processing Performance Council. TPC Benchmark H (Decision Support), Standard Specification, Revision 1.1.0, June 1999.

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

[44] C. Williams, J. Paul F. Reyolds, and B. R. de Supinski. Delta Coherence Protocols. *IEEE Concurrency*, 8(3):21–27, July-September 2000.

[45] S. C. Woo, M. Ohara, E. Torrie, J. P. Shingh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 22–24, 1995.