DESIGN AND ARCHITECTURE OF THE ALPHASERVER GS320

Mr. Gopal Behera[1], Ms. Swarnakanti Samantaray[2]*
*[1]*Assistant Professor,Dept. Of Computer Science and Engineering, NIT , BBSR*
[2]Assistant Professor,Dept. Of Computer Science and Engineering, NIT , BBSR
gopalbehera@thenalanda.com, swarnakanti@thenalanda.com*

Abstract

This paper details the design and implementation of the Compaq Al- phaServer GS320, a multiprocessor with cache-coherent non-uniform memory access. With 32 to 64 processors, the AlphaServer GS320 architecture is specifically designed for medium-scale multiprocessing. Four Alpha 21264 processors, up to 32GB of coherent memory, and an aggressive IO subsystem make up each node in the architecture. For a total of 32 processors, the current implementation supports up to 8 of these nodes. While some companies have extended snoopy-based designs to medium-scale multi-processors, supplying enough snoop bandwidth remains a significant difficulty, especially in systems with aggressive processors. In contrast to snoopy designs, directory protocols aimed for bigger scale designs have a number of intrinsic inefficiencies. .

The Compaq AlphaServer GS320, a multiprocessor with cache-coherent non-uniform memory access, is designed and implemented in detail in this work. The AlphaServer GS320 architecture, with 32 to 64 processors, is made specifically for medium-scale multiprocessing. Each node in the architecture consists of four Alpha 21264 processors, up to 32GB of coherent memory, and an aggressive IO subsystem. The current implementation supports up to 8 of these nodes for a total of 32 CPUs. Even while some businesses have expanded snoopy-based designs to medium-scale multi-processors, providing enough snoop bandwidth is still a major challenge, especially in systems with aggressive processors. Directory protocols intended for larger scale designs, in contrast to snoopy designs, have a variety of inherent inefficiencies.

1 Introduction

Shared-memory multiprocessors have been a major focus of study and development by both academia and industry, leading to sig- nificant design improvements during the past decade. Snoopy-based multiprocessors, which depend on broadcasting coherence transactions to all processors and memory, have moved well bey- ond the initial designs based on a single bus. The Sun Enterprise 10000 [10, 34], for example, extends this approach to up to 64 pro- cessors by using four-way interleaved address buses and a 16x16 data crossbar. Nevertheless, snoop bandwidth limitations, and the need to act upon all transactions at every processor, make snoopy designs extremely challenging especially in light of aggressive pro- cessors with multiple outstanding requests.

Directory-based multiprocessors [9, 28], which depend on maintaining the identity of sharers (at the directory) to avoid the need for broadcast, are much better suited for larger designs. A state-of-the-art example is the SGI Origin 2000 [27] which can scale to several hundred processors. Furthermore, the typical NUMA (non-uniform memory access) nature of directory-based designs, considered to be a liability by some, can in fact lead to major performance benefits over snoopy designs by exploiting the lower latency and higher bandwidth local memory and alleviating the need for more global bandwidth. Simple techniques, such as replication of application and operating system code, can provide major gains in commercial workloads with large instruction foot- prints [4]. More sophisticated software techniques that transpar- ently migrate and replicate pages have also been shown to be quite effective [38]. Nevertheless, existing directory protocols exhibit several inefficiencies relative to snoopy protocols, partly due to their steadfast focus on large-scale systems. For example, the use of various acknowledgement messages and multiple protocol invoca- tions at the home node (e.g., for 3-hop transactions), which help deal with races that arise due to the distributed nature of the proto-cols and the underlying scalable networks, can lead to undesirably high protocol resource occupancies.

Meanwhile, small and medium scale multiprocessors (i.e., 4 to 64 processors) account for virtually all the revenue in the server market, with small servers (i.e., 4 to 8 processors) having by far the largest volume. While it is feasible to build larger servers with hundreds of processors, the market demand for such sys- tems is extremely limited due to the lack of (i) scalable applica- tions and operating systems, and (ii) a compelling solution that ad- dresses reliability and fault-containment in larger shared-memory systems [8, 19, 35]. Yet, there has been surprisingly little research on scaling down directory protocols to provide a more efficient alternative to snoopy protocols especially for medium-scale serv- ers. One of the key goals of the AlphaServer GS320 architecture is to achieve the best-of-both-worlds by tailoring a directory-based protocol to eliminate inefficiencies associated with existing designs and to exploit the limited scale of the target systems.

The AlphaServer GS320 architecture is specifically targeted at medium-scale multiprocessing with 32 to 64 processors. Figure 1 shows a block diagram of the system. Each node consists of four Alpha 21264 [23] processors, up to 32GB of coherent memory, and an aggressive IO subsystem. The current implementation sup- ports up to 8 such nodes connected through an external crossbar switch for a total of 32 processors. This design began in early 1996. The quad-processor node first booted in March 1999, followed by the 16 and 32 processor systems booting in July and September
1999. The AlphaServer GS320 supports three different operating systems: Tru64 Unix, VMS, and Linux (for small configurations).

This paper focuses on the novel design features used in the Al- phaServer GS320 to efficiently implement coherence and consist- ency. The hierarchical nature of our design and its limited scale make it feasible to use simple interconnects such as a crossbar switch to connect the handful of nodes. One of the guiding prin- ciples for our directory-based protocol is to exploit the extra or- dering properties of the switch. The other guiding principle is to address correctness issues related to rare protocol races without burdening the common transaction flows. We have developed a protocol that deals with deadlock issues and various protocol races without resorting to typical negative-acknowledgement and retry mechanisms. This approach also naturally lends itself to simple and elegant solutions for livelock, starvation, and fairness. Our protocol exhibits lower occupancy and fewer message counts compared to previous designs. We have especially optimized occupancy issues related to 3-hop transactions, which have been shown to occur fre- quently in commercial workloads [4]. While our directory pro- tocol specifically targets small and medium-scale servers, several of the same techniques can be applied to larger-scale designs. In fact, the protocol design ideas explored in the AlphaServer GS320 have already influenced other more recent designs within Compaq, including the Alpha 21364 [3] (next-generation Alpha processor with glueless scalable multiprocessing support), Piranha [5] (re- search prototype that explores scalable chip-multiprocessing), and Shasta [29, 31] (a software DSM system). These systems do not depend on any special network ordering, with Alpha 21364 and Piranha not even depending on point-to-point order.

The AlphaServer GS320 architecture also incorporates a couple of innovative techniques that extend previous approaches for effi- ciently implementing memory consistency models. The *first* tech- nique involves generating a commit event (used for memory order- ing purposes) well in advance of formulating the reply to a trans- action. Commit events have been used in a limited form in current designs for early acknowledgement of invalidation messages ([13], Section 5.4). This allows a processor to move past ordering points (e.g., memory barrier in Alpha [33]) possibly before its invalid- ations take place in the target caches. We extend the use of early commits to all read and read-exclusive transactions, allowing a pro- cessor to go beyond ordering points before its pending transactions are serviced by the target caches or memory. It is intuitively sur- prising that this optimization actually works since the commit event is generated well before binding the value of the data reply. The *second* technique is applicable to systems that exploit any form of early commit events. Previous techniques for achieving correct- ness in such systems lead to either extra delay on inbound data (and acknowledgement) replies or extra delay at memory ordering points [13]. We eliminate these undesirable delays by separating out the commit event and allowing the time-critical reply com- ponent to bypass other inbound messages. The above two tech- niques are applicable to both larger-scale directory and smaller- scale snoopy protocols, and are complementary to existing tech- niques for efficiently implementing consistency models [14].

We also present results that characterize the latency and band- width properties of the AlphaServer GS320 and evaluate the im- pact of some of the above optimizations. The rest of the paper is structured as follows. The next section provides an overview of the AlphaServer GS320 architecture. Sections 3 and 4 present the novel aspects of our coherence protocol and consistency model im- plementation, and describe generalizations of these techniques to other directory- and snoopy-baseddesigns. The current implement- ation of this architecture is briefly described in Section 5. Section 6 presents some performance results. Finally, we discuss related work and conclude.

2 AlphaServer GS320 Architecture Overview

As shown in Figure 1, the AlphaServer GS320 architecture is a hierarchical shared-memory multiprocessor consisting of up to 8 nodes, referred to as *quad-processor building blocks (QBB)*. Each QBB consists of up to four processors, up to 32GB of memory, and an IO interface all connected via a *local switch*. The QBBs are in turn connected to an 8x8 *global switch*. A fully configured system supports 32 Alpha 21264 processors, 256GB of memory, 64 PCI buses (224 PCI adapters), with an aggregate memory bandwidth of 51.2GB/s, a global switch bi-section data bandwidth of 12.8GB/s, and an aggregate IO bandwidth of 12.8 GB/s.

Quad-Processor Building Block (QBB)

Figure 1 depicts the logical organization of a quad-processor build- ing block. The QBB is built around a 10-port local switch. Four ports are occupied by processors, four by memory, and one each by the IO interface and the global port. The switch has an aggreg- ate data bandwidth of 6.4GB/s, with each port (except global port) at 1.6GB/s (data transfer bandwidths, excluding address and error code bits). The global port is used for connecting the QBB to othernodes, and supports 1.6GB/s in each direction for a total port band- width of 3.2GB/s. The local switch is not symmetric; for example, no connections are possible between two memory ports.

The QBB supports up to four Alpha 21264 [23] processors, cur-rently running at 731 MHz. The Alpha 21264 has separate 64KB 2-way-associative on-chip instruction and data caches (64-byte linesize), and a 4MB external cache. Each processor supports up to 8 outstanding memory requests and an additional 7 outstanding vic- tims/writebacks. Each QBB also supports up to four memory mod- ules, each with 2-8GB of SDRAM memory with up to 8-way in- terleaving. The four modules provide a total capacity of 32GB and an aggregate memory bandwidth of 6.4GB/s. The IO interface sup- ports up to 8 PCI buses (64-bit, 133MHz), with support for 28 PCI slots. This interface supports a small cache (64-entry, fully asso- ciative) to exploit spatial locality for memory operations issued by IO devices, and allows for up to 16 outstanding memory operations and an additional 16 victims/writebacks. Furthermore, the inter- face supports a prefetching mechanism [14] to allow simultaneous memory accesses even though IO devices require strict ordering among memory operations.

The QBB employs a duplicate tag store (DTAG) to keep track of cached copies within the node. The DTAG is a logically cent- ralized data structure that maintains an external copy of each of the four processors' second-level cache tags, and serves as the primary module for maintaining coherence within a QBB. Maintaining co- herence across multiple QBBs requires two other modules: the dir- ectory (DIR) and the transactions-in-transit table (TTT). The dir- ectory maintains a 14-bit entry per 64-byte memory line (approx. 2.5% overhead). This includes a 6-bit field that identifies one of 41 possible owners (32 processors, 8 IO interfaces, and memory), and an 8-bit field which is used as a full bit-vector to maintain the identity of sharers at the granularity of a QBB. The identity of the owner and sharers are maintained simultaneously because our pro- tocol supports dirty-sharing. The sharing bit-vector at the directory (at QBB granularity) along with the DTAG at the target nodes to- gether identify the exact identity of the sharing processor caches. Finally, the TTT is a 48-entry associative table which keeps track of pending transactions from a node.

Two QBBs can be connected directly through their global ports to form an 8 processor configuration. Larger configurations require the use of the global switch.

Global Switch (GS)

The global switch (GS) has 8 ports, each supporting 3.2GB/s of data bandwidth (1.6GB/s in each direction), with an overall data

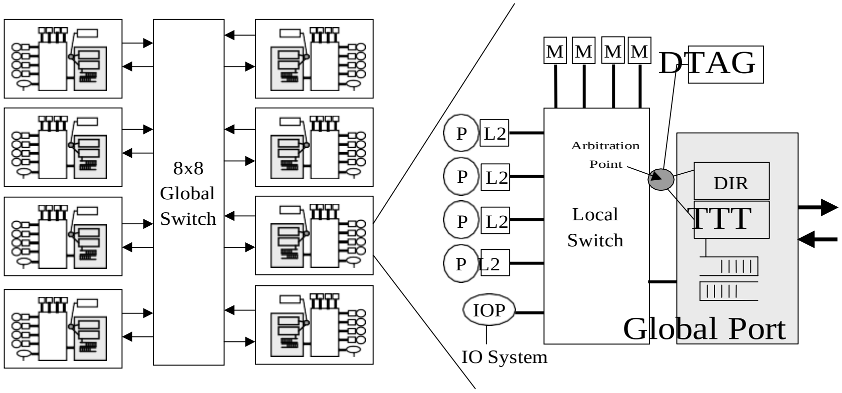


Figure 1: Block diagram of the AlphaServer GS320 architecture and the quad-processor building block.

bi-section bandwidth of 12.8GB/s. The GS is implemented as a centrally-buffered switch, and supports multiple virtual lanes to al- leviate coherence protocol deadlocks. All incoming packets are logically enqueued into a central buffer, and dequeued independ- ently by output port schedulers. This model allows us to efficiently implement totally-ordered multicast for specific virtual lanes where such ordering is desirable.

3 Optimized Cache Coherence Protocol

The design of the AlphaServer GS320 cache coherence protocol has two goals. The first goal is to reduce inefficiencies in current state-of-the-art directory-based protocols that arise from burdening common transaction flows because of the solutions used to deal with rare protocol races. The second goal is to exploit the limited size of our system, and the extra ordering properties of our inter- connect, to reduce the number of protocol messages and the cor- responding protocol resource occupancies. As we will see, there is synergy among the various mechanisms we use in our protocol, leading to a simple and efficient implementation that minimizes special case logical structures to deal with rare protocol races.

The cache coherence protocol in the AlphaServer GS320 is an invalidation-based directory protocol with support for four request types: *read*, *read-exclusive*, *exclusive* (requesting processor has a shared copy), and *exclusive-without-data*[1]. The protocol supports *dirty sharing*, which allows data to be shared without requiring
the home node to have an up-to-date copy. We also support reply forwarding from remote owners and eager exclusive replies (own- ership given before all invalidations are complete). As discussed later, we eliminate the need for invalidation acknowledgements by exploiting the ordering properties in the switch.

The intra-node protocol uses virtually the same mechanisms and transaction flows as the inter-node protocol to maintain co- herence within a node, with the local switch replacing the global switch as the transport. For a single-node system, or at a home node, the duplicate tag (DTAG) logically functions as a centralized full-map directory by providing sharing information for the four local processors. Remote memory accesses are sent directly to the home node (similar to SGI Origin [27]), without incurring delays to check whether they can be serviced by another local processor.

A key design decision in our protocol is to handle corner cases without depending on negative-acknowledgements (NAKs)/retries or blocking at the home directory. NAKs are typically used in scalable coherence protocols to: (i) resolve resource dependencies that may result in deadlock (e.g., when outgoing network lanes back up), and (ii) resolve races where a request fails to find the data at

[1] This supports the Alpha write-hint instruction (wh64) which indicates intent to write the entire cache line, thus avoiding a fetch of the line's current contents.the node or processor it is forwarded to (or, in some designs, whenthe directory at home is in a "busy" state). Similarly, blocking at the home directory is sometimes used to resolve such races.

Eliminating NAKs/retries and blocking at the home leads to several important and desirable characteristics. *First*, by guaran- teeing that an owner node (or processor) can always service a forwarded request, all directory state changes can occur immediately when the home node is first visited. Hence, all transactions com- plete with at most a single message to the home (i.e., the original request) and a single accessto the directory (and DTAG). This leads to fewer messages and less resource occupancy for all 3-hop read and write transactions (involving a remote owner) compared to pro-tocols that send extra confirmation messagesback to the home (e.g., "sharing writeback" or "ownership change" in DASH [28] and SGI Origin [27]). *Second*, our directory controller can be implemented as a simple pipelined state machine wherein transactions immedi- ately update the directory, regardless of other ongoing transactionsto the same line. Hence, we avoid blockages and extra occupancy at the directory, and instead resolve dependencies at the system periphery. *Third*, our early commit optimization for implement- ing memory consistency models (Section 4) also depends on the guarantee that an owner can always service a request. *Fourth*, we inherently eliminate livelock, starvation, and fairness problems that arise due to the presence of NAKs. In contrast, the SGI Origin [27] uses a number of complicated mechanisms such as reverting to a strict request-reply protocol, while other protocols with NAKs ignore this important problem [24, 28].

Avoiding Protocol Deadlock

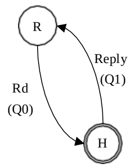
Our protocol uses three virtual lanes (Q0, Q1, and Q2) to elim- inate the possibility of protocol deadlocks without resorting to NAKs/retries. The first lane (Q0) carries requests from a processorto a home. Messages from the home directory/memory (replies or forwarded messages to third-party nodes or processors) are always carried on the second lane (Q1). Finally, the third lane (Q2) car- ries replies from a third-party node or processor to the requester. Our protocol requires an additional virtual lane (QIO, used to carry requests to IO devices) to support a subtle PCI ordering rule (bey- ond the scope of this paper). As we will see, our protocol depends on a *total ordering* of Q1 messages (comes naturally in a crossbar switch) and point-to-point ordering of QIO and Q0 (same address) messages, with no ordering requirements on Q2 messages.

Dealing with Request Races

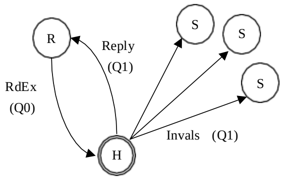
There are two possible races when a request is forwarded to an owner node or processor. The *late request race* occurs if the request
arrives at the owner after the owner has already written back the line. The *early request race* occurs if a request arrives at the ownerbefore the owner has received its copy of the data. Our solutions forthese races guarantee that the forwarded request is serviced without any retrying or blocking at the directory.

Our solution for the late request race involves maintaining a valid copy of the data at the owner until the home acknowledges the writeback, allowing us to satisfy any forwarded requests in the interim. Our protocol uses a two-level mechanism. First, when the Alpha 21264 processor victimizes a line, it awaits a victim- release signal before discarding the data from its victim buffer. The victim-release signal is effectively delayed until all pending forwar- ded requests from the DTAG to a given processor are satisfied. The above approach alleviates the need for complex address matching (used in snoopy designs) between incoming and outgoing queues. For writebacks to remote homes, the responsibility of maintaining the data is handed off to the transactions-in-transit table (TTT) in order to relieve the pressure on the processor's victim buffers. This copy is maintained until the home acknowledges the writeback.

Our solution for the early request race involves delaying the for- warded request (on Q1) until the data (on Q2) arrives at the owner. This functionality is supported within the Alpha 21264 processor



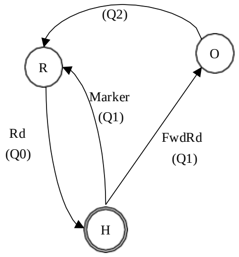
(a) 2-hop Read

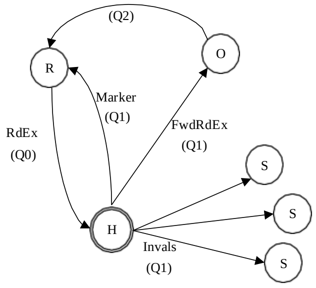


(b) 2-hop Read-Exclusive

Reply

Reply





whereby the address at the head of the inbound probe queue (Q1) is compared against addresses in the processor's miss-address-file

(c) 3-hop Read

(d) 3-hop Read-Exclusive

(tracks pending misses) and is delayed in case of a match. Buffer- ing early requests on the side to completely eliminate the possibil- ity of backing up the Q1 lane would require too large a buffer (256 entries in our design) due to the dirty-sharing property of our pro- tocol. Stalling the head of the Q1 lane at target processors provides an extremely simple resolution mechanism, and is relatively effi- cient since such stalls are rare and the amount of buffering at target nodes is sufficient to avoid impacting Q1 progress at the switch. Nevertheless, naive use of this technique can potentially lead to deadlock. All such deadlock scenarios are eliminated, however, due the total ordering of Q1 messages in our design.[2]

Finally, the hierarchical nature of our design allows transactions to be serviced within a node (e.g., at the home) without necessar- ily involving the global switch. This optimization is critical for achieving optimal system performance, yet it causes subtle interac- tions with the total ordering requirement for Q1 messages. The fol- lowing scheme is used for correctness. The transactions-in-transit table (TTT) at the home node keeps track of Q1 messages that are sent out on behalf of local processors but have not yet reached the global switch due to buffering. In the rare case that (i) a subsequent request to the same address arrives while such a Q1 message is in transit, and (ii) the request can be serviced through a local Q1 for- ward within the node, the latter Q1 forward is looped to the global switch and back to ensure total ordering on Q1 messages.

Putting it All Together: An Efficient Low Oc-cupancy Protocol

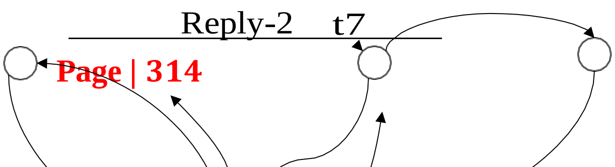
Figure 2 shows several basic transaction in our protocol. We use the following notation: R is the requester, H the home, O the owner, and S a sharer. The virtual lane (Q0, Q1, Q2) used by a message is shown in parentheses. These protocol flows also apply to our intra-node protocol, with the duplicate-tag (DTAG) behaving as home.

Figure 2(a) shows a basic 2-hop read case. Figure 2(b) shows a write to a line with multiple sharers. The data reply is sent to the requester at the same time invalidates are sent to the sharers. This flow illustrates two interesting protocol properties. First, our pro- tocol does not use invalidation-acknowledgement messages, which reduces message count and resource occupancy. Given the total or- dering property on Q1, an invalidate appears to be "delivered" to its target node when it is scheduled on the switch. Second, as an optimization, we use the multicast capability of our switch whenever the home needs to send multiple Q1 messages to different nodes as part of servicing a request. In this example, we inject a single message into the switch that atomically schedules the appropriate invalidate messages and the reply to the requester.

Figure 2: Basic protocol transaction flows.

Figure 2(c) shows a 3-hop read transaction. The home forwards the request to the owner and immediately alters the directory to re-flect the requester as a sharer. As mentioned before, the immediate change to the directory is possible because the owner is guaranteed to service the forwarded request. The owner directly responds to the requester, and the dirty-sharing property of our protocol avoids the need for a sharing writeback message to home (typical in other protocols). The message labeled *"marker"* sent from home to the requester serves several purposes in our protocol. First, the marker is used at the requester to disambiguate the order in which other requests to the same line (that are forwarded it) were seen by the directory. For example, the requester node filters out any invalidate messages that arrive before the marker, while an invalidate mes- sage that arrives after the marker is sent to the requesting processor. Second, the marker serves as the commit event for the read which is used for memory ordering purposes (discussed in Section 4.3). Finally, Figure 2(d) shows a 3-hop write transaction. Given the dirty-sharing nature of our protocol, it is possible for a line to have an owner and multiple sharers as shown in this scenario. As in the 3-hop read-case, the directory is changed immediately, and (in contrast to other protocols) there are no further messages sent to the home to complete this transaction. The marker serves the same purpose as in the read case, and is also used to trigger invalidates to other processors (sharing the line) on the requester's node.

Figure 3 shows an interesting consequence of changing direct-ory state immediately and our early request race solution. The scenario shown involves multiple nodes writing to the same line (initially dirty at a processor at the home node; "marker" messages not shown for simplicity). The writes are serialized at the directory, each is forwarded to the "current" owner and immediately changes the directory to reflect the new owner. The early request race mech-anism delays forwarded requests (on Q1) that reach their targets early. As each requester gets its reply (on Q2), the data ripples from one owner to the next without involving any further action from the



[2] Consider P1 owning A and requesting B exclusively, while P2 owns B and re- quests A exclusively. The home for A forwards a request R1/A to P1, and the home of B forwards a request R2/B to P2. Since the directories are changed immediately, they reflect P1 owning B and P2 owning A now. Assume P3 requests A causing request R3/A to P2, and P4 requests B causing request R4/B to P1. Deadlock can occur if R4/B arrives at P1 before R1/A *and* R3/A arrives at P2 before R2/B. However, the total ordering on Q1 messages disallows such reorderings.

(Q2)

R1 – R2

Reply-3 t8
(Q2)

R3

can be satisfied by the interconnect. These techniques can however be employed in scalable design that are hierarchical. While a scal- able interconnect may be used among nodes, it is possible to use switches with more ordering guarantees within each node (thus en- abling optimizations for intra-node coherence, as in Piranha).

Reply-1 FwdRdEx-2 RdEx-2

RdEx-1 (Q0)
t1
(Q2)
t6
(Q1)
t3

H

(Q0)
t2
FwdRdEx-3 (Q1)
t5
RdEx-3 (Q0)

t4

4 Efficient Implementation of ConsistencyModels

This section describes the innovative techniques used in the Al-

Figure 3: Protocol behavior with multiple writers to the same line.

directory. In such pathological cases, our protocol is much more ef- ficient than protocols (e.g., SGI Origin [27]) that resort to blocking at the directory or NAKing/retrying to deal with races.

3.4 Applicability of Protocol Optimizations toOther Designs

A number of the techniques used in the AlphaServer GS320 pro- tocol design are applicable to larger-scale directory designs, while a few of the techniques exploit interconnect ordering properties that are more feasible in small-to-medium scale designs. The ideas ex- plored in the AlphaServer GS320 have already influenced several more recent designs within Compaq: the Alpha 21364 [3, 6] (next- generation Alpha processor with glueless scalable multiprocessing support), Piranha [5] (research prototype that explores scalable chip-multiprocessing), and Shasta [29, 30, 31] (a software DSM system). These three systems provide scalable designs with no spe- cial constraints on network ordering; the Alpha 21364 and Piranha designs do not even depend on point-to-point order.

Scalable directory designs can benefit from elimination of NAKs/retries and blocking at the directory, and the use of more than two virtual lanes to avoid protocol deadlocks (used in both Al- pha 21364 and Piranha). First, all protocols can benefit from the simple and elegant solution to livelock and starvation problems. Second, our solution to the late request race is applicable to many designs (used in Piranha). However, our early request race solu- tion is more specific to our design choices. For protocols that do not support dirty-sharing, it is feasible to buffer the early request on the side (as done in Piranha) instead of stalling the request path. Finally, efficient 3-hop write transactions (with a single visit to the home) are possible in protocols where the mechanisms for dealing with early and late races do not depend on revisiting the home node in the common case (holds for Piranha). However, protocols that do not support dirty-sharing (e.g., Piranha, Alpha 21364) can not benefit from lower message count and occupancy for 3-hop reads because of the presence of a "sharing-writeback" message to home.

Software shared-memory protocols such as Shasta are quite dif- ferent from hardware protocols. Since software protocols can use main memory for extensive buffering purposes, multiple network lanes are not needed for avoiding resource deadlocks. Extensive buffers also allow support for dirty-sharing, with early requests buf- fered on the side. Finally, given that main memory (backed up by virtual memory) on each node acts as a software-controlled cache, late request races are not possible since there are no forced write- backs or replacements. The Shasta protocol was heavily influenced by the AlphaServer GS320 design, and is the only other protocol we are aware of that supports efficient 3-hop transactions for both reads and writes through altering the directory state immediately and requiring at most a single visit to the home.

Our other optimizations, such as eliminating invalidation- acknowledgements and exploiting the multicast feature of our switch for efficiency, are clearly more applicable to small- and medium-scale designs where the appropriate ordering properties

phaServer GS320 that extend previous approaches for efficiently implementing memory consistency models. Section 4.1 reviews the early invalidation acknowledgement technique which is already used in many designs. This review makes it simpler to understand the two new optimizations used in the AlphaServer GS320, which are described in Sections 4.2 and 4.3. Finally, Section 4.4 discusses the applicability of these optimizations to other designs.

Early Acknowledgement of Invalidation Re-quests

To reduce the latency of invalidations, a common optimization is to acknowledge an invalidation request as soon as the request is placed in a target destination's (e.g., a cache hierarchy) incoming queue before all stale copies are actually eliminated. However, na- ive uses of this optimization can lead to incorrect behavior since the acknowledgement no longer signifies the completion of the write with respect to the target processor. The following is a brief sum- mary of the material in Section 5.4 of Gharachorloo's thesis [13] which describes a couple of implementation techniques that enable the safe use of early acknowledgements.

Consider a write operation with multiple completion events with respect to each processor in a system. For each write, we also define a *commit event* with respect to each processor. The commit event corresponds to the time when the invalidations caused by the write are either explicitly or implicitly acknowledged, and precedes the completion event with respect to a processor in cases involving an early acknowledgement. In designs which exploit early acknow- ledgements, the program order between a write W and a following operation Y is enforced by only waiting for W to *commit* with re- spect to every processor before issuing Y (there is no longer an explicit message that signals the completion of the write).

Figure 4 shows an example to illustrate the issues related to early acknowledgements. For simplicity, assume a sequentially consistent (SC) [25] invalidation-based protocol. Consider the pro- gram segment in Figure 4(a) with all locations initialized to zero. The outcome (u,v)=(1,0) is disallowed under SC. Assume P1 ini- tially caches both locations and P2 caches location A. Without early acknowledgements, P1 issues the write to A, waits for it to com- plete, and proceeds to issue the write to B. Therefore, the stale copyof A at P2 is eliminated before P1 even issues its second write. As long as P2 ensures its reads complete in program order, the outcome (u,v)=(1,0) will indeed be disallowed.

Now consider the scenario with early invalidation acknow- ledgements. P1's write to A sends an invalidation to P2. This inval- idation is queued at P2 and an acknowledgement reply is generated. At this point, the write of A is committed but has yet to complete with respect to P2 (i.e., P2 can still read the old value of A). While the invalidation remains queued, P1 can proceed to issue its write to B, and P2 can issue its read to B. Figure 4(b) captures the state of P2's incoming buffer at this point, with both the

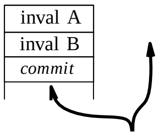
ISSN: 2278-4632
Vol-10 Issue-3 No.01 March 2020

invalidation re- quest for A and the read reply for B (return value of 1) queued. A key issue is that allowing the read reply to bypass the invalidation request in the buffer, which is desirable for performance reasons, will violate SC because P2 can proceed to read the stale value for A out of its cache after obtaining the new value for B.

P1 P2

— —

data/response



a1:

A = 1; *a2:* u = B;

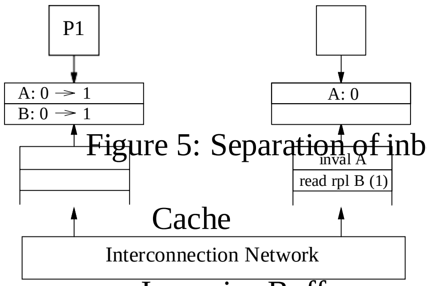
b1:

B = 1;

(a)

b2:

v = A;



P2

inbound reply

Figure 5: Separation of inbound reply to commit and data/response components.

Incoming Buffer

)

Cache

Incoming Buffer

involving remote transactions. Both cases lead to long inbound paths to the processor. The second solution described in the previ- ous section is impractical due to the overhead of flushing the long inbound path on every memory barrier. At the same time, using the first solution can delay time-critical replies behind inbound in- validate requests on the long inbound path to the processor. This section describes a simple yet powerful technique that we devised to alleviate the undesirable trade-off described above.

In most coherence protocols, processor requests are satisfied

Figure 4: Example illustrating early invalidation acknowledge- ments.

There are two known solutions to the above problem [13]. The *first* solution imposes ordering constraints among incoming mes- sages with respect to previously committed invalidations. Referring back to the example, this solution would disallow the read reply from bypassing the invalidation request, which forces the commit-ted write to A to complete with respect to P2 before the read of B completes. While FIFO ordering among all incoming messages from the commit point will work, it is sufficient to only maintain the queue order from an incoming reply (data or acknowledgement) to any previous incoming invalidates (allows for a lot of reordering on the inbound path for performance reasons) [13].

The *second* solution does not impose any ordering constraints among incoming messages. Instead, it requires previously committed invalidations to be serviced any time program order is enforced. In the example, this latter solution would allow the read reply to bypass the incom- ing invalidation, but would force the invalidation request to be ser- viced (e.g., by flushing the incoming queue) as part of enforcing the program order from the read of B to the read of A. Thus, both solutions correctly disallow the outcome (u,v)=(1,0).

The relative efficiency of the above two solutions heavily de- pends on the underlying memory consistency model. The first solution is better suited for strict models such as SC where en- forcing program orders occurs much more frequently than cache misses; the second solution is better suited for more relaxed models where enforcing program orders occurs less frequently. Further- more, for more relaxed models, the second solution may provide faster servicing of incoming replies (data or acknowledgement) by allowing them to bypass previous invalidate requests. However, the second solution can become inefficient in designs with deep inbound queues (e.g., due to a deep cache hierarchy); even though flushing of the incoming queues may be infrequent, the overhead of doing so can be quite high. The only known partial remedy to the above trade-off is a hybrid design that employs the first solution at the lower (farther) levels and the second solution at the higher levels of the logical inbound queue (or cache hierarchy) [13].

Separation of Incoming Replies into Commitand Data/Response Components

The AlphaServer GS320 supports the Alpha memory model which requires the use of explicit memory barrier instructions to impose memory ordering [33]. In addition, the commit point in our design is at the arbitration point within a node for accesses satisfied loc- ally or at the arbitration point for the global switch for accessesgh a single reply message. For example, a read or read- exclusive request receives a data reply, while an exclusive request (caused by a write to a clean copy) may receive a success or fail- ure response. Our approach separates the reply message into its two logical components when the message arrives at the inbound path: (i) the data or response component that is needed to service the request, and (ii) a commit component which is solely used for

ordering purposes.[3] This separation is illustrated in Figure 5. We allow the time-critical data/response component to bypass other in- bound (Q1) messages on its path to the processor (e.g., by using a separate lane such as Q2). To achieve correctness, the commit component is used as an ordering marker by placing it on the same path as other inbound messages and enforcing the required partial ordering with respect to other messages. For example, given the early invalidation acknowledgement optimization described in the previous section, a commit component cannot bypass any previous inbound invalidations. This approach is superior to either of the two solutions described in the previous section; we allow time-critical replies to bypass other inbound messages, and yet we do not require an explicit flush of the inbound path at memory barriers.

For the above scheme to work correctly and efficiently, support from the processor is needed to (i) expect two reply components instead of a single one, and (ii) appropriately account for the com- mit components. The Alpha 21264 maintains a count of pending requests. This count is incremented on every request issued to the system, and decremented each time a commit event is received. Re- ceiving the data/response component does not affect the count, and in fact there is no requirement for the data/response component to arrive before the commit component for achieving correct ordering. At a memory barrier, the processor waits for the count to reach zero before proceeding with other memory requests. Our design piggy- backs the commit components on other inbound Q1 messages with an additional 1-bit field; a null message is inserted if no messages are available for the piggyback. The next section describes our second optimization which allows us to generate the commit com- ponent for read and read-exclusive requests well before the actual data component is formulated.

Early Commit for Read and Read-ExclusiveRequests

The AlphaServer GS320 architecture extends the idea of early com- mits to encompass all types of processor requests instead of only read-exclusive (or exclusive) requests to a clean line with sharers (i.e., early invalidation commits described in Section 4.1). This optimization can reduce the delay whenever a processor must wait for its pending requests to complete for ordering purposes (e.g., at a

P1 P2

memory barrier). As we will discuss in the next section, the impact of this technique can be far-reaching since it is adds a fundamental

a1:

A = 1;{ t7}

a2:

u = B; { t1}

optimization to the bag-of-tricks designers can use to correctly and

b1:

MB;

b2:

MB;

{ t4}

efficiently implement memory consistency models.

In our design, early commits are generated for any read or read-exclusive request that is forwarded to be serviced by another cache's dirty or dirty-shared copy (includes forwards to a cache copy within the same node). This approach can be easily general-

c1: B = 1;

(a)

c2:

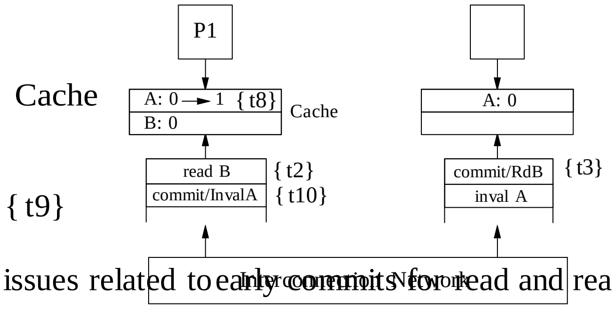
v = A; { t5}

ized to requests serviced by memory as well. However, this is not beneficial in our design because the commit can not be generated much in advance of the data reply from memory. Similar to early commits for invalidations, the early commit message is generated when a forwarded read or read-exclusive request arrives at the com- mit point (defined in the previous section). A separate data reply message is sent back once the forwarded request is serviced by the target cache. As with early invalidation commits, a processor is al- lowed to go past an ordering point (e.g., memory barrier) as long as all previous requests have received their commit replies, even if the actual data replies have not yet been received. Theoretically, the above optimization allows a processor to proceed beyond ordering points before the actual return values for its pending requests are bound. As would be expected, naive use of this optimization can

(b)

P2 *(reads A:0)* { t6}

(consumed before t4)



lead to incorrect behavior.

Figure 6 shows an example to illustrate the issues related to early commits for read and read-

[3] Optimized coherence protocols (e.g., DASH [28] or our protocol) support eager exclusive replies for writes to clean shared data. An eager reply is sent to the requester early on, with a follow-on message that signals the committing of invalidations at all sharers. These two messages cleanly map to the separation we require.

exclusive requests. Consider the program segment in Figure 6(a) with all locations initialized to zero ("MB" is a memory barrier in Alpha). The outcome (u,v)=(1,0) is disallowed under the Alpha memory model (and also under se-quential consistency). Assume P1 initially caches both locations (with dirty copy of B) and P2 caches location A. The figure shows a given order of events in time represented by $t1::t10$. Assume P2 issues read B, with the read request queued at P1. The com-mit message for the read is generated once the request is queued at P1, and is sent back to P2 (shown queued at P2 at time $t3$). Once this commit event is received by P2, P2 can go past the ordering point represented by the MB and read the value 0 for A (i.e., v=0). Note that we are allowing P2 to complete the read of A before the return value for the read of B (currently waiting to be serviced in P1's incoming queue) is even bound! Now assume P1 issues its write to A, which generates an invalidate to P2 and a correspond-ing invalidate-acknowledgement (early commit for the invalidate) to P1. Figure 6(b) shows the state of the incoming queues at this point (with the commit for read B already consumed).

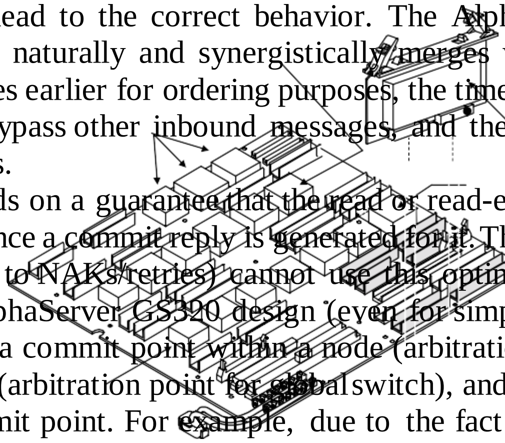
We can now illustrate the potential for incorrect behavior for the scenario in Figure 6. Consider the commit event for the inval- idation to A (commit/InvalA) bypassing the read request to B (read B) on the inbound path to P1. This reordering is allowed under the sufficient requirement for early invalidation acknowledgements since the only order that is required to be maintained is from com- mits to previous invalidation requests. Therefore, P1 can receive this commit, go through its memory barrier, and issue the write of B which would change its cache copy to have the value of 1. At this point, when the read of B (still in the inbound path) gets serviced, it will return the value of 1 (i.e., u=1). The scenario above violates the Alpha memory model since we have allowed (u,v)=(1,0).[4]

As with early invalidation acknowledgements, there are two

[4] The Alpha 21264 processor does not actually proceed past a memory barrier until both the commit and the data reply components for its previously pending requests are back (i.e., it is more conservative than the design assumed above). Furthermore, the 21264 has a small internal probe queue for incoming requests, allows replies to bypass this queue, and flushes the queue at memory barriers. Constructing the anomalous behavior is more involved in this case. Referring to Figure 6, assume the exact same order of events except that P2 (now a 21264) does not go past its memory barrier until it receives both the commit and the data reply for read B (assume u=1 as before). The anomalous behavior can still occur because the invalidate of A to P2 can be in the inbound path external to the 21264 chip and neither the data reply for B (which is allowed to bypass inbound messages) nor the commit (which is ahead of invalidate) force the invalidate into the 21264's internal probe queue. Therefore, P2 can still proceed past its MB to read the old value of A. The solutions described in the next paragraph also eliminate the possibility of incorrect behaviors with the Alpha 21264.

Figure 6: Example illustrating early commit for read requests.

solutions for guaranteeing correctness. The *first* solution involves imposing further order among inbound messages: a commit mes- sage cannot bypass any previous requests (read, read-exclusive or invalidation). This solution disallows the commit/InvalA message from bypassing the read B request in the scenario shown in Fig- ure 6. Therefore, the read B request is guaranteed to be serviced before P1 is allowed to change the value of B (P1 cannot go past its MB before it receives commit/InvalA), hence ensuring that read B returns the value 0 to P2 (i.e., u=0 leading to (u,v)=(0,0) which is an allowed outcome). The dynamics of how correctness is en- sured with the early commit optimization is quite interesting since we effectively force the return value for read B to be bound be- fore P1 is allowed to change the value for B. The *second* solution (also reminiscent of the second solution in Section 4.1 but slightly stricter) does not impose any ordering among inbound messages, but requires any request messages in the inbound path (read, read-exclusive, or invalidation) to be serviced any time a processor en- forces program order (e.g., at a memory barrier). Again, before P1 is allowed to complete its MB, it is required to service the inbound read B which will lead to the correct behavior. The AlphaServer GS320 design uses the first solution above because it naturally and synergistically merges with the optimization described in Sec-tion 4.2: the commit comes earlier for ordering purposes, the time-critical data reply which typically arrives later is allowed to bypass other inbound messages, and there is no requirement to flush the inbound path at memory barriers.



The early commit optimization described here depends on a guarantee that the read or read-exclusive request will be serviced by the target node or processor once a commit reply is generated for it. Therefore, protocols that do not make such a guarantee (e.g., due to NAKs/retries) cannot use this optimization. There are a few other subtle issues that arise in the AlphaServer GS320 design (even for simple early invalidation acknowledgements) due to the presence of a commit point within a node (arbitration point for local switch) and a commit point external to the node (arbitration point for global switch), and the fact that some requests are satisfied solely by the local commit point. For example, due to the fact that we sup- port eager exclusive replies, it is possible for a request generated at the home node to be locally satisfied while remote invalidations caused by a previous operation have still not been committed at the external switch. To avoid correctness issues, the transactions-in- transit table (TTT) detects such cases and forces the commit event for the latter operation to loop to the external switch and back in

a1: r1 = A;
b1: B = 1;

Local Switch (5 chips)
*c1:*r2 = C; *(hit)*
d1: r3 = D[r2]; Global Port
(for 8P systems)
e1: E = 1; *(hit)*
f1: r4 = F;
Processor/Memory Modules
g1: r5 = G[r4];

Broadcast Bus
IOP (3 chips)

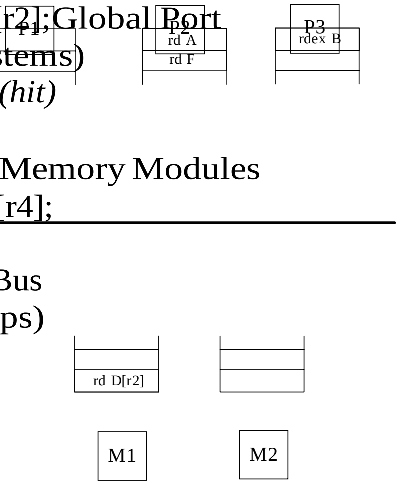


Figure 7: Early commits in a snoopy design.

order to inherit the previous operation's commit event and pull in any requests on the inbound path.

Applicability of Consistency Model Tech-niques to Other Designs

The two optimization techniques presented in Sections 4.2 and 4.3 can be used for implementing any memory consistency model, ranging from sequential consistency to aggressive relaxed memory models. The optimization of separating the commit and data/response components (Section 4.2) is primarily useful for implementation of more relaxed models where allowing the data/response to reach the processor earlier (before a memory or- dering point is encountered) is beneficial. Furthermore, the tech- nique applies to any implementation that exploits early commits, including early invalidation acknowledgements, and is superior compared to previously known solutions for enforcing correctness.

The performance benefits of the second optimization (Sec- tion 4.3) are higher for stricter memory models since a processor can continue past the frequent memory ordering points to issue new requests as soon as it receives the early commit for a previous re- quest. However, the benefits can be significant in relaxed models as well due to the reduction of delays at memory ordering points; for example, memory barrier latencies in Alpha multiprocessors constitute a significant fraction of the execution time for import- ant critical section routines in database applications. With respect to applicability to different implementations, the early commit op-timization is better suited to designs where the separation in time between generating the early commit and the actual reply is sig- nificant enough to justify generating two messages. In addition to less scalable designs with ordered networks (e.g., buses, crossbars, rings), this approach can also be beneficial in more hierarchical scalable designs where ordering is not maintained at the external interconnect but can be enforced within a node (i.e., commit point can be set at entry to node).

To illustrate the true potential of the early commit optimization, Figure 7 shows an example snoopy design supporting sequential consistency. In such a design, the commit point for a cache miss is when it is scheduled on the bus, and a processor must only await this commit event (i.e., does not need to wait for the actual reply) before issuing its next reference in order to satisfy sequential con- sistency; this of course assumes that the appropriate inbound mes- sage order is maintained as specified in the previous section. Given the example in the figure, P1 first issues the read of A (gets queued at P2). Given this read is committed, P1 can safely continue to is- sue the write to B on the bus. Once this write is scheduled on the bus (and queued at P3), P1 safely completes the read to C which is a cache hit. The value from this read is used to calculate the ad- dress of the next read, which is also issued on the bus. Next, P1 can safely complete its write to E which is a cache hit. Finally, P1 is- sues the read to F, and cannot proceed to issue the last read (G[r4])TAG

(4 chips)

Figure 8: Quad building block (QBB) motherboard (4-processor system).

since its address is not yet known. Note that at this point, P1 has 4 outstanding memory operations (one of them a read-exclusive) that are awaiting their data replies without violating sequential consist- ency! Furthermore, P1 was allowed to consume the value of a read

(C) and complete a write (E) in the middle of its miss stream. Sim- ilarly, the pending data replies can return and be consumed in any order. And it is also perfectly safe for P1 to make its writes (e.g., to E) visible to other processors even though it is awaiting data replies for previous operations. Finally, unlike speculative techniques pro-posed for implementing consistency models [14], the above ap- proach does not depend on any form of rollback; once an operation is committed, it is considered complete as far as ordering is con- cerned. However, as we will discuss in Section 7, there is potential synergy in combining the early commit technique with speculative techniques that depend on rollback [14].

5 AlphaServer GS320 Implementation

The AlphaServer GS320 is designed and packaged for modularity and easy upgradability from 4 to 32 processors. The basic build- ing blocks are: single-CPU boards, memory boards, PCI-bus inter- faces, and two types of backplanes. A 4-processor quad-building block (QBB) occupies a single rack, as does a PCI IO subsystem. Each QBB supports up to 2 IO subsystems. A cabinet supports up to 4 QBBs or 16 processors. Thus, a 32-processor system consists of two cabinets for processors and additional cabinets for IO.

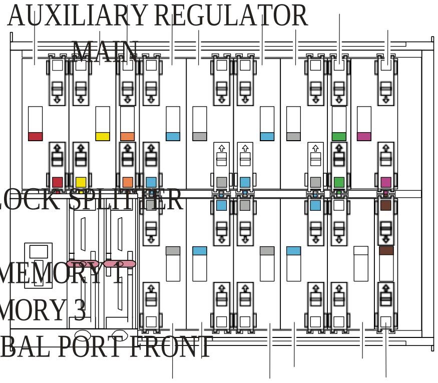
Figure 8 shows the motherboard for a 4-processor QBB, with the local switch, DTAG, and IO interface on the board. The global port chips reside on the motherboard for 8-processor systems (two 4-processor systems connected back-to-back), but become part of a plug-in module for larger systems. Other components are also attached as plug-in modules to the motherboard. For systems lar- ger than 8 processors, the QBB motherboard is mounted vertically, with one 4-processor rack facing the front and the other facing the rear. Figure 9(a) shows the placement of the plug-in modules on the rear side. Figure 9(b) depicts the rear side of the two cabinets used for a 32-processor system. Each quadrant is an 8-

processor rack. The global switch is mounted on a folding panel as shown. Ribbon cables connect the global ports to the global switch.

The complete system consists of 16 unique ASIC designs with a total of 7 million gates. There are 5 major address-path ASICs which constitute the core functionality of the system, and 4 data- path ASICs which are significantly simpler. The remaining ASICs perform simple glue logic tasks. The technology is circa 1997- 98, with about 500K useable gates on the large ASICs. The design and simulation environment included about 300 AlphaServer CPUs

Rear View of 8P Complex Showing Four Processors; Other Four Processors Appear in Front View

Table 1: Effective memory latencies for the AlphaServer GS320 with 731MHz 21264 processors.



Case	Back-to-Back Dependent Reads	Pipelined Independent Reads
L2 Cache Hit	23ns	7ns
Local, Clean	327ns	61ns
Local, Dirty	564ns	75ns
2-hop, Clean	960ns	136ns
3-hop, Dirty	1324-1708ns	196-310ns

Table 2: Impact of system load on L2 cache hit latency on three Alpha 21264-based systems.

	525Mhz GS140	500MHz ES40	731MHz GS320
L2 Hit Latency (P1-P3 idle)	35ns	41ns	23ns
L2 Hit Latency (P1-P3 active)	68ns	113ns	23ns

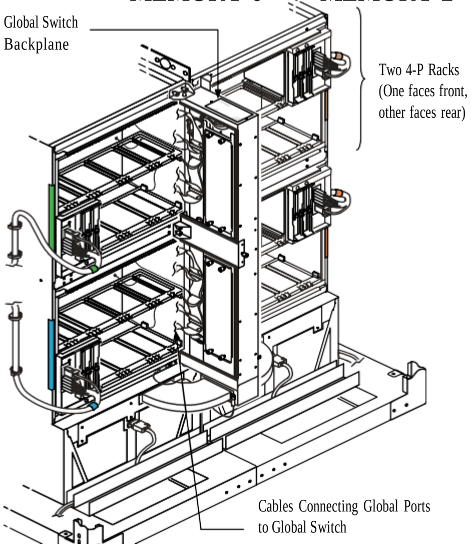


Figure 9: Rear view of (a) an 8-processor system and (b) 32- processor complex with global switch.

(about 40 clustered GS140 servers), equipped with a total of 500GB of memory and 4.5TB of disk. Systems of 100-200 million gates were simulated for a large number of cycles. Consequently, sys- tems booted with first-pass ASICs; the various configurations (4P, 8P, 32P) booted VMS and Unix within 2 weeks of assembly.

The AlphaServer GS320 design supports a number of RAS (re- liability, availability, serviceability) features. The system supports up to 8 hardware partitions at the QBB granularity, with hardware firewalls between partitions. Both Unix TruClusters and VMS Galaxy operating systems support such partitioning, with Galaxy aggressively exploiting dynamic software partitioning as well. The system also supports online removal, repair, and upgrade of QBBs, I/O subsystem, and individual CPUs. This approach permits hard- ware test and burn-in prior to reinsertion into a running system. Finally, the Tru64 Unix and VMS operating systems have been ex- tended to deal with the NUMA nature of the AlphaServer GS320 for both scheduling and memory management purposes.

6 Performance Measurements on the Al-phaServer GS320

This section presents results to characterize the basic latency and bandwidth parameters of the AlphaServer GS320 and to evaluate the impact of some of the optimizations described in Sections 3 and 4. In addition, we report results on a few industry-standard bench- marks to illustrate the competitive performance of AlphaServer GS320 on both technical and commercial workloads.

Table 1 presents measured latencies for servicing a read request at various levels in the memory hierarchy. We report two sets of latencies in each case: (i) latency for back-to-back dependent reads (representative of pointer-chasing), and (ii) effective latency for pipelined independent reads (representative of array accesses). The "3-hop" case shows a range of latencies because it includes "2.5-hop" cases where the owner or the reader processor are at the home node. One of the main reasons for the relatively high back-to-back latencies is the use of an older ASIC technology (circa 1997-98, 9.6ns cycle time) in our current implementation. The results in the table show while some of the remote latencies are high, the Alpha 21364's out-of-order issue capability and its support for multiple outstanding requests can substantially reduce the effective laten- cies (by approximately 5-7 times) in the case of independent read misses. These pipelined latencies can also be used to calculate sus- tained bandwidths (e.g., 64-bytes per 61ns for 1.05GB/s bandwidth to local memory).

We next compare the AlphaServer GS320 to two other 21264- based AlphaServer designs. The GS140 is a previous-generation bus-based snoopy design which supports up to 14 processors. The ES40 is a switch-based snoopy design which supports up to 4 pro- cessors. We use 4-processor configurations for all three systems (processor frequencies are different). Table 2 shows the dependent read latency for L2 cache hits as measured from one processor. We consider two cases: (i) the other three processors are idle, and (ii) the other three processors are actively issuing misses. As the table shows, activity by the other three processors can lead to a major degradation in the L2 hit latency (2-3 times longer) as observed by the fourth processor in snoopy-based designs. The primary reason for this is that all system transactions must be snooped by the L2 caches, causing the L2 to be busy more often (especially in the ES40 which does not use duplicate tags). The AlphaServer GS320 does not suffer from this because it uses a directory-based protocol which significantly reduces the number of requests that are forwar- ded to each L2. Given the importance of L2 hit latencies in com- mercial workloads such as transaction processing [4], the use of ef- ficient directory-based protocols (instead of snooping) can provide benefits even for systems with a small number of processors.

Table 3 presents effective latencies for write (read-exclusive) operations while varying whether the home is local or remote and

Table 3: Effective latency for write operations.

Case	Pipelined Writes	Writes Separated by Memory Barriers
Local Home, No Sharers	58ns	387ns
Local Home, Remote Sharers	66ns	851ns
Remote Home, No Sharers	135ns	1192ns
Remote Home, Remote Sharers	148ns	1257ns

Table 4: Serialization latency for conflicting writes to the same line.

Case	Serialization Latency
1 QBB, 4 procs	138ns
8 QBBs, 1 proc/QBB	564ns

whether there are any sharers. We report two sets of latencies in each case: (i) effective latency for pipelined writes, and (ii) latency for writes ordered through memory barriers (includes memory bar- rier latency: 24ns minimum). The pipelined latencies for the local home and remote home are close to the pipelined local and 2-hop read latencies in Table 1. The latency impact of sending invalida- tions (due to presence of sharers) is small since our protocol does not use invalidation acknowledgements. Furthermore, these laten-cies are independent of the number of sharers since our protocol uses multicast. The latencies for writes separated by memory bar- riers are slightly larger than the local and 2-hop dependent read latencies, partly due to the cost of the memory barrier being in- cluded in this latency. Local home writes with remote sharers take substantially longer than with no sharers because the memory bar- rier must wait for the commit event to come back from the global switch (Section 4). Finally, the sending of invalidates for remote home writes leads to little increase in the latency (again due to the lack of invalidation acknowledgements which would incur 3-hop latencies).

We next consider the impact of our low occupancy protocol design for handling conflicting writes to the same line. Figure 3 in Section 3 illustrates the behavior of our protocol for write serializa-tion. Table 4 shows the serialization latency for two scenarios. Our experiments have each processor in a tight loop doing a write fol- lowed by a memory barrier (latencies include memory barrier over- head). In steady state, our protocol continuously forwards writes to the current owner with no blocking/retrying at the directory and the forwarded writes are delayed at their target caches until the early request race is resolved. Therefore, we expect the serializ- ation of the writes to be approximately equal to a 1-hop latency in our system. The first scenario involves 4 processors on a single quad node. In this case, 4 processors are insufficient to generate sufficient throughput in the steady state to always steal the line from the previous owner as soon as it receives it. Therefore, our measured serialization latency includes some cache hits, making it smaller than the 1-hop intra-node latency (approx. 180-200ns) that we expected. The second scenario consists of 8 processors on separate QBBs. The 8 processors lead to the expected steady state behavior and the measured serialization latency of 564ns is indeed approximately half of our 2-hop write latency (from Table 3). In comparison, protocols that depend on the use of NAKs/retries to resolve such races [24, 27, 28] would have a best case serialization latency of 2-hops (with higher possible latencies based on the tim- ing of retries), and lead to substantially more traffic and resource occupancies due to the large number of in-flight NAK and retry messages.

Finally, Table 5 presents some measurements to illustrate the impact of separating the commit component and generating early commits. The back-to-back dependent read latency measures the time it takes for the data reply component to reach the processor, while the latency with memory barriers measures the time for the commit component. The intrinsic overhead of a memory barrier is 25-50ns and is included in the latter measurement. The 2-hop measurements clearly illustrate the benefit of separating the commit component from the data reply component. The difference in the latency for the two components is 205-230ns (adjusted for the memory barrier overhead), and keeping the data and commit com- ponents together would cause this extra latency to be incurred by all read misses. Furthermore, this extra latency could increase if the inbound path for the commit is occupied by other forwarded requests. The 3-hop measurements illustrate the benefit of generat- ing early commits in our design (the 2-hop case does not generate an early commit in our design). We see the difference in latency for the data reply and commit components in the 3-hop case is ap- proximately 0-25ns (adjusted for memory barrier overhead), which is much smaller than the 205-230ns separation (of the 2-hop case) we would likely observe without early commits. In fact, the early commit is likely to arrive at the processor before the data reply component in the 3-hop case, but the Alpha 21264 waits for both components to arrive before proceeding past a memory barrier.

Table 5: Impact of separating the commit component and generat- ing early commits.

Case	Back-to-Back Dependent Reads	Reads Separated by Memory Barriers
2-hop, Clean	960ns	1215ns
3-hop, Dirty	1478ns	1529ns

The AlphaServer GS320 system provides leadership perform- ance on many technical workloads. The result on Linpack is 30 GFlops/s on a 32-CPU system. The McCalpin STREAM (COPY) bandwidth is 18.4 GB/s with 32 CPUs. The AlphaServer GS320 also provides competitive performance on commercial workloads. Our system supports 2720 users (1.9s response time) on the SAP R4 two-tier client/server benchmark (audited result [21]). In com- parison, the IBM RS/6000 S80 24-CPU system supports only 1708 users (1.98s response time). On the TPC-H benchmark (300GB database), the AlphaServer GS320 reports audited results of 4952 QphH [22]. This compares to a 64-CPU IBM NUMA-Q system at 7334 QphH and a 32-CPU HP9000 V2500 system at 3714 QphH. Our early audited result for TPC-C of 122K tpmC with 32 pro- cessors [37] is in between the IBM RS/6000 S80 (24-CPU) result of 135K tpmC and the Sun Enterprise 10000 (64-CPU) result of 115K tpmC [36]. However, this result has been withdrawn in anticipa- tion of a better result after further tuning. Finally, the AlphaServer GS320 achieves a user count of 11,200 on the single system Or- acle Applications standard benchmark, compared to 14,000 users for the IBM RS/6000 S80.

7 Discussion and Related Work

The AlphaServer GS320 architecture incorporates an interesting mix of techniques for improving directory-based protocols and in- corporates a couple of novel optimizations for efficiently imple- menting consistency models. Even though cache coherence and consistency have been extremely active areas of investigation for the past decade, by far the majority of this work has focused on large-scale designs. However, there is very little market demand for such large systems, primarily due to the lack of scalable soft- ware and fault-containment mechanisms. Virtually all the tech- niques used in the

AlphaServer GS320 were initially inspired by targeting medium-scale designs and the desire to exploit their lim- ited size. As it turns out however, several of these techniques are directly applicable to larger-scale directory-based and smaller-scale snoopy-based designs (see Sections 3 and 4). Much of the related work has already been referenced in earlier sections. This section presents a brief comparison with a couple of contemporary designs, and discusses other previous work pertinent to coherence protocols and consistency model implementations.

We briefly compare the AlphaServer GS320 to the Sun Enter- prise 10000 [10, 34] and the SGI Origin 2000 [27], which rep-
resent the current state-of-the-art in snoopy-based and directory- based designs respectively. Hristea and Lenoski [20] provide de- tailed latency and bandwidth measurements on the Sun and SGI systems. The sustained bandwidth per processor is 195MB/s for Sun and 554MB/s for SGI, compared to over 1GB/s in our sys- tem. The peak bisection bandwidths for 32-CPU configurations are: about 6GB/s for Sun (10.6GB/s for 64-CPUs) [10], 6.2GB/s (12.5GB/s with Xpress links or with 64-CPUs) [32], and 12.8GB/s for our design. The local memory latencies (back-to-back depend- ent reads) are 560ns and 472ns for Sun and SGI, compared to 327ns in our design. Local dirty latencies are 742ns and 1036ns for Sun and SGI, compared to 564ns in the AlphaServer GS320. Our 2-hop and 3-hop read latencies are comparable to the SGI for a similar size system (results provided in [20] are for 4-6 CPU SGI configurations). However, our design shines with respect to pipelined independent reads.

As mentioned earlier, there has been little work on efficient pro- tocol designs for medium-scale systems. Bilir et al. [7] propose the idea of selectively using multicast snooping within a directory-based protocol as a possible mechanism for reducing 3-hop transac- tion latencies. However, the combination of snooping within a dir- ectory scheme can lead to a complex protocol with higher message counts and resource occupancies. Hagersten and Koster [17] de- scribe the implementation of the Sun Wildfire prototype which con- nects up to four large snoopy-based SMP nodes using a directory- based protocol. Given the limit of four nodes, the designers opt for simplifying the protocol at the cost of efficiency and performance. Their protocol uses extra messages, along with blocking at the dir- ectory, to eliminate the possibility of all races: (i) writebacks use a three-phase protocol to first get permission from the home node before sending the data (eliminates late request race), and (ii) three- hop transactions are augmented with extra messages to inform the home when the requester receives its reply (eliminates early request race). The above design choice leads to higher occupancy and mes- sage counts compared to typical directory protocols. Finally, the prototype incurs high memory latencies: 1762ns for a 2-hop clean read and 2150ns for a 3-hop read.

While some of the techniques used in our protocol are not novel and have been previously used in other protocols, we believe that our protocol embodies a unique combination of these techniques that leads to an efficient low-occupancy design. For example, the Scalable Coherent Interface (SCI) protocol [16] also does not re- sort to NAKs and retries. However, its uses a strict request-reply protocol which incurs an extra hop for dirty remote misses. Further- more, its linked list directory structure leads to substantial design complexity and can also result in long invalidation latencies. In contrast, our protocol uses a centralized directory, and exploits an extra network lane (i.e., total of 3 lanes) to avoid using a strict request-reply scheme.

There has been much work on more efficient implementation of consistency models. The technique of *hardware prefetching* from the instruction window [14] issues non-binding prefetches for memory operations whose addresses are known, and yet are blocked due to consistency constraints. *Speculative load execu- tion* [14] increases the benefits of prefetching by allowing the re- turn value of the load to be consumed early. The latter technique requires hardware support for detecting violations of ordering requirements due to early consumption of values and for recovering from such violations. Violations are detected by monitoring coher- ence requests (and cache replacements) for the lines accessed by outstanding speculative loads. The recovery mechanism is similar to that used for branch mispredictions and exceptions. Both of the above techniques are implemented in a number of commercial microprocessors (e.g., MIPS R10000, and various implementations of HP PA-RISC and Intel Pentium processors). More recently, Gniady et al. [15] have proposed to extend the speculative load technique to apply to stores as well. However, this idea leads to a number of complexities arising from the need to perform speculative stores in the cache hierarchy without making them visible to other pro- cessors and to roll back such stores in case a violation is detected.

The early commit technique implemented in the AlphaServer GS320 applies to both loads and stores, does not depend on any mechanisms for detecting violations and rolling back, and is com- plementary to the above speculative techniques. In fact, there is po- tential for synergy from combining these techniques. For example, the size of speculative resources (e.g., the speculative load buf- fer [14]) may be reduced since an operation can be considered non- speculative as soon as the early commits for previous memory op- erations are received (i.e., instead of waiting for the longer latency data replies).

The early acknowledgement of invalidations (Section 4.1) is re- lated to to Afek et al.'s lazy caching [1, 2], Dubois et al.'s delayed consistency [11], and Landin et al.'s race-free network [26] ideas. However, these ideas had various limitations with respect to applic- ability to a wide range of designs, efficiency of implementation, and (in some cases) correctness. Gharachorloo's thesis (Sections 5.4-5.5) [13] provides a generalization of this technique that allevi- ates the above limitations.

Finally, it would be interesting to further isolate the perform- ance effects of the various coherence and consistency optimizations used in the AlphaServer GS320. Unfortunately, using the hardware implementation for this purpose is extremely difficult since we do not have the ability to selectively turn on and off various optimiza- tions.

8 Concluding Remarks

While much of the shared-memory multiprocessor research in the past decade has focused on large-scale systems, the high-end server market is primarily characterized by systems with at most 32 to 64 processors. Even though building systems with more processors is quite feasible from a hardware standpoint, the market demand for such systems is extremely limited due to the lack of scalable applications and operating systems and compelling fault-containment solutions.

This paper described the architecture of the AlphaServer GS320 which is targeted at medium-scale multiprocessing with 32 to 64 processors. Our design incorporates a number of innovative tech- niques for improving directory-based protocols and efficiently im- plementing consistency models. Our coherence protocol exhibits lower occupancy and lower message counts compared to previous designs, and naturally lends itself to elegant solutions for deadlock, livelock, starvation, and fairness. Our design also includes a couple of novel techniques for efficiently supporting memory ordering. These techniques allow a memory operation to be considered com- plete from an ordering perspective well before its data reply is formulated, and also allow for quicker delivery of the data reply on the inbound path to the requesting processor. The above tech- niques were all initially inspired by eliminating the requirement for scalability in our design, allowing us to consider solutions which may have otherwise been overlooked [12]. As it turns out however, several of these techniques are directly applicable to larger-scale directory-based and smaller-scale snoopy-based designs.

Technology has changed quite dramatically in the past four years since we began the design of the AlphaServer GS320. The next-generation AlphaServer is based on the Alpha 21364 which integrates a 1GHz 21264 core, two levels of caches, memory con- trollers, coherence hardware, and network routers all on a single die [3, 6]. Aggressive chip-level integration is also being employed in designs based on single chip multiprocessors (CMP) [5, 18]. The protocol design ideas explored in the AlphaServer GS320 have already influenced our more recent designs within Compaq, including the Alpha 21364 [3] and Piranha [5]. Furthermore, the memory consistency implementation techniques developed for the AlphaServer GS320 are especially well-suited for future CMP designs given the tight-coupling among on-chip processors and the hierarchical nature of systems built from CMP nodes.

Acknowledgments

The AlphaServer GS320 project would not have been possible without the participation of hundreds of Compaq employees whom we would like to thank for their contributions. Of special note was the leadership and vision of Dave Fenwick, the system architect. Early in the design, many issues were resolved between the sys- tem and microprocessor teams through the ”EV6 Systems Partners Forum” and especially through contributions by Jim Keller and Bob Stewart. We would like to thank participants in several internal re- views of the protocol, and in particular contributions by Dick Sites, Chuck Thacker, and Raj Ramanujan. A significant amount of work was also done by the formal protocol verification team led by Leslie Lamport with contributions by Yuan Yu and Mark Tuttle. Finally, we thank the anonymous reviewers for their comments.

References

[1] Y. Afek, G. Brown, and M. Merritt. A lazy cache algorithm. In *Symposium on Parallel Algorithms and Architectures*, pages 209–222, June 1989.

[2] Y. Afek, G. Brown, and M. Merritt. Lazy caching. *ACM Transac- tions on Programming Languages and Systems*, 15(1):182–205, Janu- ary 1993.

[3] P. Bannon. Alpha 21364: A Scalable Single-Chip SMP. In *Microprocessor Forum '98*, October 1998. (also available at http://www.digital.com/alpha-oem/microprocessorforum.htm).

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

[5] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *Proceedings of the 27th International Symposium on Computer Ar- chitecture*, June 2000.

[6] L. A. Barroso, K. Gharachorloo, A. Nowatzyk, and B. Verghese. Im- pact of Chip-Level Integration on Performance of OLTP Workloads. In *Proceedings of the 6th International Symposium on High Perform- ance Computer Architecture*, January 2000.

[7] 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 Inter- national Symposium on Computer Architecture*, May 1999.

[8] E. Bugnion, S. Devine, and M. Rosenblum. Disco: Running Com- modity Operating Systems on Scalable Multiprocessors. In *Proceed- ings of the 16th ACM Symposium on Operating Systems Principles*, Oct. 1997.

[9] L. M. Censier and P. Feautrier. A new solution to coherence prob- lems in multicache systems. *IEEE Transactions on Computers*, C- 27(12):1112–1118, December 1978.

[10] A. Charlesworth. STARFIRE: Extending the SMP Envelope. *Micro*, pages 39–49, January/February 1998.

[11] M. Dubois, J. Wang, L. Barroso, K. Lee, and Y. Chen. Delayed con- sistency and its effects on the miss rate of parallel programs. In *Pro- ceedings of Supercomputing '91*, pages 197–206, 1991.

[12] J. Emer. Relaxing Constraints: Thoughts on the Evolution of Computer Architecture. Keynote Speech at the 6th International Symposium on High Performance Computer Architecture, Toulouse, France. January 10, 2000.

[13] K. Gharachorloo. *Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, Stanford University, December 1995.

[14] K. Gharachorloo, A. Gupta, and J. Hennessy. Two techniques to en- hance the performance of memory consistency models. In *Proceed- ings of the 1991 International Conference on Parallel Processing*, pages I:355–364, August 1991.

[15] C. Gniady, B. Falsafi, and T. Vijaykumar. Is SC + ILP = RC? In *Proceedings of the 26th International Symposium on Computer Ar- chitecture*, May 1999.

[16] D. B. Gustavson and Q. Li. The scalable coherent interface (sci). *IEEE Communications Magazine*, pages 52–63, August 1996.

[17] E. Hagersten and M. Koster. WildFire: A Scalable Path for SMPs. In *Proceedingsof the 5th International Symposium on High Performance Computer Architecture*, January 1999.

[18] L. Hammond, B. Nayfeh, and K. Olukotun. A Single-Chip Multipro- cessor. *Computer*, 30(9):79–85, September 1997.

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

[20] C. Hristea, D. Lenoski, and J. Keen. Measuring memory hierarchy performance of cache-coherent multiprocessors using micro bench- marks. In *Proceedings of Supercomputing '97*, 1997.

[21] http://www.ideasinternational.com/benchmark/sap/sap2tcs.html.

[22] http://www.ideasinternational.com/benchmark/tpc/tpch.html.

[23] R. E. Kessler. The Alpha 21264 Microprocessor. *Micro*, pages 24–36, March/April 1999.

[24] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Ghara- chorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta,
M. Rosenblum, and J. Hennessy. The Stanford FLASH multipro- cessor. In *Proceedings of the 21st International Symposium on Com- puter Architecture*, pages 302–313, April 1994.

[25] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[26] A. Landin, E. Hagersten, and S. Haridi. Race-free interconnectionnet- works and multiprocessor consistency. In *Proceedingsof the 18th An- nual International Symposium on Computer Architecture*, pages 27– 30, May 1991.

[27] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scal- able Server. In *Proceedings of the 24rd International Symposium on Computer Architecture*, June 1997.

[28] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. L. Hen- nessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Sym- posium on Computer Architecture*, pages 94–105, May 1990.

[29] D. J. Scales and K. Gharachorloo. Design and Performance of the Shasta Distributed Shared Memory Protocol. In *Proceedings of the 11th ACM International Conference on Supercomputing*, July 1997.

[30] D. J. Scales and K. Gharachorloo. Towards Transparent and Effi- cient Software Distributed Shared Memory. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Oct. 1997.

[31] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A Low- Overhead Software-Only Approach to Fine-Grain Shared Memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 174–185, Oct. 1996.

[32] http://www.sgi.com/Products/PDF/2500.pdf.

[33] R. L. Sites and R. T. Witek, editors. *Alpha Architecture Reference Manual*. Digital Press, 1998. Third Edition.

[34] Sun Microsystems. Sun Enterprise 10000 Server - Technical White Paper. http://www.sun.com/servers/white-papers/E10000.pdf.

[35] D. Teodosiu, J. Baxter, K. Govil, J. Chapin, M. Rosenblum, and
M. Horowitz. Hardware fault containment in scalable shared-memory multiprocessors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.

[36] http://www.tpc.org.

[37] http://www.tpc.org/new result/c-withdrawn-results.idc.

[38] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating system support for improving data locality on CC-NUMA computer servers. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languagesand Operating Sys tems*, pages 279–289, October 1996.