FEBRUARY 1997

WRL
Research Report 97/3

Fine-Grain Software Distributed Shared Memory on SMP Clusters

Daniel J. Scales
Kourosh Gharachorloo
Anshu Aggarwal

digital **Western Research Laboratory** 250 University Avenue Palo Alto, California 94301 USA

The Western Research Laboratory (WRL) is a computer systems research group that was founded by Digital Equipment Corporation in 1982. Our focus is computer science research relevant to the design and application of high performance scientific computers. We test our ideas by designing, building, and using real systems. The systems we build are research prototypes; they are not intended to become products.

There are two other research laboratories located in Palo Alto, the Network Systems Lab (NSL) and the Systems Research Center (SRC). Another Digital research group is located in Cambridge, Massachusetts (CRL).

Our research is directed towards mainstream high-performance computer systems. Our prototypes are intended to foreshadow the future computing environments used by many Digital customers. The long-term goal of WRL is to aid and accelerate the development of high-performance uni- and multi-processors. The research projects within WRL will address various aspects of high-performance computing.

We believe that significant advances in computer systems do not come from any single technological advance. Technologies, both hardware and software, do not all advance at the same pace. System design is the art of composing systems which use each level of technology in an appropriate balance. A major advance in overall system performance will require reexamination of all aspects of the system.

We do work in the design, fabrication and packaging of hardware; language processing and scaling issues in system software design; and the exploration of new applications areas that are opening up with the advent of higher performance systems. Researchers at WRL cooperate closely and move freely among the various levels of system design. This allows us to explore a wide range of tradeoffs to meet system goals.

We publish the results of our work in a variety of journals, conferences, research reports, and technical notes. This document is a research report. Research reports are normally accounts of completed research and may include material from earlier technical notes. We use technical notes for rapid distribution of technical material; usually this represents research in progress.

Research reports and technical notes may be ordered from us. You may mail your order to:

Technical Report Distribution
DEC Western Research Laboratory, WRL-2
250 University Avenue
Palo Alto, California 94301 USA

Reports and technical notes may also be ordered by electronic mail. Use one of the following addresses:

Digital E-net:	`JOVE::WRL-TECHREPORTS`
Internet:	`WRL-Techreports@decwrl.pa.dec.com`
UUCP:	`decpa!wrl-techreports`

To obtain more details on ordering by electronic mail, send a message to one of these addresses with the word ‘‘`help`’’ in the Subject line; you will receive detailed instructions.

Reports and technical notes may also be accessed via the World Wide Web: `http://www.research.digital.com/wrl/home.html`.

Fine-Grain Software Distributed Shared Memory on SMP Clusters

Daniel J. Scales
Kourosh Gharachorloo
Anshu Aggarwal

February 1997

d i g i t a l **Western Research Laboratory** 250 University Avenue Palo Alto, California 94301 USA

Abstract

Commercial SMP nodes are an attractive building block for software distributed shared memory systems. The advantages of using SMP nodes include fast communication among processors within the same SMP node and potential gains from clustering where remote data fetched by one processor is used by other processors on the same node. The widespread availability of SMP servers with small numbers of processors has led several researchers to consider their use as building blocks for Shared Virtual Memory (SVM) systems. These systems exploit the SMP cache-coherence hardware to support fine-grain communication within a node, and use software to support communication across nodes at a coarser page-size granularity. Our goal is to explore the use of SMP nodes in the context of the Shasta system. A unique aspect of Shasta compared to most other software distributed shared memory systems is that shared data can be kept coherent at a fine granularity. Shasta implements this coherence by inserting inline code that checks the cache state of shared data before each load or store. In addition, Shasta allows the coherence granularity to be varied across different shared data structures in a single application. This approach alleviates potential inefficiences that arise from the fixed large granularity of communication typical in most software systems.

This paper describes a major extension to the Shasta system that supports fine-grain shared memory across SMP nodes. Allowing processors to efficiently share memory within the same SMP is complicated by race conditions that arise because the inline state check is non-atomic with respect to the actual load or store of shared data. We present a novel and efficient protocol that avoids such race conditions without the use of costly synchronization in the inline checking code. The above protocol is fully functional and runs on a prototype cluster of Alpha multiprocessors connected through Digital's Memory Channel network. To characterize the benefits of using SMP nodes in the context of Shasta, we also present detailed performance results for nine SPLASH-2 applications running on this cluster.

1 Introduction

Clusters of workstations or symmetric multiprocessors (SMPs) are potentially powerful platforms for executing parallel applications. In order to simplify the programming of such clusters, researchers have developed a number of software distributed shared memory (DSM) systems that support a shared address space across nodes through a layer of software. The most common approach, called Shared Virtual Memory (SVM), uses the virtual memory hardware to detect access to data that is not available locally [2, 9, 10]. These systems communicate data and maintain coherence at a fixed granularity equal to the size of a virtual page. As an alternative, a few systems have explored the feasibility of supporting fine-grain sharing of data entirely in software [12, 14]. Fine-grain access to shared data is important to reduce false sharing and the transmission of unneeded data, both of which are potential problems in systems with large coherence granularities. In addition, by supporting variable coherence granularities across different shared data structures in the same application, systems such as Shasta [12] can selectively exploit any potential gains from larger communication granularities for specific shared data. Fine-grain access to shared data is supported by inserting code in an application executable before loads and stores that checks if the data being accessed is available locally in the appropriate state. Recent work in the context of the Shasta system has shown that the cost of the inline checks can be minimized by applying appropriate optimizations [12], making this approach a viable alternative to SVM systems.

Commercial SMP nodes are an attractive alternative to uniprocessors as a building block for software DSM systems. At a minimum, the faster communication interconnect within an SMP can be used for all intra-node messages. Furthermore, the hardware support for cache coherence can be used to allow processors within an SMP to share application memory (and software cache state), thus eliminating software intervention for intra-node data sharing. Sharing memory also provides a clustering effect, whereby remote data fetched by one processor may be readily used by other processors on the same node. Sharing other protocol-dependent data structures among processors on the same node may provide further gains. For example, in a directory-based protocol, sharing the directory state may eliminate the need for an internal message when the requester and home are located on the same node. In addition, SMP nodes provide the opportunity to handle incoming messages on any processor on a node for load-balancing purposes.

The widespread availability of commercial SMP servers with small numbers of processors has led several researchers to consider their use as building blocks for Shared Virtual Memory (SVM) systems [1, 3, 4, 8, 18]. These systems exploit the SMP cache coherence hardware to support fine-grain sharing within a node, and use a software protocol to support sharing across nodes at a coarser page-size granularity. Most of the above work is based on simulation studies. SoftFLASH is the only actual implementation based on commercial multiprocessor nodes [4].

The goal of this paper is to explore the benefits of SMP nodes, especially the sharing of application memory among processors, in the context of systems such as Shasta that support fine-grain sharing of data across nodes. Exploiting SMP nodes efficiently in this context is a non-trivial task. The primary difficulty arises from race conditions caused by the fact that the inline state check used to support fine-grain access control is non-atomic with respect to the actual load or store of shared data, since the two actions consist of multiple instructions. In contrast, the virtual memory hardware provides an atomic state check and data access in SVM systems. An example of the race condition that can arise is as follows. Assume processors P1 and P2 are on the same node, and that an exclusive copy of the data at address A is cached on this node. Assume P1 detects the exclusive state at its inline check for address A and proceeds to do a store to the address, while P2 is servicing a write request from another node. The undesirable race arises if P2 downgrades the data to an invalid state and reads the value of A before P1's store is complete and visible to P2. One possible solution is to add sufficient synchronization to ensure that P2 cannot downgrade the state and read the value of A in between the inline state check and store on P1. However, this solution results in a large increase in the checking overhead at every load or store to shared data, and may lead to an overall performance loss compared to a system that doesn't exploit sharing within each SMP node. Therefore, a more efficient solution is required. In addition to the above races, there are other types of race conditions caused by multiple processors invoking protocol actions for the same address. Due to the lower frequency of protocol actions (as compared to inline checks), this latter category of races can be handled reasonably efficiently through careful synchronization.

This paper describes a major extension to the Shasta system that efficiently supports fine-grain shared memory across a cluster of SMP nodes. We present a novel and efficient solution that allows sharing of memory among processors on the same node, and avoids the race conditions described above without the use of costly synchronization in the inline checking code. Our overall solution involves the use of locking during protocol operations, and the use of explicit messages between processors on the same node for protocol operations that can lead to the race conditions involving the inline checks. Our protocol also maintains some per-processor state information in order to minimize the number of such intra-node messages.

The above protocol has been implemented on our prototype cluster and is fully functional. The cluster consists of four Alpha multiprocessors connected through the Memory Channel [6], with a total of sixteen processors. We present detailed performance results for nine SPLASH-2 applications running on the above cluster. Our new protocol is successful in reducing the parallel execution time of most of the SPLASH-2 applications. Although individual protocol operations are more expensive (due mainly to locking in the protocol code), overall performance improves significantly in most cases because of the reduced number of remote misses and protocol messages.

The following section describes the basic design of Shasta, including the inline state checks and the protocol that is invoked in case of a miss. Section 3 describes the extensions to the base Shasta protocol required to efficiently support shared memory across SMP nodes. We present detailed performance results in Section 4. Finally, we describe related work and conclude.

2 Basic Design of Shasta

In this section, we present an overview of the base Shasta system, which is described more fully in previous papers [11, 12]. Shasta divides the virtual address space of each processor into private and shared regions. Data in the shared region may be cached by multiple processors at the same time, with copies residing at the same virtual address on each processor. The base Shasta system adopts the memory model of the original SPLASH applications [15]: data that is dynamically allocated is shared, but all static and stack data is private.

2.1 Cache Coherence Protocol

As in hardware cache-coherent multiprocessors, shared data in the Shasta system has three basic states:

- invalid - the data is not valid on this processor.
- shared - the data is valid on this processor, and other processors have copies of the data as well.
- exclusive - the data is valid on this processor, and no other processors have copies of this data.

Communication is required if a processor attempts to read data that is in the invalid state, or attempts to write data that is in the invalid or shared state. In this case, we say that there is a *shared miss*. Shasta inserts code which does a *shared miss check* on the data being referenced at loads and stores in the application executable.

As in hardware shared-memory systems, Shasta divides up the shared address space into ranges of memory, called *blocks*. All data within a block is in the same state and is always fetched and kept coherent as a unit. A unique aspect of the Shasta system is that the block size can be different for different ranges of the shared address space (i.e., for different program data). To simplify the inline code, Shasta divides up the address space further into fixed-size ranges called *lines* and maintains state information for each line in a *state table*. The line size is configurable at compile time and is typically set to 64 or 128 bytes. Each block consists of a single line or an integer number of lines.

Coherence is maintained using a directory-based invalidation protocol. The protocol supports three types of requests: *read*, *read-exclusive*, and *exclusive* (or *upgrade*). A home processor is associated with each virtual page of shared data, and each processor maintains *directory* information for all blocks on pages that are assigned to it. The protocol maintains the notion of an *owner* processor for each block, which corresponds to the last processor that had an exclusive copy of the block. The directory information consists of two components: (i) a pointer to the current owner processor, and (ii) a full bit vector of the processors that are sharing the data. A read or read-exclusive request that arrives at the home is always forwarded to the current owner; as an optimization, the home processor serves read requests directly if it has a copy of the data.

Because of the high cost of handling messages via interrupts, messages from other processors are serviced through a polling mechanism. The Shasta system polls for incoming messages whenever the protocol waits for a reply and at every loop backedge. Polling is inexpensive (three instructions) on a Memory Channel cluster, because the Shasta implementation arranges for a single cachable location that can be tested to determine if a message has arrived. The use of polling also simplifies the inline miss checks, since the Shasta system ensures that there is no handling of messages between a shared miss check and the load or store that is being checked.

The Shasta protocol aggressively exploits the release consistency model [5] by emulating the behavior of a processor with non-blocking stores and a lockup-free cache. Information about a pending request for a line is maintained in an entry in a *miss table*. The protocol supports non-blocking stores by simply issuing a read-exclusive or exclusive request, recording where the store occurred in the miss entry, and continuing. This information allows the protocol to appropriately merge the reply data with the newly written data that is already in memory. The protocol also supports aggressive lockup-free behavior for lines that are in a pending state. Writes to a pending line are allowed to proceed by storing the newly written data into memory and adding the location of the new stores to the appropriate miss entry when the miss handling routine is invoked (due to the pending state).

2.2 Basic Shared Miss Check

Figure 1 shows Alpha assembly code that does a store miss check. This code first checks if the target address is in the shared memory range and if not, skips the remainder of the check. Otherwise, the code calculates the address of the state table entry corresponding to the target address and checks that the line containing the target address is in the exclusive state.

This code has been optimized in a number of ways. For example, the code does not save or restore registers. The Shasta system does live register analysis to find unused registers (labeled `rx` and `ry` in the figure) at the point where it inserts the miss check. In addition, the code has been scheduled for efficient execution on a pipelined and superscalar Alpha processor. Shasta does not need to check accesses to non-shared (i.e., private) data, which includes all stack and static data in the current implementation. Therefore, a load or store whose base register uses the stack pointer (SP) or global pointer (GP) register, or is calculated using the contents of the SP or GP, is not checked.

2.3 Optimizations to Reduce Check Overhead

Despite the simple optimizations applied to the basic checks, the overhead of the miss checks can be significant for many applications, often approaching or exceeding 100% of the original sequential execution time. The Shasta

```
        lda       rx, offset(base)
        srl       rx, SHARED_HEAP_BITS, ry
        srl       rx, LINE_BITS, rx
        beq       ry, nomiss
        ldq_u     ry, 0(rx)
        extbl     ry, rx, ry
        beq       ry, nomiss

        ...call function to handle store miss

    nomiss:
        ... store instruction
```

Figure 1: Store miss check code.

system applies a number of more advanced optimizations that dramatically reduce this overhead to an average of about 20% (including polling overhead) across the SPLASH-2 applications [12]. The two most important optimizations are described below.

Whenever a line on a processor becomes invalid, the Shasta protocol stores a particular "invalid flag" value in each longword (4 bytes) of the line. The miss check code for a load can then just compare the value loaded with the flag value. If the loaded value is not equal to the flag value, the data must be valid and the application code can continue immediately. If the loaded value is equal to the flag value, then a miss routine is called that first does the normal range check and state table lookup. These checks distinguish an actual miss from a "false miss" (i.e., when the application data actually contains the flag value), and simply return back to the application code in the case of a false miss. Since false misses almost never occur in practice, the above technique can greatly reduce the load miss check overhead. In addition, the flag technique essentially combines the load of the data and the check of its state into a single atomic event, which is a useful property for our SMP implementation, as we discuss later.

Another important technique for reducing the overhead of miss checks is to batch together checks for multiple loads and stores. Suppose there are a sequence of loads and stores that are all relative to the same (unmodified) base register and the offsets (with respect to the base register) span a range whose length is less than or equal to the Shasta line size. These loads and stores can collectively touch at most two consecutive lines in memory. Therefore, if inline checks verify that these lines are in the correct state, then all the loads and stores can proceed without further checks. The batching technique also applies to interleaved loads and stores via multiple base registers. For each set of loads and stores that can be batched, the Shasta system generates code to check the state of the lines that may be referenced via each base register. A batch miss handling routine is called if any of the lines are not in the correct state.

3 Protocol Extensions for SMP Clusters

This section describes modifications to the basic Shasta protocol to exploit SMP nodes. We begin by motivating the use of SMP nodes in the context of Shasta. Section 3.2 describes the difficult race conditions that arise due to sharing protocol state and data within each SMP. We present our general solution for efficiently dealing with these races in Section 3.3. The last section presents further detail about the extended Shasta protocol.

3.1 Motivation for Exploiting SMP Nodes

In this section, we characterize the optimizations that are possible when using SMP nodes in the context of a Shasta-like protocol. The goal for most of these optimizations is to contain a significant amount of communication within each node in order to fully exploit the SMP hardware support for cache coherence and fast messaging. Some of the optimizations require sharing additional protocol state and data among processors on the same node. This leads to a potential trade-off, since the shared data structures typically require extra synchronization that may increase frequent protocol path lengths and can also lead to a more complex design. The possible optimizations are as follows:

Faster Intra-Node Messaging. An obvious optimization is to implement a faster messaging layer for processors on the same node by exploiting the lower-latency and higher-bandwidth interconnect in each SMP. This goal can be achieved by transmitting messages through message queues allocated in the cache-coherent shared memory on each node. This optimization leads to a distinction between *intra-node* (or *local*) and *inter-node* (or *remote*) messages.

Intra-Node Data Sharing without Software Protocol Intervention. Hardware support for cache coherence can also be used to support efficient sharing of application data between processors on the same node. Therefore, software intervention for misses along with explicit software messages can be limited to data sharing between processors on different nodes. In addition to sharing application data, this optimization requires the state table for application data to be shared among processors on the same node in order to allow software to detect cases where shared data is not available locally.

Eliminating Remote Requests. The sharing of application data and the state table also enables gains from clustering, whereby remote data fetched by one processor may be readily used by other processors on the same node without use of explicit messages. Effectively, the node's local memory is used as a shared cache for remote application data. This optimization eliminates remote requests if the data is available locally in the appropriate state, or turns a remote write request into the more efficient upgrade request if the local copy is in the shared state. Note that even if the data requested by one processor is not yet back, it is possible to filter requests for the same data from other processors on the same node. This latter optimization may require the sharing of protocol data structures that keep track of pending remote requests.

Eliminating Intra-Node Messages/Hops. The use of a directory-based protocol provides the opportunity to eliminate intra-node messages in two other cases. The first case is on an incoming remote read request to the home processor when the owner processor is on the same node. The home can trivially satisfy the request if the node has a clean copy, thus eliminating the need for an explicit message to the owner. The second case arises when the requester and home processors are colocated. By sharing the directory state, the requester can directly look up and modify directory information, thus avoiding an internal hop to the home processor. Note that the latter optimization occurs only if the request cannot be serviced within the local node.

Load-Balancing Incoming Requests from Other Nodes. Even though we have been assuming that messages are destined to specific processors on a node, sharing the incoming message queues from remote nodes provides the opportunity to load-balance (or schedule) the handling of remote messages on *any* processor at the destination node. Servicing a reply, or a request to the owner processor, at any processor on a node requires application data and the state table to be shared.[1] Servicing a request to the home by any processor on a node further requires sharing the directory state. A degenerate form of the above optimization involves dedicating one or more processors on the SMP for message or protocol processing only (as in Typhoon-0 [13]).

Our current implementation exploits all of the above optimizations except eliminating local messages when the requester and home are colocated and load-balancing the service of incoming requests. These optimizations are not

[1] In some protocols, the owner processor must always be consulted on certain protocol transactions. For these cases, handling the incoming request at a different processor may not provide any gain.

P1	**P2**
Read State	Write State = *invalid*
if *(state is exclusive)* then	Read A
Write A = *new value*	Write A = *invalid flag*
else *protocol miss code*	

(a) exclusive-to-invalid downgrade due to incoming write

P1	**P2**
Read State	Write State = *shared*
if *(state is exclusive*) then	Read A
Write A = *new value*	
else *protocol miss code*	

(b) exclusive-to-shared downgrade due to incoming read

P1	**P2**
Read State	Write State = *invalid*
if *(state is shared*) then	Write A = *invalid flag*
Read A	
else *protocol miss code*	

(c) shared-to-invalid downgrade due to incoming write

P1	**P2**
Read A	Write State = *invalid*
if *(value != invalid flag*) then	Write A = *invalid flag*
continue	
else *protocol miss code*	

(d) shared-to-invalid downgrade with invalid flag optimization

Figure 2: Examples of race conditions due to cache state downgrades.

used because we do not currently support the sharing of directory state or incoming remote message queues among colocated processors.

3.2 Difficult Race Conditions

The sharing of application data and protocol data structures among processors on the same SMP leads to numerous race conditions. Most of these race conditions can be handled reasonably efficiently by using sufficient locking in the protocol code. However, races involving the inline checking code cannot be handled efficiently in this way due to the high frequency of the required synchronization. Since the above sharing is required for exploiting the majority of the optimizations described in the previous section, it is critical to find an efficient way to deal with the resulting races. We describe this class of races in more detail below.

The difficult race conditions mentioned above are caused by the fact that the inline check of the state table entry is non-atomic with respect to the corresponding load or store to application data. This problem is best illustrated through the examples shown in Figure 2. For all the examples, assume that P1 and P2 are on the same node, P1 is doing an inline check followed by a load or store to the block A, and P2 is running protocol code to service an incoming request for the block from a remote node. All of the examples involve a downgrade of the state of the block due to the handling of the remote request. Figure 2(a) shows the scenario where A is initially cached in exclusive mode at this node, P1 is executing an application write to A, and P2 is handling an incoming write (or read-exclusive) request to A. As shown, servicing the incoming write requires P2 to read the latest contents of A to send back to the requester and to set the state of A at this node to invalid. The undesirable race occurs if P1 detects the initial exclusive state and proceeds to do the application store to A, yet P2 does its read before P1's write to A is complete. The coherence protocol has failed, since the application write on P1 will not be reflected at the new owner's copy. This race can occur even if P2 reads A before updating the state. Note that the problem would disappear if the state check and application store are somehow made to appear atomic with respect to operations on P2. Figure 2(b) shows a similar example where P2 services an incoming read. Again, if P1 detects the initial exclusive state, it is possible for P2 to read the old value of A, and an incorrect copy will be sent to the new sharer.

Figure 2(c) shows the scenario where A is initially in a shared state, P1 performs an application read, and P2 serves an incoming write (or invalidate) request. Even if the state check on P1 does not use the invalid flag optimization (which may not be applicable to all loads), P2 must still write the invalid flag value in addition to setting the state to invalid, since other application loads may use the optimization. The undesirable race occurs if P1 detects the initial shared state and proceeds to do the load, yet P2 writes the invalid flag value before the load on P1 completes. This race results in the wrong value (the flag value) being returned to the application code on P1. Again, changing the order of operations on P2 does not alleviate the race. The above problem would not occur in protocols that do not exploit the invalid flag optimization, since there would be no need for P2 to write the invalid flag to memory. Figure 2(d) shows a similar example except that the invalid flag optimization is used for checking the load on P1. Interestingly, this case does not lead to a race, since the invalid flag optimization makes the state check and load on P1 appear atomic with respect to operations on P2.

Assuming that the inline state checks precede the actual load or store of application data (as in the above examples), the difficult race conditions all arise when the local cache state of a block is downgraded because of servicing a request from another node. The cases for a local state *downgrade* are as follows:

- exclusive-to-invalid state downgrade, caused by an incoming write request,
- exclusive-to-shared state downgrade, caused by an incoming read request, and
- shared-to-invalid state downgrade, caused by an incoming write or invalidate request.

These cases directly correspond to the examples shown in Figure 2. The only remote request that does not lead to a downgrade is an incoming read request when the local state is shared.

There are several possible solutions for dealing with the above race conditions. One solution is to disallow the sharing of application and protocol data among processors. However, this solution forgoes all of the optimization enabled by SMP nodes (covered in Section 3.1) except for faster intra-node messaging. Another possible solution is to use sufficient synchronization (e.g., lock or flag synchronization) to enforce atomicity of the inline check and load or store sequence with respect to downgrade protocol actions on other processors. However, the overhead of such synchronization is prohibitive due to the high frequency of the inline checks that must be protected. SMP nodes that support a relaxed memory model further increase synchronization costs, due to the need for expensive fence instructions (called memory barriers on the Alpha [16]) at synchronization points that enforce ordering of memory operations. These fences take a minimum of ten processor cycles for the Alpha multiprocessors used in our experiments.

3.3 Our General Solution to Race Conditions

The prohibitive cost of synchronization and fence instructions heavily favors solutions that avoid the use of these instructions in the frequent inline state checks. Our overall solution depends on the selective use of explicit messages to deal with the state downgrade transitions described in the previous section, along with polling which is appropriately inserted to avoid the handling of such messages between the inline check and the corresponding load or store. Explicit locking is used to avoid race conditions between two protocol operations on the same block, as described in Section 3.4.2.

Consider the example in Figure 2(a) again. Our solution downgrades the state of A and sends explicit *downgrade messages* to other local processors that may be accessing the same block. Other protocol actions (e.g., reading or writing application data) required for servicing the incoming request are delayed until downgrade messages are received and serviced by all recipient processors. Downgrade messages are handled using the basic message handling layer in Shasta, which is based on polling. As described in Section 2.1, Shasta only polls whenever the protocol waits for a reply and at every loop backedge, so messages are never handled between a successful inline state check and the corresponding application load or store instruction. Therefore, any successful inline checks that occur before the reception of the downgrade message are guaranteed to have completed their corresponding load or store to application data. Similarly, inline checks that occur after the reception of the downgrade message are guaranteed to see the state that reflects the downgrade. The above two properties eliminate the race conditions that were described in the previous section. Downgrade messages effectively enforce an explicit synchronization across processors when necessary, thus avoiding the need for synchronizing every inline check.

The main source of inefficiency in the above scheme arises from the fact that every incoming request that causes a downgrade in the state of a block requires downgrade messages to be sent to *all other* processors on that node. The number of downgrade messages can be greatly reduced by maintaining a *private state table* for each processor in addition to the *shared state table* for the node. This private state table is used to keep track of whether a processor has actually accessed the block and may therefore need to be downgraded. In addition to enabling selective downgrade messages, the use of a private state table efficiently solves some subtle race conditions that arise with SMP nodes that support a relaxed memory model. For example, if we reference the shared state table in the inline state check for a load (as in Figure 2(c)), we would need to insert a fence instruction (memory barrier in Alpha) between the state check and actual load to ensure that the load would get an up-to-date value if another local processor changes the state of the block from invalid to shared or exclusive after fetching a copy of the block. The use of the private state table eliminates the need for such a fence instruction in the inline code.

Inline checks now read the processor's private state table (without synchronization or fence instruction) instead of the shared state table. The private state table is updated by protocol miss handling code when it detects that a block is available locally based on the shared state table. For example, consider the scenario when an exclusive copy of a block is fetched by processor P1. Both the shared state table and P1's private state table indicate an exclusive state. However, the private state table for other processors on the node still indicate an invalid state. When another processor attempts to read (write) the block, the inline check (based on that processor's private state table) invokes protocol miss code that upgrades the private state to shared (exclusive). Loads that use the invalid flag optimization do not have to do a state check and therefore can succeed without upgrading the private state.

The private state table for a given processor may be examined by other processors on the node to determine whether the processor has accessed a given block.[2] Downgrade messages can therefore selectively be sent to only those processors that have actually accessed the block. A downgrade initiated by an incoming read message requires downgrade messages to be sent to every local processor that shows an exclusive private state, while a downgrade initiated by an incoming write message requires downgrade messages to be sent to every local processor that shows a shared or exclusive private state. As we will see in Section 4.4, this optimization allows a large number of incoming requests to be serviced with zero or at most one downgrade message generated. Each processor downgrades its private state appropriately when it receives a downgrade message. Section 5 describes the similarities and differences of the above approach to related TLB shootdown techniques used in SVM systems such as SoftFLASH.

[2] Modifications of the private state table by its owner processor and reads of other processor's private state table occur only within the protocol code (i.e., not in the inline code) and are protected by the same synchronization used for the shared state table.

3.4 Implementation Details

In this section, we describe in detail the changes that have been made to Shasta to run efficiently on clusters of SMP nodes. We refer to this version of Shasta as SMP-Shasta and refer to the original implementation (which uses message passing between all processors) as Base-Shasta.

3.4.1 Changes to the Inline Checking Code

There are two main changes to the inline checking code. First, checks of floating-point loads via the invalid flag technique must be made atomic. For a floating-point load, the miss check code actually does the compare in an integer register (because this is faster than a floating-point compare). Because current Alpha processors do not have an instruction for transferring a value from a floating-point to an integer register, the Base-Shasta system inserts an integer load to the same target address as the floating-point load. However, the use of the two loads makes this technique non-atomic. The SMP-Shasta system therefore inserts code to store the floating-point register value onto the stack and then load the value into an integer register. This method adds several cycles to the cost of checking a floating-point load (but the new cost is still cheaper than doing a check via the state table). The recently announced Alpha 21264 processor supports instructions fnor transferring values between integer and floating-point registers that will reduce the cost of this method.

Second, the batch checks in SMP-Shasta must always use the private state table. In Base-Shasta, we use the invalid flag technique in doing a check on a batch range which includes only loads. However, the batched loads are not executed atomically with respect to the batch checks. Therefore, all batch checking code in SMP-Shasta must access the private state table (as described in the previous section) instead of checking for the invalid flag. This change to the inline code typically causes the largest increase in checking overhead relative to Base-Shasta, since most of the commonly executed code in the SPLASH-2 applications makes heavy use of batching.

3.4.2 Changes to the Basic Protocol

One of the main changes to the Shasta protocol is the use of locking to eliminate race conditions between protocol operations executed by processors on the same node. First, all protocol operations on a block hold a lock on that block for the entire operation. For blocks that consist of multiple lines, only the lock for the first line of the block is acquired (the block size for a data structure does not change during a run). We currently use a fixed set of line locks and associate each line with a particular lock using a hashing function. Second, the protocol uses locks in accessing buckets of the miss table, since miss table entries for pending requests are shared among all the processors. Modifications to miss table entries are protected by the line locks, since the miss entries apply to specific lines. There are only a few other locks used by the protocol, and all are used only in unusual cases. Therefore, processors that are accessing different blocks should not encounter contention on locks (assuming that lock hashing function works well).

The basic Shasta protocol remains largely the same in SMP-Shasta. Requests that cannot be satisfied locally are sent to the home processor for the requested block. The home processor maintains the identity of the current owner processor and the processors that are sharing the block. The home is only aware of the sharing by the one processor on a node that initially requested the data. This property is important to make sure that protocol requests for a block are serialized at one processor on a node. A read or read-exclusive request that arrives at the home is forwarded to the current owner; this forwarding is avoided if the home and owner are on the same node or the home node has a copy of the data.

The SMP-Shasta protocol merges requests by multiple processors on a node for the same block. The first processor to require the block sends a message to fetch the block and sets the shared state of the block to a pending state. Other processors that require the block will enter the protocol, but will not send another request, since the block is already in a pending state. If multiple processors stall on a block (e.g. due to loads), then the protocol ensures that all the stalled processors resume execution when the data returns. If multiple processors attempt to store in a block, the locations of all the stores are merged in the miss table entry for the block and the processors can proceed without stalling. When a processor receives a block that it has requested, it updates the entry in the shared state table and its private table to the appropriate state (either shared or exclusive). The entries in the private state tables of other processors on the node are *upgraded* only if the processors attempt to access the data.

Shasta implements an eager release-consistent protocol in which a processor only stalls at a release until all its previous requests have completed. In addition, the Shasta protocol allows a processor to use data returned by a read-exclusive request before all the invalidation acknowledgements have arrived. However, this optimization causes some complexity in SMP-Shasta, since other processors on the same node may also access the data while acknowledgements are still outstanding. Another processor can access the valid data without even entering the protocol via a load that uses the invalid flag optimizations. To avoid any problems with correctness, we use an epoch-based solution similar to SoftFLASH [4] whereby each release starts a new epoch on the node, and a release is not performed until all stores on the node that were issued during previous epochs have completed.

3.4.3 Downgrades

In SMP-Shasta, the protocol routines that handle incoming requests first determine if a downgrade will be necessary. If so, the routines access the private state table entries of the other processors and send downgrade messages to the appropriate processors. If any downgrade messages are sent, the handler sets the shared state of the block to a *pending-downgrade state*, saves a count of the number of downgrades sent and terminates; otherwise, it executes its normal action. As each processor receives the downgrade message, it downgrades its private state table entry and decrements the downgrade count in the miss entry. The processor that handles the last downgrade message (i.e., the downgrade count reaches zero) also executes the normal protocol action associated with the request, including updating the shared state and sending the reply.

Processors are not stalled during a downgrade and continue to access the block being downgraded. If a processor has not yet received the downgrade message, then it may still load and store to that block without entering the protocol. There is no race condition, since the downgrade cannot complete until after the processor receives and handles the downgrade message. Processors that have already handled the downgrade message will invoke a protocol miss handler if their downgraded private state is not sufficient for a particular access. However, if the block is still in pending-downgrade state, the miss handler can immediately service the load or store as long as the state prior to the downgrade was sufficient for handling the request. There is no race condition in this case either, because the miss handler services the load or store while holding a lock on the block.

Because downgrades are not instantaneous, the SMP-Shasta protocol must also handle requests that arrive for a block that is in the pending-downgrade state. This case can only happen during a downgrade from exclusive to shared state. In this case, the incoming request is queued and served when the downgrade completes.

3.4.4 Batching

Although the batch miss handler sends out requests for any missing blocks, it cannot guarantee that all the blocks required by the batched code will be in the appropriate state once all replies come back. While the handler is waiting for the replies, requests from other processes may invalidate some of the blocks in the batch. To ensure that batched loads to the block will still get the correct value, the batch miss handler delays storing the invalid flag to any invalidated blocks. To handle this rare protocol case, the batch miss handler marks the state table entry for each block that is accessed by a batch and removes the marker at the end of the batch. If a block is invalidated in the middle of a batch, the storing of the invalid flag into the block is delayed until the batch ends and the marker is removed.[3]

4 Performance Results

This section presents performance results for our SMP-Shasta implementation. We first describe our prototype SMP cluster and the applications used in our study. We then present detailed performance results that show the effectiveness of our SMP protocol in improving overall performance along with reducing the number of misses and protocol messages.

4.1 Prototype SMP Cluster

Our SMP cluster consists of four AlphaServer 4100s connected by a Memory Channel network. Each AlphaServer 4100 has four 300 MHz 21164 processors, which each have 8 Kbyte on-chip instruction and data caches, a 96 Kbyte on-chip combined second-level cache, and a 2 Mbyte board-level cache. The individual processors are rated at 8.1 SpecInt95 and 12.7 SpecFP95, and the system bus has a bandwidth of 1 Gbyte/s.

The Memory Channel is a memory-mapped network that allows a process to transmit data to a remote process without any operating system overhead via a simple store to a mapped page [6]. The one-way latency from user process to user process over Memory Channel is about 4 microseconds, and each network link can support a bandwidth of 60 MBytes/sec.

Shasta uses a message-passing layer that runs efficiently on top of the Memory Channel, and exploits shared memory segments within an SMP when the communicating processors are on the same node. By using separate message buffers between each pair of processors, the message-passing system avoid the need for any locking when adding or removing messages from the buffers. In Base-Shasta, the minimum latency to fetch a 64-byte block from a remote processor (two hops) via the Memory Channel is 20 microseconds, and the effective bandwidth for large blocks is about 35 Mbytes/s. The latency to fetch a 64-byte block from another processor on the same SMP node is 11 microseconds, and the bandwidth is about 45 Mbytes/s.

4.2 Applications

We report results for nine of the SPLASH-2 applications [17]. Table 1 shows the input sizes used in our experiments along with the sequential running times. We have increased some of the standard input sizes in order to make sure that the applications run for at least a few seconds on our cluster. Table 1 also shows the single processor execution times for each application after the Base-Shasta and SMP-Shasta miss checks are added, along with the percentage increase in the time over the original sequential time. The average overhead of the SMP-Shasta miss checks is higher than for the Base-Shasta miss checks (24.0% vs. 14.7%) because of the changes described in Section 3.4.1. The three applications that are most affected by this are Raytrace and the two versions of Water, with the overheads increasing by as much as three times for Raytrace.

[3] In order to avoid accessing stale data, a processor stalls at an acquire if any blocks on the same SMP are in this intermediate state.

	problem size	sequential time	with Base-Shasta miss checks	with SMP-Shasta miss checks
Barnes	16K particles	9.08s	9.83s (8.3%)	10.20s (12.3%)
FMM	32K particles	13.76s	15.39s (11.8%)	16.30s (18.4%)
LU	1024x1024 matrix	27.06s	32.84s (21.3%)	32.65s (20.6%)
LU-Contig	1024x1024 matrix	17.52s	21.41s (22.2%)	22.57s (28.8%)
Ocean	514x514 ocean	11.07s	13.15s (18.7%)	13.82s (24.8%)
Raytrace	balls4	71.94s	78.31s (8.8%)	90.32s (25.5%)
Volrend	head	1.63s	1.76s (7.9%)	1.77s (8.5%)
Water-Nsq	1000 molecules	7.87s	9.15s (16.2%)	10.40s (32.1%)
Water-Sp	1728 molecules	6.70s	7.86s (17.3%)	8.74s (30.4%)

Table 1: Sequential times and checking overheads for the SPLASH-2 applications.

4.3 Parallel Performance

This section presents the parallel performance of the applications for both the Base-Shasta and SMP-Shasta protocols. To ensure a fair comparison, we use the same placement of processes on each SMP for both Base-Shasta and SMP-Shasta. Two- and four-processor runs always execute entirely on a single node, and 8-processor and 16-processor runs use two nodes and four nodes, respectively. [4] Processors on the same node share the Memory Channel bandwidth when sending messages to destinations on other nodes. The network bandwidth available per processor is therefore identical for corresponding Base-Shasta and SMP-Shasta runs. Keeping the network bandwidth per processor constant for both protocols allows us to better isolate the benefits of sharing provided by SMP-Shasta. Applications using Base-Shasta could also be run on individual workstations, each with their own link to the Memory Channel, thus providing more bandwidth per processor. Since SMP nodes typically provide a larger number of I/O buses than individual workstations, the bandwidth per processor may also be increased on an SMP node by using more network links. To ensure that we are not limiting the performance of Base-Shasta in our experiments, we compared results for 8-processor runs on Base-Shasta using 2 processors per node with results using 4 processors per node. The runs with 4 processors per node had better performance for all applications (partly because Base-Shasta exploits faster messaging within an SMP), except for Ocean and Raytrace, where the difference in performance was less than 10%.

We report SMP-Shasta results for SMP clustering of 1, 2, and 4 processors. For SMP clustering of 1, each process acts as if it is on a node by itself, and therefore sends messages to all other processes. Processes that are on the same physical node communicate via message passing through a shared-memory segment. Similarly, for an SMP clustering of 2, each process shares memory with only one other process on the node and communicates only via messages with the other two processes on the node. We use a fixed Shasta line size of 64 bytes. Unless specified otherwise, the block size of objects less than 1024 bytes is automatically set to the size of the object, while larger objects use a 64 byte block size. In addition, for FMM, LU-Contiguous and Ocean, we use the standard home placement optimization, as is done in most studies of the SPLASH-2 applications.

We should note that the Base-Shasta implementation has been tuned for good performance, while the SMP-Shasta implementation has not yet been tuned. We expect that the performance of applications on SMP-Shasta will improve with changes such as reducing the locking overhead in the protocol, eliminating false sharing among protocol data structures, and improving the implementation of the lock and barrier primitives used by the application.

[4] Results for 2, 4, and 8 processors are not directly comparable with those presented in in another Shasta paper [11] because of different assignment of processes to nodes.

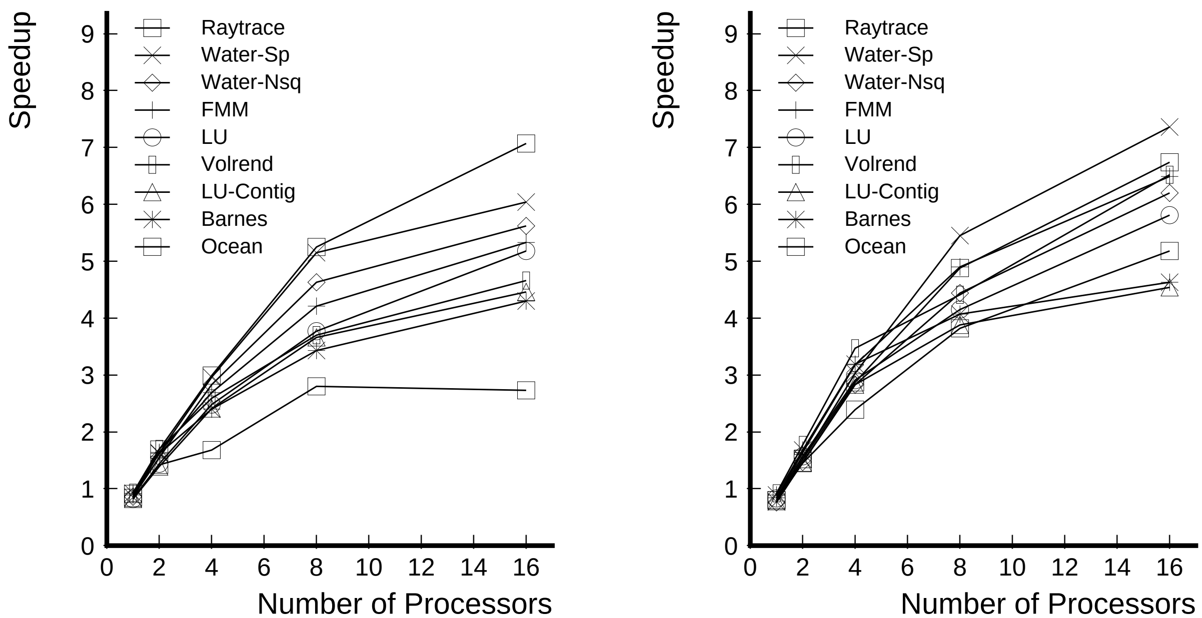

Figure 3: Speedups of SPLASH-2 applications running with Base-Shasta (left) and SMP-Shasta (right).

To gauge the efficiency of SMP-Shasta, we did a comparison with the applications running on a single 4-processor AlphaServer 4100 using an efficient implementation of ANL macros that directly uses the hardware coherence protocol. All of our applications get a speedup of 3.8 or better on 4 processors using the ANL macros, except for LU and Ocean which got speedups of 3.4 and 3.0, respectively. We ran the same applications using SMP-Shasta on 4 processors using a clustering of 4 processors, so that communication is mainly via hardware shared memory and protocol actions are only invoked for synchronization and for initial upgrades of the private state table entries. The absolute running times on 4 processors using SMP-Shasta are slower than the ANL runs by an average of 12.7%. In general, the difference in running times reflects the extra overhead due to the inline checking code, with a few applications also getting affected by the non-optimized synchronization primitives in SMP-Shasta.

Figure 3 shows the speedups for the applications running on our prototype cluster for both Base-Shasta and SMP-Shasta. For SMP-Shasta we use a clustering of 2 at 2 processors and a clustering of 4 at 4, 8, and 16 processors. The speedups shown are based on the execution time of the application running via Shasta on 1 to 16 processors relative to the execution of the original sequential application (with no miss checks).

Figure 4 presents the change in the execution time of 8- and 16-processor runs when the applications are run using SMP-Shasta. For each application, the height of the first bar (labeled “B”) represents the execution time for the run on Base-Shasta, and bars representing other times are normalized to this bar. The height of the second bar for each application shows the normalized execution time when the SMP-Shasta protocol is used with a clustering of 1 processor. This execution time is always larger with respect to Base-Shasta, because of the extra checking overhead and extra protocol overhead incurred by SMP-Shasta. The second and third bars show the normalized times for clustering of 2 and 4 processors. The execution time always goes down as the clustering increases because of a reduction in the

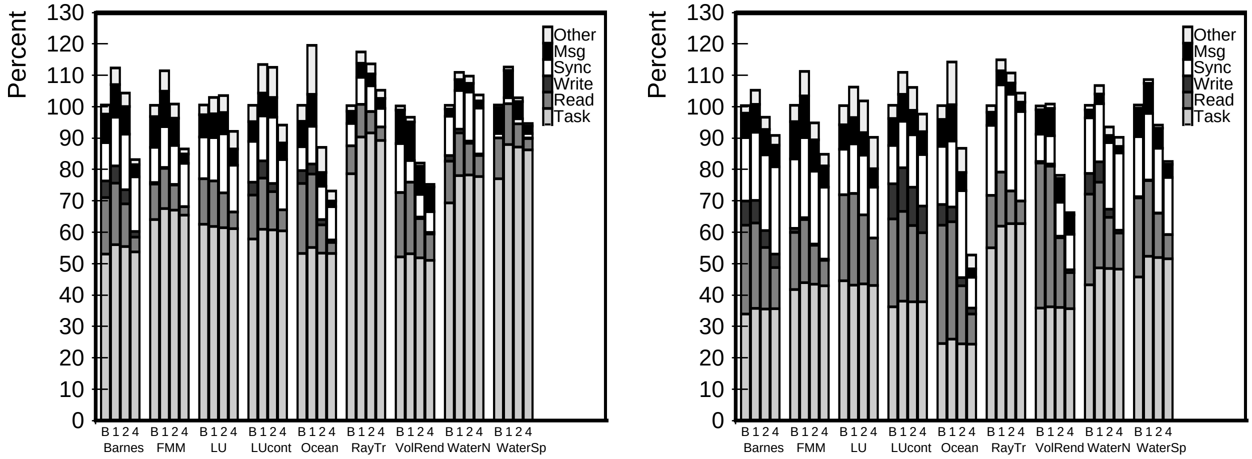

Figure 4: Execution time breakdowns for 8-processor (left) and 16-processor (right) runs on Base-Shasta and SMP-Shasta.

number of misses and messages. For most applications, there is a significant performance benefit from clustering of 4 at both 8 and 16 total processors.

Figure 4 also shows the breakdown of the execution time for each of the runs. Task time represents the time spent executing the application, including hardware cache misses. Task time also includes the time for executing the inline miss checks and the code necessary to enter the protocol (such as saving registers). Read time and write time represent the stall time for read and write misses that are satisfied by other processors through the software protocol. (Even though Shasta supports non-blocking stores, there is still some stall time for stores because of protocol limitations on the distribution and number of outstanding stores.) Synchronization time represents the stall time for application locks and barriers. Message time represents the time spent handling messages when the processor is not already stalled. Processors also handle messages while stalled on data or synchronization, but this time is hidden by the read, write, and synchronization times. The “other” category includes the overhead of dealing with non-blocking stores to pending blocks. In SMP-Shasta, this category also includes the time to upgrade a processor’s private state table and any overheads for dealing with blocks in a pending-downgrade state.

Most of the time components increase when we go from Base-Shasta to SMP-Shasta with a clustering of 1. The task time increases because of the extra checking costs in SMP-Shasta. Other times typically increase because of extra protocol overheads (mainly due to locking). As we increase the degree of clustering, however, the read and write stall times (and other miss-related times) typically decrease because of a reduction in the number of misses handled by the software protocol. The synchronization time does not change much however because it is more a function of application behavior.

Ocean shows the highest gains from clustering at both 8 and 16 total processors. These gains are somewhat expected due to the nearest-neighbor nature of the communication in Ocean. For several applications, the relative reduction in the read latency due to clustering is larger at 8 processors as compared to 16 processors. However, the overall gain in performance is larger at 16 processors, since the read time constitutes a larger fraction of the execution time. Another visible difference between the breakdowns for 8 and 16 processor runs is that the task time constitutes a larger portion of the execution time at 8 processors, since communication and synchronization overheads typically

	selected data structure(s)	specified block size (bytes)	16-proc. speedup (Base-Shasta)	
			default block size (64 bytes)	specified block size
Barnes	cell, leaf arrays	512	4.3	5.2
FMM	box array	256	5.3	5.8
LU	matrix array	128	5.2	6.8
LU-Contig	matrix block	2048	4.5	8.8
Volrend	opacity, normal maps	1024	4.7	5.3
Water-Nsq	molecule array	2048	5.6	6.1

Table 2: Effects of variable block size in Base-Shasta.

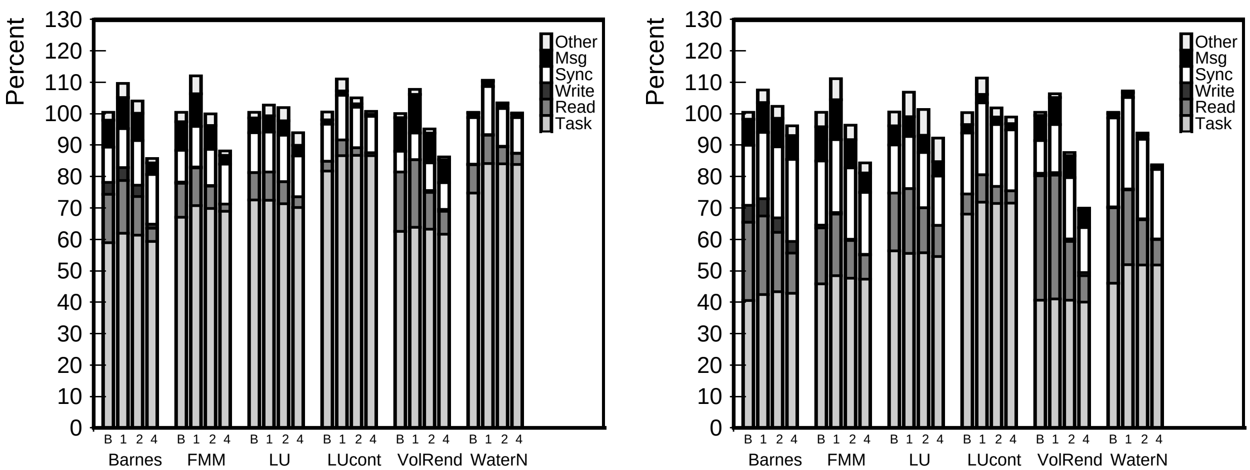

Figure 5: Time breakdowns in 8-processor (left) and 16-processor (right) runs using variable granularity on Base-Shasta and SMP-Shasta.

increase as we go to more processors. Because of the above, the increases in task time due to extra checking overheads in SMP-Shasta have a more significant effect at 8 processors. The three applications most affected by the additional checking overheads are Raytrace and the two versions of Water. In fact, Raytrace and Water-Nsquared are the only two applications that run slower under SMP-Shasta (relative to Base-Shasta) at 8 processors even for a clustering of 4. At 16 processors, SMP-Shasta with a clustering of 4 provides an improvement over Base-Shasta for all the applications except Raytrace, with Ocean showing a 1.9 times improvement in performance. Finally, a clustering of 2 does not provide sufficient benefits from data sharing (at 16 processors) to counteract the overheads introduced by the SMP protocol for several other applications (e.g., LU or LU-Contig).

To study the effects of variable coherence granularity, we made single-line changes to six of the applications to make the coherence granularity of one or a few of the main data structures greater than 64 bytes. (The coherence granularity is a hint that can be specified at allocation time as a parameter to a modified `malloc` routine.) Table 2 shows the affected data structures along with the larger block size. We also show the change in speedups for 16-processor runs under Base-Shasta when the larger granularity is used. (All speedups are with respect to the execution time of original sequential code with no miss checks.) The variable granularity improves performance by transferring data in larger units and reducing the number of misses on the main data structures. Therefore, SMP-Shasta might provide a smaller

	problem size	sequential time	checking overhead		16-proc speedup	
			Base	**SMP**	**Base**	**SMP**
Barnes	64K particles	41.25s	9.3%	15.9%	5.8	6.0
FMM	64K particles	28.28s	11.4%	20.2%	6.3	6.8
LU	2048x2048 matrix	220.34s	20.5%	19.5%	7.4	8.0
LU-Contig	2048x2048 matrix	141.05s	22.7%	29.0%	5.8	6.3
Ocean	1026x1026 ocean	44.90s	20.0%	21.9%	4.2	7.2
Water-Nsq	4096 molecules	126.08s	17.4%	33.2%	9.7	9.3
Water-Sp	4096 molecules	15.92s	17.9%	31.5%	8.6	9.7

Table 3: Execution times for larger problem sizes (64-byte line size).

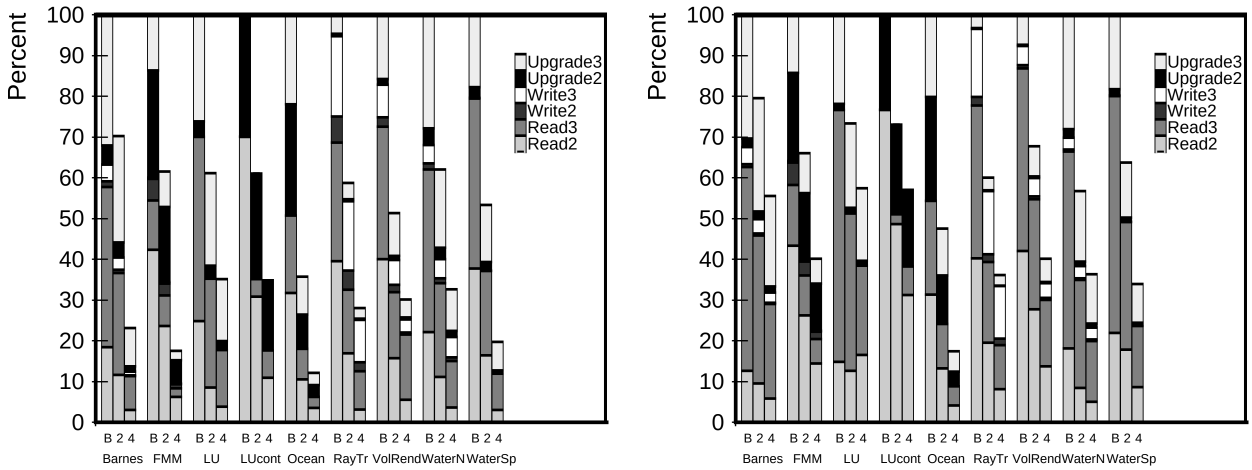

Figure 6: Misses in 8-processor (left) and 16-processor (right) runs on Base-Shasta and SMP-Shasta.

performance benefit over Base-Shasta for applications whose performance has already been improved via granularity changes. Figure 5 shows the breakdown for execution times of 8- and 16-processor runs on SMP-Shasta with the indicated granularity changes. For the 16-processor results, while Barnes, and LU-Contig do not show much gain from SMP-Shasta, FMM, LU, Volrend, and Water-Nsquared still get a large benefit at a clustering of 4. Finally, the combination of variable granularity and SMP-Shasta always provides the highest overall performance on our cluster.

To demonstrate that performance of both Base-Shasta and SMP-Shasta improve with increasing problem sizes, Table 3 gives speedups for some larger inputs sets. For each application, we give the sequential running time for the specified input size, the miss check overheads for both Base-Shasta and SMP-Shasta, and the 16-processor speedups for Base-Shasta and SMP-Shasta (for a clustering of 4). The Base-Shasta speedups are clearly improved over the default speedups in Table 2. In addition, the use of SMP-Shasta still provides significant improvements in performance over the use of Base-Shasta for the larger input sizes (except for Water-Nsquared). These results are for 64-byte lines and do not use the larger granularities described above; speedups would be larger if the granularity changes above were used.

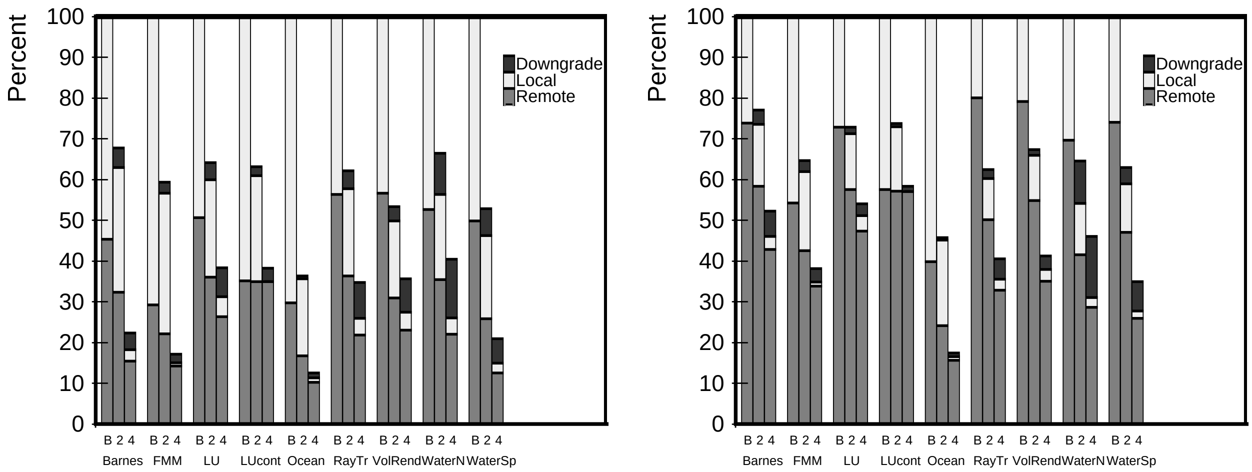

Figure 7: Messages in 8-processor (left) and 16-processor (right) runs on Base-Shasta and SMP-Shasta.

4.4 Statistics on Misses and Messages

Figure 6 shows the decrease in the number of misses as clustering increases. For each application there are three bars. The leftmost represents the total misses in the application under Base-Shasta and is normalized to 100 percent. The other two bars show the relative number of misses when SMP-Shasta is used with clustering of 2 and 4 processors. The bars are divided into six segments, which classify the misses based on the request type (read, write, and upgrade) and the number of hops needed to satisfy the request (2 or 3). In SMP-Shasta, a request is considered 3 hops if the reply is from a processor other than the home processor, even if it is from the same SMP as the home. We see that the total number of misses decreases dramatically with SMP-Shasta, especially for a clustering of 4 processors. In addition, the number of 3-hop requests always goes down with increasing clustering. The number of 2-hop requests can sometimes go up with increasing clustering, but only because many 3-hop requests are converted into 2-hop requests. Ocean shows the most dramatics decrease in total misses, which explains its large performance improvement.

We have measured the average latency for read requests in our parallel runs. For most applications, the average latency for SMP-Shasta is a few microseconds higher than for Base-Shasta, largely due to the extra locking in the protocol. However, in some applications, the average latency goes down, probably because there are many fewer protocol messages and therefore less contention in handling incoming messages.

Figure 7 shows the decrease in the number of messages as clustering increases. Again, the leftmost bar for each application represents the total messages sent under Base-Shasta and is normalized to 100 percent. The other two bars show the relative number of messages when SMP-Shasta is used with clustering of 2 and 4 processors. The first segment (“remote”) represents the number of protocol messages sent between processors on different nodes. The second segment of each bar (“local”) represents the number of protocol messages sent between processors on the same SMP, excluding downgrade messages. The third segment represents the number of downgrade messages (which only occur in SMP-Shasta) sent between processors on the same SMP. We note that around 40-60% of the messages sent in 8-processor Base-Shasta runs and 20-40% of the message sent in 16-processor runs are local, so these runs are taking advantage of the fast communication within a single node for a significant fraction of the messages. Clearly, the total number of protocol messages (including downgrade messages) goes down as the clustering increases. At a clustering of 4 processors, the number of local messages is always a small fraction of the total messages sent. Similarly, the

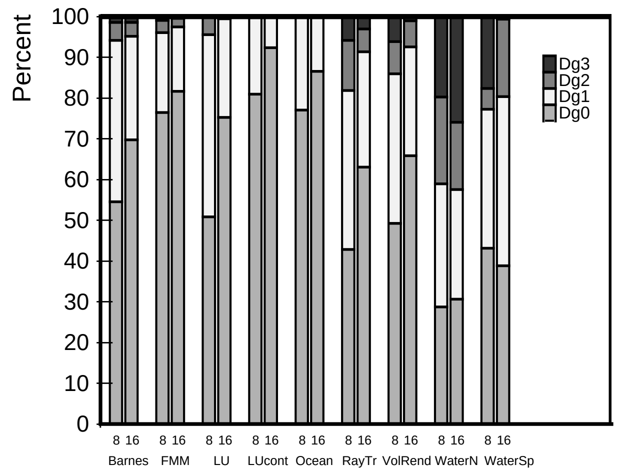

Figure 8: Distribution of number of downgrade messages sent during block downgrades in 8- and 16-processor runs on SMP-Shasta.

number of downgrade messages is typically a small fraction of the total messages sent for 16 processor runs with a clustering of 4. Water-Spatial and Water-Nsquared are notable exceptions to this observation.

In Figure 8, we show the distribution in the number of downgrade messages that must be sent each time a block is downgraded in SMP-Shasta. For each application, there are two bars, which give these results for 8-processor and 16-processor runs, each with a clustering of 4 processors. The individual bars show the percentage of time that 0, 1, 2, and 3 downgrade messages must be sent when downgrading a block. In most applications, the large majority of downgrades require zero or one downgrade messages to be sent, and only a very small fraction of the downgrades require three downgrade messages. In addition, the average number of downgrade messages that must be sent typically goes down significantly in going from 8 to 16 processor runs. Water-Nsquared (and Water-Spatial at 8 processors) are somewhat unusual in that a lot of downgrades require three downgrade messages. This effect may be caused by migratory data that tends to move among the nodes of an SMP before migrating to another node. When the data finally moves to a processor on another node, all of the local processors must be downgraded, since they have all accessed the data.

Using a small microbenchmark code, we have measured the latency for a read request when zero, one, two, and three downgrades are required. The latency increases by about ten microseconds in going from zero to one downgrades, and by five microseconds for each additional downgrade.

4.5 Summary of Results

Our results show that despite the extra checking and protocol costs, SMP-Shasta provides performance benefits at both 8 and 16 processors with a clustering of four processors. For 16-processor runs, one application improves by nearly a factor of two, six applications speed up by factors of 1.1 to 1.4, one application improves slightly, and one application gets slightly worse. The sharing of application data by processors on the same node greatly reduces the number of misses, and the use of SMP-Shasta greatly reduces the total number of messages, as compared to Base-Shasta. Although it has a smaller effect for some applications, SMP-Shasta can improve performance even for applications whose performance has been tuned by changing the coherence granularity of key data structures. Our best results are always achieved when using SMP-Shasta in conjunction with variable granularity optimizations. Finally, the use of private state tables is successful in minimizing the number of local downgrade messages that must be sent out during a downgrade.

5 Discussion and Related Work

Our work builds on previous research on Blizzard-S [14] and Shasta [12] that study software support for fine-grain distributed shared memory. We have extended the Shasta protocol to exploit data sharing and clustering benefits within SMP nodes. This protocol is fully functional and runs on our prototype SMP cluster.

Several researchers have considered using SMP nodes as building blocks for software Shared Virtual Memory (SVM) systems [1, 3, 4, 8, 18]. Among these, SoftFLASH [4] and MGS [18] are the only real implementations, with SoftFLASH being the only implementation based on commercial multiprocessor nodes. The primary difference with our study is that these systems support coherence across nodes at a fixed coarse granularity equal to the size of a virtual memory page, while we support both fine and variable coherence granularity. Furthermore, the above systems depend on the virtual memory hardware to provide atomic state lookup and data access at the granularity of a page. The following provides a more detailed discussion of these systems.

SoftFLASH [4] uses a modified version of the Stanford FLASH protocol to support coherence in software at the granularity of 16 KByte pages across a cluster of SGI Power Challenge machines (each with 16-18 90MHz MIPS R8000 processors) connected by 100 MByte/sec HIPPI links. In contrast to most recent SVM systems, SoftFLASH supports only a single-writer protocol with no "diffing". The paper presents results for four SPLASH-2 applications (Barnes, FFT, LU, Ocean) using large problem sizes. The study shows that while clustering is effective in reducing internode communication, it is often accompanied by an increase in the latency of such communication; approximately 250 μs of the best-case read latency of 1419 μs is due to TLB shootdowns that are used to reduce the privilege of a page within a node. The best performance is achieved when the number of processors per node is maximized. Ocean (2050x2050 grid) obtains the best speedups among the applications, with a speedup of around 13 with 16 processors (across 2 nodes) used for application code. However, it is important to qualify these speedups to account for the extra 5 processors per node dedicated to interrupt handling (which improve performance by 20%), the large problem sizes, and the larger number of processors on each node (which provide more potential gain from clustering).

There are some similarities between the TLB shootdown mechanism in SoftFLASH and the downgrade of private state in SMP-Shasta. The TLB entry can be thought of as a local copy of the state table entry for a page; it is upgraded through a TLB fault and downgraded through explicit TLB shootdown interrupts. However, the TLB hardware provides atomic state lookup and data access, allowing downgrades to be handled through interrupts; in contrast, our downgrade mechanism uses polling (which is likely more efficient anyway) to avoid downgrades between the inline state check and data access. In addition, the private state table in SMP-Shasta is used to selectively send downgrades to only the processors that have already accessed a line; in contrast, SoftFLASH sends TLB shootdowns to all processors on a node on every downgrade transition. Furthermore, SoftFLASH requires that the processor receiving a request wait for all shootdowns to complete before handling the request. The SMP-Shasta protocol allows all processors to continue executing during a downgrade; the incoming request is actually handled at the processor that downgrades last. The above differences result in a much lower frequency and cost for explicit downgrades in SMP-Shasta.

MGS implements a Munin-like protocol on top of the MIT Alewife machine [18]. The processors are slow (20 MHz) relative to the network. Because Alewife lacks TLB translation hardware, MGS uses inline code that has a high overhead (18-24 processor cycles per translation) to emulate the TLB for pointer and array references, effectively making the processors appear even slower. The above effects make a comparison with our results quite difficult.

Cox et al. [3] provide the earliest study (that we are aware of) of using SMP nodes in software DSM systems. They simulate the TreadMarks protocol [9] on both single processor and eight processor nodes. Their results show that clustering is beneficial for the three applications they consider. Karlsson et al. [8] provide a simulation study of the TreadMarks protocol running on an ATM cluster. They find that, given the parameters in their study (e.g., high latency of ATM interface), dedicating a processor in the SMP to protocol processing does not pay off since there is a high likelihood of finding spare cycles on the compute processors on a node. Finally, Bilas et al. [1] provide simulation studies of both a TreadMark-like protocol and the AURC protocol [7] running on an SMP cluster. Their results for the TreadMarks-like protocol are optimistic, since they do not model the cost of TLB shootdowns. The results of the AURC protocol show a slowdown from exploiting SMP nodes for 4 out of 5 applications compared to using uniprocessor nodes; the slowdown is due to the fact that AURC uses more bandwidth than a TreadMarks-like protocol and that the bandwidth per processor is decreased in their study when they go to SMP nodes.

Many software DSM systems (including Treadmarks and AURC) depend on properly-labeled programs [5] for correct execution. While restricting a system to properly-labeled programs potentially allows additional optimizations, this approach sacrifices the ability to transparently execute all legal programs for a given architecture. On the other hand, Shasta will correctly execute any Alpha program, whether or not the program exhibits races. In addition, Shasta does not require exact labeling of acquires and releases, as required by protocols that exploit lazy release consistency. Such information is not available in the executables of any commercial processors, even those that support a relaxed memory model (e.g. Alpha, PowerPC, and Sparc). Again, the need for exact labeling sacrifices transparency.

A few software or hybrid hardware/software DSM systems have explored dedicating the second processor on a dual-processor SMP node for protocol and message handling (e.g., Typhoon-0 [13], Home-Based LRC [19]). These systems do not exploit any of the intra-node data sharing and clustering benefits of SMP nodes. Furthermore, any speedup numbers reported for P processors must be qualified by the fact that the system actually uses 2P general-purpose processors to achieve that performance.

We plan to extend our work on SMP-Shasta in several areas. There are numerous protocol actions that can be further tuned. We also plan to exploit benefits that may arise from sharing more data structures among local processors (such as the directory state or incoming message queues, as discussed in Section 3.1), and to implement more efficient lock and barrier synchronization primitives by exploiting the SMP hardware.

6 Conclusion

Shasta is a software distributed shared memory system that supports fine-grain access to shared memory by inserting code before loads and stores in an application that checks the state of the shared data being accessed. We have explored opportunities for improving performance when Shasta is used to execute parallel applications on a cluster that consists of SMP nodes. We have developed modifications to the base Shasta protocol that allow application data to be shared among processors on a single SMP via the hardware cache-coherent shared memory. The protocol eliminates potential races between the inline checking code and other protocol operations without introducing any synchronization operations in the inline code. Our method is to send “downgrade” message to local processors for operations that can lead to race conditions involving inline checks and to maintain private state information to minimize the number of downgrade messages that must be sent.

We have implemented this protocol in Shasta for our cluster of four Alpha SMPs connected by the Memory Channel. Despite the extra checking and protocol costs, the performance of eight of the nine SPLASH-2 applications improves when this protocol is used for 16-processor runs. One application improves by nearly a factor of two, six applications speed up by factors of 1.1 to 1.4, and one application only slightly improves. The SMP-Shasta protocol is effective in improving performance even for applications that make use of the variable granularity mechanism in Shasta. The best performance is always achieved by using SMP-Shasta along with the variable granularity optimizations. The SMP-Shasta protocol appears to be successful in effectively exploiting the fast communication provided by SMP nodes, and we expect its performance benefit to improve as we tune the current implementation.

Acknowledgments

We would like to thank Marc Viredaz, Drew Kramer, and Luiz Barroso for their help in setting up and maintaining our cluster of AlphaServers.

References

[1] A. Bilas, L. Iftode, D. Martin, and J. P. Singh. Shared Virtual Memory Across SMP Nodes Using Automatic Update: Protocols and Performance. Technical Report TR-517-96, Department of Computer Science, Princeton University, 1996.

[2] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, Oct. 1991.

[3] A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwaenepoel. Software Versus Hardware Shared-Memory Implementation: A Case Study. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 106–117, April 1994.

[4] A. Erlichson, N. Nuckolls, G. Chesson, and J. Hennessy. SoftFLASH: Analyzing the Performance of Clustered Distributed Virtual Shared Memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 210–220, Oct. 1996.

[5] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[6] R. B. Gillett. Memory Channel Network for PCI. *IEEE Micro*, 16(1):12–18, Feb. 1996.

[7] L. Iftode, C. Dubnicki, E. Felten, and K. Li. Improving Release-Consistent Shared Virtual Memory Using Automatic Update. In *Proceedings of the 2nd Symposium on High-Performance Computer Architecture*, Feb. 1996.

[8] M. Karlsson and P. Stenstrom. Performance Evaluation of a Cluster-Based Multiprocessor Built from ATM Switches and Bus-Based Multiprocessor Servers. In *Proceedings of the 2nd International Symposium on High-Performance Computer Architecture*, pages 4–13, February 1996.

[9] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the 1994 Winter Usenix Conference*, pages 115–132, January 1994.

[10] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, Nov. 1989.

[11] D. J. Scales and K. Gharachorloo. Performance of the Shasta Distributed Shared Memory Protocol. Technical Report 97/2, Western Research Laboratory, Digital Equipment Corporation, Feb. 1997.

[12] 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.

[13] I. Schoinas, B. Falsafi, M. D. Hill, J. R. Larus, C. E. Lukas, S. S. Mukherjee, S. K. Reinhardt, E. Schnarr, and D. A. Wood. Implementing Fine-Grain Distributed Shared Memory on Commodity SMP Workstations. Technical Report 1307, University of Wisconsin Computer Sciences, Mar. 1996.

[14] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-grain Access Control for Distributed Shared Memory. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297–306, Oct. 1994.

[15] J. P. Singh, W. D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. *Computer Architecture News*, 20(1):5–44, Mar. 1992.

[16] R. L. Sites and R. T. Witek, editors. *Alpha AXP Architecture Reference Manual*. Digital Press, 1995. Second Edition.

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

[18] D. Yeung, J. Kubiatowicz, and A. Agarwal. MGS: A Multigrain Shared Memory System. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 44–56, May 1996.

[19] Y. Zhou, L. Iftode, and K. Li. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, Oct. 1996.

WRL Research Reports

‘‘Titan System Manual.’’ **Michael J. K. Nielsen.** WRL Research Report 86/1, September 1986.

‘‘Global Register Allocation at Link Time.’’ **David W. Wall.** WRL Research Report 86/3, October 1986.

‘‘Optimal Finned Heat Sinks.’’ **William R. Hamburgen.** WRL Research Report 86/4, October 1986.

‘‘The Mahler Experience: Using an Intermediate Language as the Machine Description.’’ **David W. Wall and Michael L. Powell.** WRL Research Report 87/1, August 1987.

‘‘The Packet Filter: An Efficient Mechanism for User-level Network Code.’’ **Jeffrey C. Mogul, Richard F. Rashid, Michael J. Accetta.** WRL Research Report 87/2, November 1987.

‘‘Fragmentation Considered Harmful.’’ **Christopher A. Kent, Jeffrey C. Mogul.** WRL Research Report 87/3, December 1987.

‘‘Cache Coherence in Distributed Systems.’’ **Christopher A. Kent.** WRL Research Report 87/4, December 1987.

‘‘Register Windows vs. Register Allocation.’’ **David W. Wall.** WRL Research Report 87/5, December 1987.

‘‘Editing Graphical Objects Using Procedural Representations.’’ **Paul J. Asente.** WRL Research Report 87/6, November 1987.

‘‘The USENET Cookbook: an Experiment in Electronic Publication.’’ **Brian K. Reid.** WRL Research Report 87/7, December 1987.

‘‘MultiTitan: Four Architecture Papers.’’ **Norman P. Jouppi, Jeremy Dion, David Boggs, Michael J. K. Nielsen.** WRL Research Report 87/8, April 1988.

‘‘Fast Printed Circuit Board Routing.’’ **Jeremy Dion.** WRL Research Report 88/1, March 1988.

‘‘Compacting Garbage Collection with Ambiguous Roots.’’ **Joel F. Bartlett.** WRL Research Report 88/2, February 1988.

‘‘The Experimental Literature of The Internet: An Annotated Bibliography.’’ **Jeffrey C. Mogul.** WRL Research Report 88/3, August 1988.

‘‘Measured Capacity of an Ethernet: Myths and Reality.’’ **David R. Boggs, Jeffrey C. Mogul, Christopher A. Kent.** WRL Research Report 88/4, September 1988.

‘‘Visa Protocols for Controlling Inter-Organizational Datagram Flow: Extended Description.’’ **Deborah Estrin, Jeffrey C. Mogul, Gene Tsudik, Kamaljit Anand.** WRL Research Report 88/5, December 1988.

‘‘SCHEME->C A Portable Scheme-to-C Compiler.’’ **Joel F. Bartlett.** WRL Research Report 89/1, January 1989.

‘‘Optimal Group Distribution in Carry-Skip Adders.’’ **Silvio Turrini.** WRL Research Report 89/2, February 1989.

‘‘Precise Robotic Paste Dot Dispensing.’’ **William R. Hamburgen.** WRL Research Report 89/3, February 1989.

‘‘Simple and Flexible Datagram Access Controls for Unix-based Gateways.’’ **Jeffrey C. Mogul.** WRL Research Report 89/4, March 1989.

‘‘Spritely NFS: Implementation and Performance of Cache-Consistency Protocols.’’ **V. Srinivasan and Jeffrey C. Mogul.** WRL Research Report 89/5, May 1989.

‘‘Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines.’’ **Norman P. Jouppi and David W. Wall.** WRL Research Report 89/7, July 1989.

‘‘A Unified Vector/Scalar Floating-Point Architecture.’’ **Norman P. Jouppi, Jonathan Bertoni, and David W. Wall.** WRL Research Report 89/8, July 1989.

''Architectural and Organizational Tradeoffs in the Design of the MultiTitan CPU.'' **Norman P. Jouppi.** WRL Research Report 89/9, July 1989.

''Integration and Packaging Plateaus of Processor Performance.'' **Norman P. Jouppi.** WRL Research Report 89/10, July 1989.

''A 20-MIPS Sustained 32-bit CMOS Microprocessor with High Ratio of Sustained to Peak Performance.'' **Norman P. Jouppi and Jeffrey Y. F. Tang.** WRL Research Report 89/11, July 1989.

''The Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance.'' **Norman P. Jouppi.** WRL Research Report 89/13, July 1989.

''Long Address Traces from RISC Machines: Generation and Analysis.'' **Anita Borg, R.E.Kessler, Georgia Lazana, and David W. Wall.** WRL Research Report 89/14, September 1989.

''Link-Time Code Modification.'' **David W. Wall.** WRL Research Report 89/17, September 1989.

''Noise Issues in the ECL Circuit Family.'' **Jeffrey Y.F. Tang and J. Leon Yang.** WRL Research Report 90/1, January 1990.

''Efficient Generation of Test Patterns Using Boolean Satisfiablilty.'' **Tracy Larrabee.** WRL Research Report 90/2, February 1990.

''Two Papers on Test Pattern Generation.'' **Tracy Larrabee.** WRL Research Report 90/3, March 1990.

''Virtual Memory vs. The File System.'' **Michael N. Nelson.** WRL Research Report 90/4, March 1990.

''Efficient Use of Workstations for Passive Monitoring of Local Area Networks.'' **Jeffrey C. Mogul.** WRL Research Report 90/5, July 1990.

''A One-Dimensional Thermal Model for the VAX 9000 Multi Chip Units.'' **John S. Fitch.** WRL Research Report 90/6, July 1990.

''1990 DECWRL/Livermore Magic Release.'' **Robert N. Mayo, Michael H. Arnold, Walter S. Scott, Don Stark, Gordon T. Hamachi.** WRL Research Report 90/7, September 1990.

''Pool Boiling Enhancement Techniques for Water at Low Pressure.'' **Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.** WRL Research Report 90/9, December 1990.

''Writing Fast X Servers for Dumb Color Frame Buffers.'' **Joel McCormack.** WRL Research Report 91/1, February 1991.

''A Simulation Based Study of TLB Performance.'' **J. Bradley Chen, Anita Borg, Norman P. Jouppi.** WRL Research Report 91/2, November 1991.

''Analysis of Power Supply Networks in VLSI Circuits.'' **Don Stark.** WRL Research Report 91/3, April 1991.

''TurboChannel T1 Adapter.'' **David Boggs.** WRL Research Report 91/4, April 1991.

''Procedure Merging with Instruction Caches.'' **Scott McFarling.** WRL Research Report 91/5, March 1991.

''Don't Fidget with Widgets, Draw!.'' **Joel Bartlett.** WRL Research Report 91/6, May 1991.

''Pool Boiling on Small Heat Dissipating Elements in Water at Subatmospheric Pressure.'' **Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.** WRL Research Report 91/7, June 1991.

''Incremental, Generational Mostly-Copying Garbage Collection in Uncooperative Environments.'' **G. May Yip.** WRL Research Report 91/8, June 1991.

''Interleaved Fin Thermal Connectors for Multichip Modules.'' **William R. Hamburgen.** WRL Research Report 91/9, August 1991.

''Experience with a Software-defined Machine Architecture.'' **David W. Wall.** WRL Research Report 91/10, August 1991.

‘‘Network Locality at the Scale of Processes.’’ **Jeffrey C. Mogul.** WRL Research Report 91/11, November 1991.

‘‘Cache Write Policies and Performance.’’ **Norman P. Jouppi.** WRL Research Report 91/12, December 1991.

‘‘Packaging a 150 W Bipolar ECL Microprocessor.’’ **William R. Hamburgen, John S. Fitch.** WRL Research Report 92/1, March 1992.

‘‘Observing TCP Dynamics in Real Networks.’’ **Jeffrey C. Mogul.** WRL Research Report 92/2, April 1992.

‘‘Systems for Late Code Modification.’’ **David W. Wall.** WRL Research Report 92/3, May 1992.

‘‘Piecewise Linear Models for Switch-Level Simulation.’’ **Russell Kao.** WRL Research Report 92/5, September 1992.

‘‘A Practical System for Intermodule Code Optimization at Link-Time.’’ **Amitabh Srivastava and David W. Wall.** WRL Research Report 92/6, December 1992.

‘‘A Smart Frame Buffer.’’ **Joel McCormack & Bob McNamara.** WRL Research Report 93/1, January 1993.

‘‘Recovery in Spritely NFS.’’ **Jeffrey C. Mogul.** WRL Research Report 93/2, June 1993.

‘‘Tradeoffs in Two-Level On-Chip Caching.’’ **Norman P. Jouppi & Steven J.E. Wilton.** WRL Research Report 93/3, October 1993.

‘‘Unreachable Procedures in Object-oriented Programing.’’ **Amitabh Srivastava.** WRL Research Report 93/4, August 1993.

‘‘An Enhanced Access and Cycle Time Model for On-Chip Caches.’’ **Steven J.E. Wilton and Norman P. Jouppi.** WRL Research Report 93/5, July 1994.

‘‘Limits of Instruction-Level Parallelism.’’ **David W. Wall.** WRL Research Report 93/6, November 1993.

‘‘Fluoroelastomer Pressure Pad Design for Microelectronic Applications.’’ **Alberto Makino, William R. Hamburgen, John S. Fitch.** WRL Research Report 93/7, November 1993.

‘‘A 300MHz 115W 32b Bipolar ECL Microprocessor.’’ **Norman P. Jouppi, Patrick Boyle, Jeremy Dion, Mary Jo Doherty, Alan Eustace, Ramsey Haddad, Robert Mayo, Suresh Menon, Louis Monier, Don Stark, Silvio Turrini, Leon Yang, John Fitch, William Hamburgen, Russell Kao, and Richard Swan.** WRL Research Report 93/8, December 1993.

‘‘Link-Time Optimization of Address Calculation on a 64-bit Architecture.’’ **Amitabh Srivastava, David W. Wall.** WRL Research Report 94/1, February 1994.

‘‘ATOM: A System for Building Customized Program Analysis Tools.’’ **Amitabh Srivastava, Alan Eustace.** WRL Research Report 94/2, March 1994.

‘‘Complexity/Performance Tradeoffs with Non-Blocking Loads.’’ **Keith I. Farkas, Norman P. Jouppi.** WRL Research Report 94/3, March 1994.

‘‘A Better Update Policy.’’ **Jeffrey C. Mogul.** WRL Research Report 94/4, April 1994.

‘‘Boolean Matching for Full-Custom ECL Gates.’’ **Robert N. Mayo, Herve Touati.** WRL Research Report 94/5, April 1994.

‘‘Software Methods for System Address Tracing: Implementation and Validation.’’ **J. Bradley Chen, David W. Wall, and Anita Borg.** WRL Research Report 94/6, September 1994.

‘‘Performance Implications of Multiple Pointer Sizes.’’ **Jeffrey C. Mogul, Joel F. Bartlett, Robert N. Mayo, and Amitabh Srivastava.** WRL Research Report 94/7, December 1994.

‘‘How Useful Are Non-blocking Loads, Stream Buffers, and Speculative Execution in Multiple Issue Processors?.’’ **Keith I. Farkas, Norman P. Jouppi, and Paul Chow.** WRL Research Report 94/8, December 1994.

‘‘Drip: A Schematic Drawing Interpreter.’’ **Ramsey W. Haddad.** WRL Research Report 95/1, March 1995.

‘‘Recursive Layout Generation.’’ **Louis M. Monier, Jeremy Dion.** WRL Research Report 95/2, March 1995.

‘‘Contour: A Tile-based Gridless Router.’’ **Jeremy Dion, Louis M. Monier.** WRL Research Report 95/3, March 1995.

‘‘The Case for Persistent-Connection HTTP.’’ **Jeffrey C. Mogul.** WRL Research Report 95/4, May 1995.

‘‘Network Behavior of a Busy Web Server and its Clients.’’ **Jeffrey C. Mogul.** WRL Research Report 95/5, October 1995.

‘‘The Predictability of Branches in Libraries.’’ **Brad Calder, Dirk Grunwald, and Amitabh Srivastava.** WRL Research Report 95/6, October 1995.

‘‘Shared Memory Consistency Models: A Tutorial.’’ **Sarita V. Adve, Kourosh Gharachorloo.** WRL Research Report 95/7, September 1995.

‘‘Eliminating Receive Livelock in an Interrupt-driven Kernel.’’ **Jeffrey C. Mogul and K. K. Ramakrishnan.** WRL Research Report 95/8, December 1995.

‘‘Memory Consistency Models for Shared-Memory Multiprocessors.’’ **Kourosh Gharachorloo.** WRL Research Report 95/9, December 1995.

‘‘Register File Design Considerations in Dynamically Scheduled Processors.’’ **Keith I. Farkas, Norman P. Jouppi, Paul Chow.** WRL Research Report 95/10, November 1995.

‘‘Optimization in Permutation Spaces.’’ **Silvio Turrini.** WRL Research Report 96/1, November 1996.

‘‘Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory.’’ **Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath.** WRL Research Report 96/2, November 1996.

‘‘Efficient Procedure Mapping using Cache Line Coloring.’’ **Amir H. Hashemi, David R. Kaeli, and Brad Calder.** WRL Research Report 96/3, October 1996.

‘‘Optimizations and Placement with the Genetic Workbench.’’ **Silvio Turrini.** WRL Research Report 96/4, November 1996.

‘‘Performance of the Shasta Distributed Shared Memory Protocol.’’ **Daniel J. Scales and Kourosh Gharachorloo.** WRL Research Report 97/2, February 1997.

‘‘Fine-Grain Software Distributed Shared Memory on SMP Clusters.’’ **Daniel J. Scales, Kourosh Gharachorloo, and Anshu Aggarwal.** WRL Research Report 97/3, February 1997.

WRL Technical Notes

''TCP/IP PrintServer: Print Server Protocol.'' **Brian K. Reid and Christopher A. Kent.** WRL Technical Note TN-4, September 1988.

''TCP/IP PrintServer: Server Architecture and Implementation.'' **Christopher A. Kent.** WRL Technical Note TN-7, November 1988.

''Smart Code, Stupid Memory: A Fast X Server for a Dumb Color Frame Buffer.'' **Joel McCormack.** WRL Technical Note TN-9, September 1989.

''Why Aren't Operating Systems Getting Faster As Fast As Hardware?.'' **John Ousterhout.** WRL Technical Note TN-11, October 1989.

''Mostly-Copying Garbage Collection Picks Up Generations and C++.'' **Joel F. Bartlett.** WRL Technical Note TN-12, October 1989.

''Characterization of Organic Illumination Systems.'' **Bill Hamburgen, Jeff Mogul, Brian Reid, Alan Eustace, Richard Swan, Mary Jo Doherty, and Joel Bartlett.** WRL Technical Note TN-13, April 1989.

''Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers.'' **Norman P. Jouppi.** WRL Technical Note TN-14, March 1990.

''Limits of Instruction-Level Parallelism.'' **David W. Wall.** WRL Technical Note TN-15, December 1990.

''The Effect of Context Switches on Cache Performance.'' **Jeffrey C. Mogul and Anita Borg.** WRL Technical Note TN-16, December 1990.

''MTOOL: A Method For Detecting Memory Bottlenecks.'' **Aaron Goldberg and John Hennessy.** WRL Technical Note TN-17, December 1990.

''Predicting Program Behavior Using Real or Estimated Profiles.'' **David W. Wall.** WRL Technical Note TN-18, December 1990.

''Cache Replacement with Dynamic Exclusion.'' **Scott McFarling.** WRL Technical Note TN-22, November 1991.

''Boiling Binary Mixtures at Subatmospheric Pressures.'' **Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.** WRL Technical Note TN-23, January 1992.

''A Comparison of Acoustic and Infrared Inspection Techniques for Die Attach.'' **John S. Fitch.** WRL Technical Note TN-24, January 1992.

''TurboChannel Versatec Adapter.'' **David Boggs.** WRL Technical Note TN-26, January 1992.

''A Recovery Protocol For Spritely NFS.'' **Jeffrey C. Mogul.** WRL Technical Note TN-27, April 1992.

''Electrical Evaluation Of The BIPS-0 Package.'' **Patrick D. Boyle.** WRL Technical Note TN-29, July 1992.

''Transparent Controls for Interactive Graphics.'' **Joel F. Bartlett.** WRL Technical Note TN-30, July 1992.

''Design Tools for BIPS-0.'' **Jeremy Dion & Louis Monier.** WRL Technical Note TN-32, December 1992.

''Link-Time Optimization of Address Calculation on a 64-Bit Architecture.'' **Amitabh Srivastava and David W. Wall.** WRL Technical Note TN-35, June 1993.

''Combining Branch Predictors.'' **Scott McFarling.** WRL Technical Note TN-36, June 1993.

''Boolean Matching for Full-Custom ECL Gates.'' **Robert N. Mayo and Herve Touati.** WRL Technical Note TN-37, June 1993.

''Piecewise Linear Models for Rsim.'' **Russell Kao, Mark Horowitz.** WRL Technical Note TN-40, December 1993.

''Speculative Execution and Instruction-Level Parallelism.'' **David W. Wall.** WRL Technical Note TN-42, March 1994.

''Ramonamap - An Example of Graphical Groupware.'' **Joel F. Bartlett.** WRL Technical Note TN-43, December 1994.

''ATOM: A Flexible Interface for Building High Performance Program Analysis Tools.'' **Alan Eustace and Amitabh Srivastava.** WRL Technical Note TN-44, July 1994.

''Circuit and Process Directions for Low-Voltage Swing Submicron BiCMOS.'' **Norman P. Jouppi, Suresh Menon, and Stefanos Sidiropoulos.** WRL Technical Note TN-45, March 1994.

''Experience with a Wireless World Wide Web Client.'' **Joel F. Bartlett.** WRL Technical Note TN-46, March 1995.

''I/O Component Characterization for I/O Cache Designs.'' **Kathy J. Richardson.** WRL Technical Note TN-47, April 1995.

''Attribute caches.'' **Kathy J. Richardson, Michael J. Flynn.** WRL Technical Note TN-48, April 1995.

''Operating Systems Support for Busy Internet Servers.'' **Jeffrey C. Mogul.** WRL Technical Note TN-49, May 1995.

''The Predictability of Libraries.'' **Brad Calder, Dirk Grunwald, Amitabh Srivastava.** WRL Technical Note TN-50, July 1995.

WRL Research Reports and Technical Notes are available on the World Wide Web, from `http://www.research.digital.com/wrl/techreports/index.html`.