To Appear in the *16th ACM Symposium on Operating System Principles*, October, 1997

Towards Transparent and Efficient Software Distributed Shared Memory

Daniel J. Scales and **Kourosh Gharachorloo**
Western Research Laboratory
Digital Equipment Corporation
{scales,kourosh}@pa.dec.com

Abstract

Despite a large research effort, software distributed shared memory systems have not been widely used to run parallel applications across clusters of computers. The higher performance of hardware multiprocessors makes them the preferred platform for developing and executing applications. In addition, most applications are distributed only in binary format for a handful of popular hardware systems. Due to their limited functionality, software systems cannot directly execute the applications developed for hardware platforms. We have developed a system called Shasta that attempts to address the issues of efficiency and transparency that have hindered wider acceptance of software systems. Shasta is a distributed shared memory system that supports coherence at a fine granularity in software and can efficiently exploit small-scale SMP nodes by allowing processes on the same node to share data at hardware speeds.

This paper focuses on our goal of tapping into large classes of commercially available applications by transparently executing the same binaries that run on hardware platforms. We discuss the issues involved in achieving transparent execution of binaries, which include supporting the full instruction set architecture, implementing an appropriate memory consistency model, and extending OS services across separate nodes. We also describe the techniques used in Shasta to solve the above problems. The Shasta system is fully functional on a prototype cluster of Alpha multiprocessors connected through Digital's Memory Channel network and can transparently run parallel applications on the cluster that were compiled to run on a single shared-memory multiprocessor. As an example of Shasta's flexibility, it can execute Oracle 7.3, a commercial database engine, across the cluster, including workloads modeled after the TPC-B and TPC-D database benchmarks. To characterize the performance of the system and the cost of providing complete transparency, we present performance results for microbenchmarks and applications running on the cluster, include preliminary results for Oracle runs.

1 Introduction

There has been much research on supporting a shared address space in software across a cluster of workstations or servers. A variety of such distributed shared memory (DSM) systems have been developed, using various techniques to minimize the software overhead for supporting the shared address space. The most common approach uses the virtual memory hardware to detect access to data that is not available locally [2, 7, 9, 19]. These systems communicate data and maintain coherence at a fixed granularity equal to the size of a virtual page.

Despite all the research on software DSM systems, software platforms have yet to make an impact on mainstream computing. At the same time, hardware shared memory systems have gained wide acceptance. The higher performance of hardware systems makes them the preferred platform for application development. Software vendors typically distribute applications in binary format only for a few popular hardware platforms. Due to their limited functionality, software systems cannot directly execute these applications, and thus fail to capitalize on the increasing number of applications available for hardware systems. For example, software systems typically require the use of special constructs for synchronization and task creation and severely limit the use of system calls across the cluster. We have attempted to address some of the above issues of efficiency and transparency in the Shasta system [14]. Shasta is a software DSM system that supports sharing of data at a fine granularity by inserting code in an application executable that checks if data being accessed by a load or store is available locally in the appropriate state. This paper focuses on the issues in transparently executing hardware binaries in the context of the Shasta system.

Transparent execution of binaries encompasses several challenging problems which fall into two broad categories, correctly supporting the complete instruction set architecture and extending OS services across separate nodes. As an example in the instruction set category, software systems have to directly support atomic read-modify-write instructions as opposed to depending on special high-level synchronization constructs (as is done in virtually all current software DSM systems). Software systems must also correctly support the memory consistency model specified by a given instruction set architecture. Much of the recent research on software DSM systems involves protocol innovations related to exploiting or further relaxing the memory consistency model to solve false sharing problems that arise from page-level coherence. However, many important commercial architectures, including the Intel x86 architecture, support rather strict memory consistency models that disallow virtually all the critical performance optimizations that are used in such page-based DSM systems. Even architectures that support aggressive relaxed models (i.e., Alpha, PowerPC, and Sparc) fail to provide sufficient information (in the executable) to allow many of the optimizations based on release consistency that are exploited by several of the software systems [1, 7]. Furthermore, the above issues related to hardware memory consistency models are unlikely to change in the foreseeable future.

Transparently executing applications that use OS services leads to another set of challenging problems. Some of the issues are similar to those faced by cluster operating systems such as Locus [11], Sprite [10], and Solaris-MC [8]. The functionality of the OS must be extended so that system calls work transparently across the cluster as if all processes were on the same machine. However, there are a number of additional issues when software is also used to support shared memory across the cluster. One problem is that the OS is typically unaware of the software DSM layer that is providing shared memory across the nodes. Therefore, all parameters to system calls that are located in shared memory (and therefore may not currently be local) must be validated before the system call is made. Other problems can occur in complex applications because individual application processes maintain protocol state information about the application data. This arrangement works well for typical scientific applications in which a fixed number of long-lived processes are created at startup that are all active until the computation has been finished. However, many difficult issues arise if the application dynamically creates and destroys processes or if the application executes with more processes than available processors (to help hide I/O latency, for example). Our general solution to these problems is to ensure that one process per processor remains running regardless of how many application processes are created and destroyed, and to allow all processes running on the same processor (or node) to handle each other's incoming messages.

In this paper we discuss the above issues in detail and describe the corresponding solutions that we have adopted in Shasta. We have implemented a complete solution for supporting the full Alpha instruction set architecture. Due to the large amount of effort necessary to extend all OS services, we chose a short-term goal of supporting sufficient functionality to execute a commercial database such as Oracle, which still uses a relatively rich set of OS services. The Shasta system is fully functional on our prototype cluster which consists of a total of sixteen 300 MHz Alpha processors connected through the Memory Channel [6]. We can currently run the Oracle 7.3 executable across the cluster using Shasta and execute applications that are similar to the TPC-B and TPC-D benchmarks. We present performance results for various microbenchmarks and applications, including Oracle, running on the above cluster. Overall, our results show that transparent execution of binaries can be achieved in most cases with only a small reduction cost in performance.

The following section describes the basic design of Shasta. Sections 3 and 4 discuss issues related to supporting a hardware instruction set architecture and extending OS services across nodes, respectively, and describe the solutions that we have adopted in Shasta. Section 5 discusses some issues related to the use of code modification in Shasta. We present detailed performance results in Section 6. Finally, we describe related work and conclude.

2 Basic Design of Shasta

This section presents an overview of the Shasta system, which is described more fully in previous papers on support for fine-grain coherence [14], the base cache coherence protocol [12], and protocol extensions to exploit SMP nodes [13].

2.1 Cache Coherence Protocol

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. Shared data in the Shasta system has three basic states at each processor: *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; and *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 in the application executable at each load and store to determine if there is a shared miss and, if necessary, invoke protocol code.

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 program data. To simplify the inline code, Shasta divides up the blocks into fixed-size ranges called *lines* (typically 64 or 128 bytes) and maintains state information for each line in a *state table*.

Coherence is maintained using a directory-based invalidation protocol. The protocol supports three types of requests: *read*, *read-exclusive*, and *exclusive* (or *upgrade*, if the requesting processor already has the line in shared state). A *home* processor is associated with each block and maintains a *directory* for that block, which contains a list of the processors caching a copy of the block. Because of the high cost of handling messages via interrupts, messages from other processors are serviced through a polling mechanism. The Shasta implementation polls for incoming messages whenever the protocol waits for a reply. To ensure reasonable response times, Shasta also insert polls at every loop backedge. Polling is inexpensive (three instructions) in our Memory Channel cluster because the implementation arranges for a single cachable location that can be tested to determine if a message has arrived.

2.2 Shared Miss Checks

Shasta inserts the shared miss checks in the application executable using a modified version of ATOM [18]. An efficient shared miss check requires about seven instructions. Since the static and stack data areas are not shared, Shasta does not insert checks for any loads or stores that are clearly to these areas. Shasta also uses a number of other optimizations that reduce the checking overhead to an average of about 20% (including polling overhead) across the SPLASH-2 applications [14]. The two most important optimizations are described below.

Whenever a block on a processor becomes invalid, the Shasta protocol stores a particular "flag" value in each 4-byte word of the block. 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 the protocol code also checks the state of the block to distinguish an actual miss from a "false miss" (i.e., when the application data actually contains the flag value). Since false misses almost never occur in practice, the above technique can greatly reduce the load miss check overhead.

Shasta also attempts to batch together checks for multiple loads and stores. There are often sequences of loads and stores to addresses which are a small offset from the contents of a base register. These loads and stores therefore access nearby data within consecutive lines. If all the lines are in the correct state, then the batch of loads and stores can proceed without further checking. The batching technique also applies to loads and stores via multiple base registers.

2.3 Exploiting SMP Nodes

Commercial small-scale SMPs (symmetric multiprocessors) are an attractive building block for software distributed shared memory systems. The primary advantage of SMP nodes is that processors within an SMP can share memory via the hardware, thus eliminating software intervention for intra-node data sharing. The widespread availability of SMP servers has led researchers to consider their use in page-based systems [1, 3, 4, 19, 22], with SoftFLASH [4] and Cashmere [19] being the only actual implementations based on commercial multiprocessor nodes.

Exploiting SMP nodes efficiently in the context of a fine-grain system like Shasta is a bit more complex. The primary difficulty arises from race conditions caused by the fact that the state check inserted at a load or store does not execute atomically with the actual load or store of shared data, since the two actions consist of multiple instructions. In contrast, the virtual memory hardware provides atomic state check and data access in page-based 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 read request from another node. The undesirable race arises if P2 downgrades the data to a shared 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.

We have implemented a 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 [13]. The overall solution depends on the selective use of explicit messages between processors on the same node for protocol operations that can lead to the race conditions involving the inline checks. In addition to a shared state table for all processes on the same node, each process maintains a private state table that is only modified by the owner. Explicit messages are sent to another process if servicing an incoming request requires downgrading an entry in the private state table of that process (e.g., an incoming read that changes the state of a line from exclusive to shared). Because processes are allowed to read each other's private state tables, explicit *downgrade messages* are only sent to the processes that are actively sharing that line. We refer to the system with the above modifications as SMP-Shasta and the original system as Base-Shasta. When using SMP-Shasta on a cluster of SMP nodes, we have observed significant performance improvements (of as high as two times) in applications over Base-Shasta because of the reduced number of remote misses and software protocol messages [13]. As we will see, the SMP-Shasta implementation is also important for running efficiently when an application executes with more than one process per processor.

2.4 Avoiding OS Interactions for Frequent Protocol Actions

One of the key philosophies in the design of Shasta is to avoid expensive OS interactions for frequent protocol actions. By far the most important design decision in Shasta is to use inline state checks instead of depending on virtual memory hardware to detect sharing misses. This approach frees Shasta from supporting coherence at the fixed large granularity dictated by a system's page size. Physical page sizes have remained large (e.g., minimum 8KB pages under Digital Unix, and 16KB pages under SGI Irix) primarily as a way to reduce the number of TLB misses. These large page sizes exacerbate problems such as false sharing that arise from maintaining coherence at a page-level. Aside from allowing Shasta to support coherence at a fine granularity, inline state checks eliminate the need for expensive OS interactions for manipulating page protection and detecting protection faults. Shasta also avoids OS interactions for sending and receiving messages. Instead of depending on expensive interrupts, Shasta uses an efficient polling mechanism (described in Section 2.1) to detect incoming messages. Finally, OS interaction is avoided for sending messages by exploiting networks, such as Digital's Memory Channel, that provide protected user-level access.

3 Fully Supporting an Instruction Set Architecture

This section discusses some of the challenging issues that arise in fully supporting an instruction set architecture, and describes how Shasta solves them in the context of the Alpha architecture. We focus on issues related to supporting atomic read-modify-write instructions and correctly implementing the required memory consistency model.

3.1 Atomic Read-Modify-Write Instructions

All software DSM systems that we are aware of support only a limited number of high-level synchronization constructs, such as locks and barriers. Furthermore, such synchronization is typically not implemented on top of the shared memory abstraction, but is supported through specialized messages and handlers. While this approach leads to efficient support for synchronization, it is not general and fails to support transparent execution of binaries which achieve synchronization through either normal loads and stores or through specialized atomic read-modify-write instructions.

3.1.1 Semantics of Load-Locked and Store-Conditional

The Alpha architecture provides a pair of instructions, load-locked and store-conditional, that can be used together to support atomic read-modify-write functionality [16]. Similar instruction pairs are provided by the MIPS and IBM PowerPC architectures. Figure 1 shows an example of how acquisition of a lock can be implemented using this pair of instructions. The current value for the lock location is read by the load-locked (LL). This value is first checked to see whether the lock is free. In case the lock is free, the store-conditional (SC) is used to store the modified "lock-taken" value. The SC succeeds if no other processor has done a successful store to the same cache line since this processor did the LL; otherwise, the SC fails, no store is done, and failure is signalled through a zero return value. LL and SC instructions are quite general and can be used to implement numerous other atomic operations such as compare-and-swap, fetch-and-add, etc..

```
try_again:
        ld_l    t0, 0(a0)
        bgt     t0, try_again
        or      t0, 1, t0
        stl_c   t0, 0(a0)
        beq     t0, try_again
```

Figure 1: Example use of load-locked and store-conditional to implement a binary lock.

The exact semantics of the Alpha LL and SC instructions are as follows. The Alpha architecture includes a logical notion of a *lock-address* and a *lock-flag* per physical processor. The LL instruction sets the lock-flag and the lock-address. The lock-flag is reset if another processor successfully writes to the cache line specified by lock-address. An SC instruction succeeds if the lock flag is set, and fails otherwise. To ensure that an SC does not succeed incorrectly, the lock flag is also reset when there is a process switch and when an SC is executed, regardless of its success or failure. There are also situations in which an implementation *may* reset the lock flag, e.g., if there are any loads, stores, or taken branches in the execution path between the LL and SC, or if the LL and SC are not to the same 16-byte aligned address. Finally, to avoid livelock, an SC that fails should not cause failure of other LL/SC sequences; for example, livelock may occur if a failed SC still sends out invalidations to other copies of the line.

3.1.2 Solution in Shasta

A straightforward way of implementing LL and SC instructions in software is to directly emulate the lock-flag and lock-address functionality as described above. However, this solution requires saving the load-locked address and setting the lock-flag on every LL instruction and checking and resetting the lock-flag at every SC, even if the sequence does not involve any remote misses. Instead, we use a more efficient implementation that applies to LL/SC sequences that satisfy the following conditions: (a) for every SC, there is a unique LL that dominates the SC, (b) there are no load, store, LL, or SC instructions between the unique LL and SC, and (c) the LL and SC are to the same cache line. Since the Alpha architecture deprecates use of sequences that fail the above conditions, virtually all sequences in real applications fall in the simple category. [1]

Our solution uses the state of the line prior to the LL as an indication of whether the SC can run directly in hardware or requires software intervention. In the case when the line is in the exclusive state, the LL can proceed without entering the protocol. Since there are no other loads or stores between the LL and SC, the state will still be exclusive at the SC and the SC can be directly executed without entering the protocol. The hardware then ensures that the SC succeeds if and only if no other process on the same node has written to the line. The Shasta protocol is called at the SC for all other cases. The protocol returns failure if the state of the line before the SC is either invalid or pending. In the case of a shared state, the success or failure of the SC must be determined by the directory at the home processor. We have extended the protocol with a new exclusive (or upgrade) request for store-conditionals. The exclusive request is serviced (i.e., invalidations are sent out) if the directory still indicates the node is a sharer; otherwise, a failure response is sent back. The store (corresponding to the store-conditional) is completed within the protocol in the successful case. In all cases, the return from the protocol code jumps around the actual SC instruction in the application.

The following describes the inline code used to implement the LL and SC instructions. Code inserted immediately before the LL instruction loads the state of the line into a register, and calls protocol code to get the latest copy of the line if the state is invalid or pending. No explicit polls are inserted in the simple path between the LL and the SC, so as to make sure that incoming requests (or downgrade messages in SMP-Shasta) can't change the state of the line. Code inserted before the SC also checks the state in the register and proceeds to execute the SC instruction if the line is in the exclusive state. Otherwise, the code calls the protocol to handle an SC miss as described above. Note that the above scheme actually depends on the functionality of the LL and SC instructions on the underlying hardware only if (a) we are exploiting the SMP extension of Shasta, and (b) the line is locally in exclusive state. To achieve correct behavior in this case, we ensure that we do not add any taken branches, loads, or stores in the success path from the LL to the SC.

The above solution satisfies our goal of executing atomic sequences virtually at hardware speeds (and with no protocol invocation) when the line is already held in exclusive state. We have also implemented another optimization to speed up cases where the local node does not have a copy of the line. The atomic sequence typically leads to two remote misses: one to fetch a shared copy of the line for the LL, and another to fetch ownership for the SC. By adding a single prefetch-exclusive (as part of the binary rewrite phase) that fetches the line in exclusive state before the start of the atomic sequence, we can achieve a successful sequence with only a single remote miss. We insert the prefetch-exclusive prior to the loop containing the LL and SC so it is executed only once, in order to avoid the possibility of livelock among multiple sequences.

[1] For correctness, the Shasta implementation reverts to the less efficient way of implementing atomic sequences (i.e., by emulating the lock-flag and lock-address) in the unlikely case of an application that exhibits deprecated sequences.

3.1.3 Implementation in Other Software DSM Systems

It would be very difficult to implement LL and SC instructions (or other atomic memory operations) correctly on most existing software DSM systems, because (as we discuss in the next section) most assume that there are no races on memory locations. For example, multi-writer protocols assume that there are no unsynchronized writes to the same location and delay propagating writes. A page-based DSM system using a single-writer protocol (e.g. SoftFLASH) could potentially implement LL and SC in a manner similar to Shasta, but these operations would probably be very expensive. It would be crucial to send only "diffs" upon a read request; otherwise, a single atomic operation could involve transferring an entire virtual memory page.

3.2 Memory Consistency Model Issues

The memory consistency model is a fundamental issue to consider for software DSM systems, especially when the goal is to transparently support binaries for commercial architectures.

3.2.1 Current Software DSM Systems

Much of the recent research in software DSM systems has been dedicated to relaxing memory consistency models further and developing protocols that aggressively exploit such models [1, 2, 7, 19]. In general, the use of a relaxed memory model allows a system to delay protocol actions, since the ordering requirements on memory operations are relaxed. Page-based systems typically use protocols that delay the effects of writes in order to alleviate false sharing issues that arise because of the large coherence granularity. The use of a relaxed model also allows a system to reduce communication overhead by coalescing outgoing requests.

In order to exploit especially aggressive optimizations based on relaxed memory models, a large number of page-based DSM systems heavily depend on programs being properly labeled [5]. They also typically require that applications synchronize via a few high-level constructs, such as locks and barriers, that are directly implemented through message passing as opposed to on top of the shared-memory abstraction. Because of these two assumptions, all memory operations that are supported by the software shared-memory layer are guaranteed to be race-free. This property often simplifies the underlying software cache coherence protocol and allows for a large number of further optimizations. For example, the software protocols typically do not enforce serialization of writes to the same location. Similarly, writes are not guaranteed to eventually be visible to other nodes in the absence of explicit release and acquire operations (as a result of aggressively delaying protocol actions). Finally, explicit knowledge of acquire and release synchronization allows for further optimizations such as lazily following synchronization chains across different processors to avoid communicating unnecessary data (as in implementations of lazy release consistency [1, 7]).

3.2.2 Commercial Architectures

Many of the above optimizations and simplifications lead to incorrect behavior if applied to binaries from commercial architectures, either because the corresponding memory model is more strict or because the binary does not provide sufficient information about memory operations.

Several important commercial architectures support relatively strict memory consistency models. The MIPS/SGI architecture requires the system to support sequential consistency, while the popular Intel x86 architecture supports processor consistency [5] which is a little less strict. The requirement to support either model would virtually disallow all the key performance optimizations exploited in page-based systems.

A number of commercial architectures, including Alpha, PowerPC, and Sparc, support more relaxed models, which allow aggressive reordering of read and write operations. The following discussion focuses on the Alpha memory model, but the issues raised also apply to the PowerPC and Sparc models due to the similarity among these models. The Alpha memory model [16] provides special fence instructions, called *memory-barrier* (MB) instructions, for enforcing program order among memory operations where necessary. A memory barrier instruction ensures that all read and write operations preceding the MB are performed prior to any reads and writes following the MB. The Alpha model allows aggressive reordering of memory operations between memory-barrier instructions; hence, the optimizations allowed by the model are similar to weak ordering or release consistency. Nevertheless, even a commercial architecture such as Alpha that aggressively exploits relaxed models disallows a number of important optimizations that are typically exploited in page-based systems.

The fundamental issue with respect to commercial memory models such as the Alpha is that ordering information is only conveyed through special fence instructions, such as the MB, as opposed to through special flavors of loads and stores as in the release consistency model.[2] Therefore, the coherence protocol must conservatively assume that any read(s) preceding a memory barrier may potentially behave as an acquire synchronization for operations following the memory barrier, and any write(s) following a memory barrier may potentially behave as a release synchronization for operations preceding

```
P1          P2          P3                       P4

A = 1;      A = 2;      while (Flag1 != 1) ;     while (Flag2 != 1) ;
MB;         MB;         while (Flag3 != 1) ;     while (Flag4 != 1) ;
Flag1 = 1;  Flag3 = 1;  MB;                      MB;
Flag2 = 1;  Flag4 = 1;  r1 = A;                  r2 = A;
```

Figure 2: Example to illustrate issues with commercial memory models.

the memory barrier. This property fundamentally disallows aggressive optimizations, such as implementations of lazy release consistency [1, 7], that require exact knowledge about the synchronization chain across multiple processors.

Because any operation may be involved in a race, the Alpha memory model also requires that (i) all writes must eventually be propagated and made visible to other processors and (ii) writes to the same location be serialized (i.e., appear in the same order to all processors). These requirements disallow many of the optimizations even in systems that do not exploit lazy release consistency (e.g., Cashmere [19]). Figure 2 presents a contrived example to illustrate why a number of these optimizations are not correct under the Alpha memory model. The example shows P1 and P2 both writing to location A and setting some flags, while P3 and P4 wait for the flags to be set and then read location A. Assume all locations are initialized to 0. Under the Alpha memory model, the only allowable outcomes are (r1,r2)=(1,1) or (r1,r2)=(2,2). The first thing to note is that the both flag writes following the MB on P1 (or P2) behave as release synchronization, and both reads of the flags on P3 (or P4) behave as acquire synchronization. Therefore, as mentioned above, it would be incorrect to assume that only the operations that immediately precede or follow an MB behave as synchronization. In addition, an approach that lazily delays propagation of writes and servicing of invalidates until the next MB will not behave correctly. Therefore, the system has to periodically propagate writes and service invalidates even in the absence of MB instructions. Finally, the system must enforce serialization of writes to the same location to disallow outcomes such as (r1,r2)=(1,2) (which occurs if P3 and P4 observe the writes to A in different orders).

Overall, supporting commercial memory models, even those that are quite aggressive, may lead to drastic performance losses in the case of page-based systems, because the protocol optimizations that are required to achieve good performance cannot be used.

3.2.3 Approach in Shasta

The Shasta coherence protocol closely resembles that of a hardware DSM system, especially since coherence is maintained at a fine granularity. Therefore, many of the correctness and performance issues are similar to those in hardware systems. We describe a few of the issues that arise because we are supporting shared memory in software.

To support the Alpha memory model, we need to correctly implement the functionality associated with memory barri-

[2] This will not change in the foreseeable future since instructions are still 32 bits in length (even though architectures have moved to 64-bit data and addresses), and adding flavors such as acquire and release for every type of load and store is not a viable option due to opcode space limitations.

ers. This requires invoking protocol code at every MB to make sure operations before the memory barrier are completed and any incoming invalidations that are received are serviced. The binary rewrite capability in Shasta allows us to easily insert an appropriate call to the protocol after each MB instruction; we still execute the hardware MB instruction as well to ensure that proper ordering is maintained within an SMP node. With respect to eventual propagation of writes, Shasta supports a relatively eager protocol that leads to timely propagation. In addition, by inserting polls at every loop backedge, Shasta ensures that invalidations will also be serviced in a timely fashion (e.g., consider a loop waiting for a flag to be set). Finally, serialization of writes to the same location is achieved in a similar way to hardware DSM systems.

The Shasta protocol can also support more strict memory models. For sequential consistency, the protocol simply stalls on every store miss until all invalidation acknowledgments have been received. The handling of batch misses remains the same, except that some more complicated processing must be done in the rare case that the miss handler cannot fetch all lines in the correct state. However, in contrast to page-based systems, the performance of Shasta is quite good even when we support a strict model [12]. This effect is primarily because Shasta supports coherence at a fine granularity, and therefore does not depend heavily on relaxed memory models for alleviating problems associated with larger coherence granularities. Section 6.4 presents results that illustrate this point.

4 Providing OS Functionality

This section describes the techniques we have used to allow transparent execution of applications that use a rich set of operating system services. Compared to the problems solved by cluster operating systems, a number of new issues arise because we also support shared memory in software. There are three main areas that must be addressed. First, all arguments to system calls must be validated, since they may reference shared data that may not be available locally. Second, all system calls and OS services (or some specified subset) must be implemented to work correctly across a cluster of machines running independent operating systems. Third, we must address issues that arise when applications create or destroy processes dynamically or create more processes than processors. Our Shasta implementation runs under Digital Unix 3.2 and 4.0, but all of these issues apply generally to most operating systems.

4.1 Validating System Call Arguments

In most software DSM systems, the operating system as a whole is unaware of the shared memory layer that is supported through software. Therefore, a system call may not operate correctly if one of its arguments references data that is located in the global shared memory (i.e., one of its argument is a pointer to the shared memory area). For example, in a page-based software DSM, the operating system may encounter a page protection error when it references the system call arguments. Similarly, in Shasta, a read by the system call may return invalid data if the referenced line is not cached locally, and a write by the system call may be lost if the line is not already held in exclusive state (since exclusive ownership for the line will never be requested).

A simple method for validating system call arguments is to copy shared data referenced by the system call into local memory, using a copy routine that fetches the latest version of the shared data. The system call can then be invoked with arguments that point to the local copy of the data. When the system call returns, any data that has been written by the system call must be copied back to the shared region in a coherent fashion. The obvious disadvantage of this approach is the extra copying overhead, especially for system calls that may read or write a large amount of data (e.g., the *read* and *write* system calls).

A better approach is to ensure that the shared data referenced by a system call is in the correct state before invoking the system call, so the system call can operate on the original arguments. The protocol may need to request exclusive copies of data that is written to, and shared or exclusive copies of data that is read by the system call. Interestingly, the exact same functionality is required to implement batches of loads and stores, as described in Section 2.2. A system call can be logically treated as a batch of loads and stores to several ranges of lines, with validation done in the same way as for a batch. We do this validation by replacing system call routines with "wrapper" routines that validate any regions in shared memory that are referenced by the arguments (according to the semantics of the system call).

The Shasta routine for handling batches goes through each range of data and makes the appropriate requests when lines are not in the correct state. However, it cannot guarantee that the lines will all be in the appropriate state once all the replies have come back. For example, the batch miss handler may request a line for reading and receive the contents of the line, but may subsequently receive an invalidate request for the line while it is still waiting for some of its other requests to complete. However, even though a line may not be in the right state, loads to the line will still get the correct value (assuming a relaxed model such as the Alpha memory model) as long as the original contents of the line remain in memory. We therefore delay storing the invalid flag value (see Section 2.2) for invalidated lines until after the end of the batch. Similarly, the batch miss handler may request a line in exclusive state, but lose exclusive access while it is still waiting for other requests. We may therefore also have to reissue stores to lines that were not in the exclusive (or pending-shared) state when the batch miss handler returned and the batched code was executed. The above invalidations and reissues are done at the time of the next entry into the protocol code (due to explicit polls or misses) after the batch code is complete.

4.2 Extending System Calls across the Cluster

The issues of extending OS functionality across a cluster have been addressed extensively by a number of systems [8, 10, 11]. These systems typically attempt to provide almost complete transparency by reimplementing all system calls so that they work across the cluster. However, these systems are usually not concerned with efficient software support for shared memory across the cluster. Conversely, software DSM systems support shared memory across the cluster, but do not typically support system calls across the cluster. In fact, applications are typically required to limit system calls to the master process that spawns the other processes. Such limitations can be tolerated for scientific applications, but make it impossible to execute applications such as databases that make extensive use of OS functionality.

Since our goal is to extend the range of applications that can be executed on software DSM systems, we are primarily interested in ensuring that the most common system calls execute correctly across the cluster. Our short-term goal of executing the Oracle database on Shasta requires us to support several classes of system calls: calls for managing processes, calls for managing shared memory segments, and calls for accessing a common file system.

Our approach for supporting system calls across a cluster involves replacing the system call routine in the original executable with a routine that implements the new functionality. Regarding process management, Shasta supports system calls such as *fork*, *wait*, *kill*, *pid_block*, *pid_unblock*, and *getpid*. Our *fork* call creates a copy of an existing process that can run on the same node or another node on the cluster. We implement the cluster fork by explicitly copying all of the writable, non-shared data of the parent process (the stack and the static data) to the new process. Because local process ids on different nodes might conflict, Shasta assigns a unique global process id to each of the processes executing an application. When a process exits, Shasta arranges for information to be sent to the parent process so that system calls such as *wait* can be implemented correctly. Shasta also uses messages to implement system calls, such as *kill* and *pid_unblock*, that are used to change the state of other processes.

The system calls for creating and mapping shared memory segments, *shmget* and *shmat*, are implemented by allocating sufficient space in the shared memory region for that segment and returning the starting address of the region. A mapping between the segment id and the region is maintained at each process so that later calls that refer to the segment work correctly. Because the shared memory segment must be allocated in the shared region, Shasta does not support the option of attaching a shared memory segment at a specified address.

The default model of memory provided by Shasta supports applications that share memory among multiple processes via shared memory segments. To support thread-based applications that share the entire address space, Shasta would have to insert inline checks for loads and stores to both the static and the stack data segments. However, the overhead of these extra checks, especially for the stack accesses, may lead to lower performance. Since the stack is not commonly accessed by multiple threads except during thread creation and termination, an interesting alternative is to support shared access to stack segments via a simple page-based protocol (that supports the Alpha instruction set), while using Shasta to support sharing of static and dynamically allocated data.

To support the use of system calls that access files across the cluster, in general we need to implement a distributed file system such as Frangipani [20]. Frangipani provides all nodes with coherent access to a shared set of files and is highly available despite component failures. We do not currently have a distributed file system available for our cluster, and instead approximate a distributed file system by mounting the same filesystems at the same locations on each node via NFS. Accesses to files by different nodes are not kept strictly coherent, because of the caching and buffering required for good NFS performance. However, this functionality is sufficient for running decision support database applications (like TPC-D) which execute mainly read operations on the database.

There are numerous limitations to our cluster extensions to system calls. For example, we do not currently support the passing of open file descriptors between processes. Also, our current implementation of a remote *fork* does not implement the full semantics of the UNIX *fork*. In particular, it does not duplicate process state such as the current open files, memory mappings, signal mask, etc. In addition, there are many system calls which we have not extended at all. However, we believe that the OS requirements of Oracle are extensive enough to bring out many of the important issues. In addition, there are many sets of applications which make use of the same or a lower level of OS services.

4.3 Handling Complex Process Graphs

One of the most interesting issues that arises when supporting complex applications on software DSM systems is dealing with applications that have complex process creation graphs. Software DSM systems have almost exclusively been used to run applications which create a fixed number of processes at startup that is equal to the number of processors and do not create or destroy processes during the remainder of the execution. Such a structure can be easily applied to most scientific applications, but is inappropriate for many commercial applications. For example, a database application such as Oracle creates a number of long-lived "daemon" processes, while "server" processes that do most of the work may be created and destroyed in response to client requests. In addition, the number of database processes may exceed the number of physical processors, because of the use of daemon processes and extra server processes as a way to help hide I/O latencies.

The following section describes some of the issues that arise in the presence of complex process graphs in more detail. We next present our general solution to these problems. Our solution is not specific to Shasta and may apply to other software DSM systems. Next, we describe our implementation in Shasta which includes a number of simplifications to the general solution. Finally, we describe a method for reducing the number of downgrade messages in SMP-Shasta that is especially critical when the number of application processes exceeds the number of processors.

4.3.1 Issues

Along with application data, software DSM systems typically maintain some global protocol information in each application process. For instance, in Shasta's directory-based protocol, each block of application data has an assigned "home" process that is responsible for maintaining the directory information for that block. Application processes that are dynamically created and destroyed raise a number of issues with respect to maintaining such protocol state. First, how should protocol state be preserved if a process is terminating? Second, how should the responsibility for maintaining protocol information be distributed among processes to provide good protocol efficiency? Should protocol responsibility be continually redistributed as processes are created or should a fixed set of processes automatically be created at startup so that they can start maintaining and serving protocol information immediately (even though they may not execute application code until later)? Similar issues arise with respect to application data. For example, how should the application data be preserved if a process is terminating, given that it may hold exclusive copies of some data?

Another set of issues arise when an application creates more processes than processors. Applications may be structured in this way for reasons of security or modularity, or as a way of overlapping computation with I/O operations. Since all requests from other nodes are served in software by the application processes, a request can be greatly delayed if it is sent to a process that is currently not active. Because the time slice of the current process must end before the target process is scheduled [3], the latency of the request can easily increase to several milliseconds (which is two orders of magnitude larger than the nominal latencies in Shasta). We assume here that context switches are caused only by an application's own processes. Problems resulting from switches between processes of different applications can potentially be dealt with through techniques such as dynamic coscheduling [17].

Even if there is only one process per processor, a process can be suspended due to a system call. For instance, a process in a database application may be frequently suspended waiting for I/O system calls to complete or for a signal from another process. Again, a response to any incoming requests will be delayed until the system call completes and the process resumes execution.

A final issue has to do with load balancing. In a complex application, some processes may be very active, while other processes may be mostly inactive. What if one node ends up with more active processes than another node? In general, the most active processes must be distributed evenly among the nodes, and this balance may need to be adjusted dynamically during a run.

[3] We are assuming that incoming messages are detected through polling instead of interrupts, because interrupts are much more expensive and would lead to higher latencies in the frequent cases when the target processes are scheduled.

4.3.2 Our General Solution

In this section, we describe our general solution to the issues described above. In the next section, we describe our greatly simplified but still workable implementation of the general solution.

The general solution is as follows. The user (or application) somehow specifies the pool of nodes on which to run the application and exactly how many processors to use. When the application starts, we immediately create one *protocol process* per processor which remains alive during the entire application. The protocol process and all application processes assigned to the same processor share protocol data structures and memory. As application processes are created, they are mapped to run on a processor in accordance with a particular load-balancing policy. Application processes are allowed to terminate whenever they wish to exit. Since the associated protocol process always exists, no protocol or application data is ever lost as application processes are destroyed.

In order to avoid the problem of long latencies that result when a request is made to a process that is not currently running, it is essential that all processes assigned to the same processor can serve all incoming requests for any of the processes. Each application process (and the protocol process) on a processor is able to access the incoming message queues of all the other processes. The shared access to message queues may require locking that was not previously required for private message queues.

Protocol processes have a lower priority than application processes and run a simple loop that checks for and handles incoming messages. If there are any active application processes, they will take over the processor and serve all incoming messages. However, if there are no application processes (none have been created or all have terminated), then the protocol process will execute and respond to requests. Similarly, the protocol process will run if all application processes are suspended in system calls. Protocol processes are analogous to the shared-memory hardware in a multiprocessor, since they preserve all the protocol and application data and are always available to serve requests.

If a cluster consists of SMP nodes, then we have a choice of using one protocol process per processor or per SMP node. If there is one protocol process per processor, then we associate application processes with specific protocol processes and may attach protocol and application processes to specific processors. In this case, locking costs are reduced, since we only share message queues among processes running on the same processor. If there is only one protocol process per node, then we no longer need to attach processes to specific processes. However, synchronization costs are higher, since the message queues of all processes on the node must be shared.

4.3.3 Our Implementation

We currently have an implementation that simplifies the above general design in a number of ways but still functions usefully. In particular, there are no protocol processes. Instead, the user specifies a fixed number of Shasta processes that are created when an application starts up. This number should be the maximum number of processes that will ever be alive during the application run. The user also specifies the assignment of these processes to processors on the various nodes in the cluster. The user can additionally specify which processes should maintain directory information and serve directory requests. We do not currently have a load-balancing algorithm that moves active processes among nodes, so we require the user to do an assignment that achieves good load balance.

New application processes that are created by *fork* are assigned to the existing Shasta processes. Applications that are assigned to the same processor share application memory and protocol data structures. We essentially use the SMP-Shasta protocol and treat these processes as part of the same SMP (even if they are running on a uniprocessor node). In addition, processes on the same processor use shared message queues, so that any active process can serve incoming messages for all processes. One current limitation of our protocol is that only the process that made a request for data can handle the response to that request.

When an application process terminates, the original Shasta process remains alive and continues to serve requests for its protocol and application data (since there are no protocol processes). It can also be reused to run another application process. However, a terminated process that doesn't receive requests for a while is put to sleep for successively longer time periods to not take CPU time away from other active processes.

We also allow new processes to join an existing group of Shasta processes that are sharing memory. This functionality is important for database and other commercial applications, where server processes can be started up by a new client long after the initial application processes have been started. The joining process notifies the existing processes via a signal that it wants to join the group, so that they establish communication with it.

Using our simplified implementation, we are currently able to start up an Oracle 7.3 database on our cluster using Shasta and run applications modeled after the TPC-B and TPC-D benchmarks. Such runs involve creating a number of daemon processes (as well several processes that die almost immediately) and then creating server processes that do most of the database work. We will give performance results for the Oracle application in Section 6.

4.3.4 Reducing the Number of Downgrades Messages

There is one remaining case in the Shasta protocol where a process may need to contact an inactive process. The SMP-Shasta implementation sometimes needs to send explicit downgrade messages to other processes as part of servicing an incoming request (as described in Section 2.3). Even though Shasta sends these messages selectively, the long latency problem may still arise if the target of the downgrade message is not currently scheduled. We have developed a technique called *direct downgrade* that greatly reduces the number of downgrade messages that must be sent.

We observe that a process P1 can directly downgrade the private state table entry of another process P2 if P2 is not in application code, since then the races described in Section 2.3 cannot occur. Therefore, all protocol routines and system calls set a per-process flag when called and reset the flag when they return to the application. Given proper synchronization, a process P1 can then directly downgrade a private state table entry of process P2 if P2's flag indicates that it is not in application code.[4] This optimization is crucial for cases where processes may block in system calls (e.g. *pid_block*) for long periods of time, since otherwise the response to the original request will be delayed until the process finally wakes up and handles the downgrade message.

5 Code Modification Issues

Shasta depends extensively on code modification both for supporting fine grain coherence and for supporting transparent execution. This section briefly discusses some of the issues related to code modification, which largely arise because the executed code is different from the original application code.

One issue is how and when code modification is triggered. We currently do the Shasta code modification as an extra step in building an application, and explicitly invoke the new executable when we want to execute the application on a cluster. However, this process can be automated by augmenting the system loader to trigger the modification at application load time when the user tries to run the application. The system loaders for most modern UNIX systems already provide a related functionality for automatically linking in any necessary shared libraries.

As with dynamically linking libraries, one issue with doing code modification at load time is the amount of extra time required. We provide some data on code modification times under Shasta in Section 6.3. The loader can address some of the speed issues by caching translations of the most commonly executed applications in the file system. This technique is also useful for ensuring that multiple invocations of the same application share the same modified text image. For systems that use shared libraries, the loader can

[4] The protocol keeps track of the shared-memory addresses that may be accessed by a system call, and disallows this optimization if the line that is to be downgraded is within those ranges.

also cache translations of the most common shared libraries, thereby reducing the amount of code that must be modified when a new application is executed.

Code modification may also become an issue for developers who attempt to debug a Shasta application. When it modifies an application, our version of ATOM correctly updates the symbol table for the application to reflect the code changes. Therefore, source code debugging functions normally, and code changes are completely invisible for a programmer debugging at the source code level. [5] The code modifications are currently visible if the programmer debugs at the machine code level, but the code changes could be mostly hidden even at this level with some modifications to the debugger.

A final problem with code modification relates to applications that actually examine or generate their own code. For example, an application may do a checksum on its own code as a security measure, or may dynamically generate code as a way of improving its performance. Since there is no easy solution for cases such as these, Shasta cannot necessarily be used to execute such applications.

6 Performance Results

This section presents performance results for the Shasta implementation. We first describe our prototype SMP cluster. We next provide results for a few microbenchmarks that characterize the cost of transparently executing hardware binaries. The next set of results characterize the static and dynamic overheads associated with inline checks for SPLASH-2 applications and the Oracle database. Finally, we present a number of parallel performance results for the SPLASH-2 applications and for Oracle.

6.1 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. Each node in our cluster is connected to a single network link.

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. In the base Shasta protocol, the minimum latency to fetch a 64-byte block from a remote node (two hops) via the Memory Channel is 20 microseconds, and the effective bandwidth for large blocks is about 35 Mbytes/s.

[5] The use of Shasta is visible in the debugger, however, in that some data may be invalid in the local process, and this data is not automatically fetched by Shasta when examined by the debugger.

	MP locks	**SM locks**	**SM locks with prefetch**
Cached latency	1.11	1.88	1.91
Uncontended miss latency	15.63	44.12	25.70
Contended miss latency	81.02	136.48	137.90

Table 1: Lock acquire latencies (in microseconds).

6.2 Synchronization and Validation Costs

Shasta supports two different ways for applications to do synchronization. Applications can make use of high-level lock and barrier routines provided by Shasta that use an efficient message-passing protocol. Alternatively, as described in Section 3.1, applications can use atomic Alpha instructions which are transparently supported by Shasta. As a way of measuring the costs of supporting transparency, Table 1 shows the average time to acquire a lock via the message-passing protocol (labeled MP) and via Alpha load-locked and store-conditional instructions (labeled SM for shared memory) in SMP-Shasta. The last column represents SM locks augmented with a single prefetch-exclusive before the load-locked instruction (as described in Section 3.1.2). The first row gives the time for acquiring a lock that is free and is cached locally. Even though the load-locked and store-conditional are executed in hardware for SM locks, the protocol must still be called to enforce the memory barrier operation that is called after acquiring the lock. The second row gives the time for acquiring a free lock that resides on a remote node. The MP locks have the lowest latency, because they require sending only a single request to the remote node. SM locks result in two round-trip requests, one for the load-locked and another for the store-conditional. SM locks with the prefetch-exclusive optimization have a lower latency than standard SM locks because they eliminate one of the round-trip requests. The last row gives the time for acquiring a lock from a remote node when there is contention. The MP locks have lowest latency under contention as well, because they are queue-based. The prefetch-exclusive optimization does not help in this case because the lock is not free when the prefetch is issued. As expected, the message passing implementation is superior to transparently supporting the Alpha synchronization instructions. However, as we will see, the effect of this difference on the overall performance of an application can be much smaller.

We have measured the cost of doing a memory barrier with no outstanding stores pending as 0.32 microseconds for Base-Shasta and 1.68 microseconds for SMP-Shasta (vs. 0.03 microseconds for a standard SMP application). The cost for a memory barrier in Base-Shasta is to make a call into the

	Standard app	Shasta app	
		Base-Shasta	SMP-Shasta
Open	58	66	79
Read of 4 bytes	12	16	20
Read of 8192 bytes	51	70	126
Read of 65536 bytes	370	576	845

Table 2: Average times for system calls (in microseconds) for standard and Shasta applications.

protocol to check if there are any outstanding requests. The extra cost for a memory barrier in SMP-Shasta is because of the use of per-processor request counts as a way of reducing contention during individual load and store misses. The cost of a memory barrier could potentially be reduced by making the appropriate check directly in inline code. In the case of SMP-Shasta, the protocol would have to modified to allow a simpler memory barrier check. Alternately, we could completely eliminate the memory barrier check by making the Shasta protocol sequentially consistent (in particular, stalling on all store misses).

We have also measured the costs of validating the arguments to system calls. Table 2 gives the average times to execute an *open* system call, and a *read* of 4 bytes, 8192 bytes, and 65536 bytes. The first column gives times for a standard application, and the second and third columns give times for a Shasta application using Base-Shasta and SMP-Shasta, respectively. The file name argument of the open call and the read buffer argument of the read call are in shared memory, so the Shasta times include the validation overhead. The times are higher for SMP-Shasta because of locking costs. The Shasta validation overheads are certainly measurable, but not excessive. In addition, these results are for files that have been recently accessed, so no disk operations are involved. The relative overhead of validation would be much less for system calls that accessed the disk.

6.3 Applications and Overhead Measurements

We report results for nine of the SPLASH-2 applications [21]. Table 3 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 3 also shows the single processor execution times for each application after the Shasta miss checks are added, along with the percentage increase in the time over the original sequential time (which averages 21.7%). [6] The last column indicates the increase in the static code size due to the Shasta miss checks.

The last three rows of the table gives the overheads for Oracle 7.3 executing a transaction processing application (OLTP) modeled on the TPC-B database benchmark and decision support queries (DSS-1 and DSS-2) modeled on the TPC-D database benchmark. The first number in each row reports the time for the standard Oracle executable to run on a single Alpha processor. The second number gives the time for the Shasta version of the Oracle executable to perform the same run on a single processor. Although multiple processes are used in the Oracle run, we set up the processes to still share memory via UNIX shared memory segments rather than via Shasta so that we can isolate the checking overhead. Therefore, the indicated overhead corresponds solely to the extra cost of doing the inline Shasta checks and polls. The overhead for DSS-1 is fairly high. We believe this effect is because the DSS-1 benchmark has fairly good locality (as it searches entire tables), but does not have any simple inner loop whose accesses can be batched. The overhead of DSS-2, a much larger query, is somewhat lower.

We also measured the time to generate the new Shasta executables. For the SPLASH-2 applications, which have from 255 to 485 procedures, the time ranges from 4.0 to 7.3 seconds. About 3 seconds of that time is the cost of reading in the old executable and writing out the new executable, and the remainder is the overhead for doing the Shasta analysis and code insertion. For Oracle 7.3, which has over 12000 procedures, the conversion time is 202 seconds. Of this time, about 26 seconds is to read the old and write the new executable, 104 seconds is to do the necessary dataflow analysis, and 72 seconds is to do the other Shasta analysis and code insertion. We have not optimized any of the Shasta processing phases, and believe that the dataflow analysis routines and the other Shasta analysis routines can be significantly sped up. The code modification delays for the SPLASH-2 applications seem acceptable, especially if optimizations lower them to 2-3 seconds. While the modification time for Oracle is large, we assume this initial conversion time is not significant for such production applications which are executed for long periods of time.

6.4 SPLASH-2 Parallel Performance

This section presents the parallel performance of the SPLASH-2 applications running on Shasta. In our results, two- and four-processor runs always execute entirely on a single node, and 8-processor runs use two nodes. We are using the Shasta SMP protocol (SMP-Shasta) that allows processes on the same SMP node to share application data through the hardware coherence mechanism. Processors on the same node share the Memory Channel bandwidth when sending messages to destinations on other nodes. We use a fixed Shasta line size of 64 bytes. For FMM, LU-Contiguous and Ocean, we use the standard home placement optimization, as is done in most studies of the SPLASH-2 applications.

Figure 3 shows the speedups for the unmodified applications running on our prototype cluster. The speedups shown are based on the execution time of the application running via

[6] The running times of LU and LU-Contig are much shorter than those in previous papers [12, 13] because of much better optimization by the compiler in Digital Unix 4.0. The more efficient sequential runs also reduce the parallel speedups attainable with Shasta for a given input size.

	problem size	sequential time	with Shasta miss checks	code size increase
Barnes	16K particles	9.19s	10.08s (9.6%)	59%
FMM	32K particles	14.43s	16.69s (15.6%)	58%
LU	1024x1024 matrix	15.64s	19.93s (27.4%)	65%
LU-Contig	1024x1024 matrix	9.97s	13.55s (35.9%)	57%
Ocean	514x514 ocean	10.55s	13.86s (31.3%)	111%
Raytrace	balls4	65.50s	81.41s (24.2%)	67%
Volrend	head	1.67s	1.71s (2.3%)	65%
Water-Nsq	1000 molecules	8.30s	10.26s (23.6%)	59%
Water-Sp	1728 molecules	6.37s	8.06s (26.5%)	60%
Oracle	OLTP	31.09s	37.06 (19.2%)	96%
Oracle	DSS-1	8.83s	14.85s (68.1%)	96%
Oracle	DSS-2	83.76s	114.93 (37.2%)	96%

Table 3: Sequential times and checking overheads for the SPLASH-2 applications and Oracle.

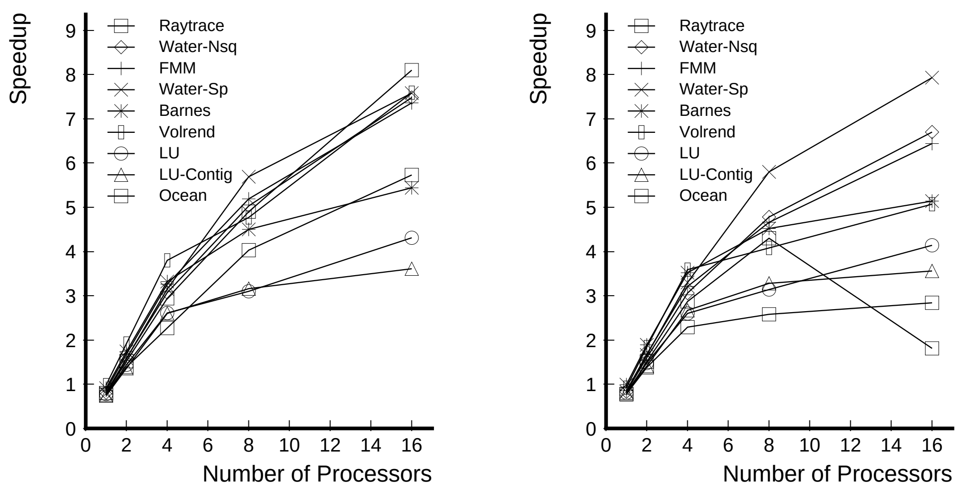

Figure 3: Speedups of SPLASH-2 applications with message-passing (left) and Alpha (right) synchronization.

Shasta on 1 to 16 processors relative to the execution of the original sequential application (with no miss checks). The left graph gives speedups for the SPLASH applications when using the message-passing version of locks and barriers. The right graph gives speedups for SPLASH-2 binaries that are compiled for Alpha hardware multiprocessor platforms and hence use load-locked, store-conditional, and memory barrier instructions to achieve synchronization and ordering. Overall, the speedups achieved by Shasta are quite promising given the extremely fast processors (300MHz Alpha 21164) and the small problem sizes used in this experiment.

When using the native Alpha binaries, 16-processor runs of six applications slow down by just 2-10%. However, 16-processor runs of Raytrace, Volrend, and Ocean slow down by 78%, 50%, and 34% respectively. Raytrace slows down because it has a custom memory allocator protected by a single lock which is highly contended, and for which the queue-based message-passing implementation performs much better. Volrend also slows down because of a few highly contended locks. Ocean slows down because of a high rate of executing barriers, which can also lead to contention, since the barrier implementation requires each processor to increment the barrier count atomically. We found that doing a prefetch-exclusive prior to a load-locked/store-conditional sequence, as described in Section 3.1.2, speeds up some of the lock-intensive applications by 3-7%; however, the applications that exhibit high contention locks or frequent barriers can slow down by up to 20%. One possible technique is to use runtime information to do prefetches only for addresses which do not have a lot of contention.

In contrast to page-based software DSM systems, the performance of Shasta is quite insensitive to the underlying memory consistency model. To illustrate this point, Figure 4 shows normalized execution times for 16-processor runs (on Base-Shasta) of the SPLASH-2 applications with blocking stores (labeled SC for sequential consistency) and with nonblocking stores (labeled RC for a relaxed model such as release consistency). The figure also shows the breakdowns of the execution into time spent executing the application, time spent stalled for reads, time spent stalled for writes (because of limits on the number of outstanding writes), time spent stalled on synchronization, and time spent handling messages while not stalled. The loss in performance from using a more strict memory consistency model is at most 10% across the SPLASH-2 applications. This result indicates that Shasta is also suited for executing commercial binaries

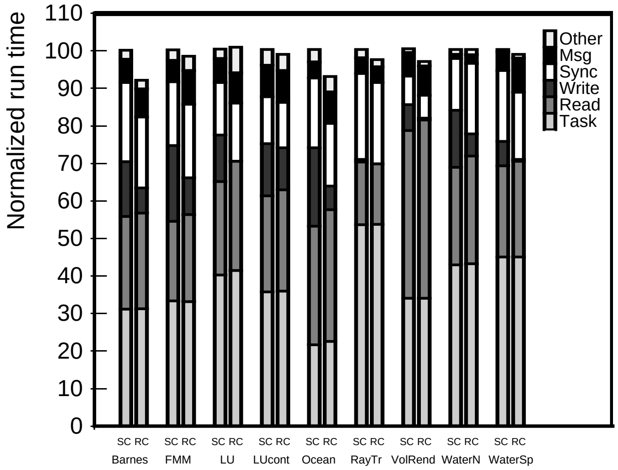

Figure 4: Effect of nonblocking stores for 16-processor runs.

that require a strict memory consistency model (e.g., Intel x86). A similar experiment with a typical page-based system would exhibit a much larger reduction in performance, because page-based systems heavily depend on relaxed models to alleviate problems such as false sharing that arise due to the page-sized coherence granularity.

6.5 Oracle Performance

We have only preliminary results for running Oracle on Shasta, because our implementation still suffers from a number of limitations. First, our current protocol implementation allows a process to serve most incoming messages directed to another process on the same SMP, but does not allow a process to handle replies to another process' request for data. The active process may therefore stall at a memory barrier because a store made by a now inactive process can not yet completed. Second, we do not do dynamic load balancing of processes. In Oracle, a server process will often block to allow a daemon process to complete its request. However, if the daemon process is not on the same node as the server processor, then the daemon process will not be able to take advantage of the idle processor. Because of these limitations, our Oracle runs are not performing as well as they might. Finally, we do not currently have a distributed file system across the cluster. Since transaction processing runs do frequent writes to the database and require a coherent file system across the nodes, we are able to execute OLTP (modeled after TPC-B) only when all processes are on the same node (but still using Shasta to share memory). We therefore only report results for decision support runs.

Table 4 gives a small set of results when the DSS-1 query is run on Oracle using one, two, or three servers to compute the query response in parallel. These results are for a query which is searching tables that are already cached in memory by the database. Because the data being analyzed is cached, the main server processes spend almost no time in I/O system calls. However, some of the daemon processes do read and write system calls as part of the normal database operation.

	Oracle on SMP	**Oracle on Shasta**	
		extra proc	1 proc/server
One server	8.83s	15.51s	15.40s
Two servers	4.77s	12.57s	19.29s
Three servers	3.06s	8.11s	11.11s

Table 4: Run times for DSS-1 on Shasta with varying numbers of servers.

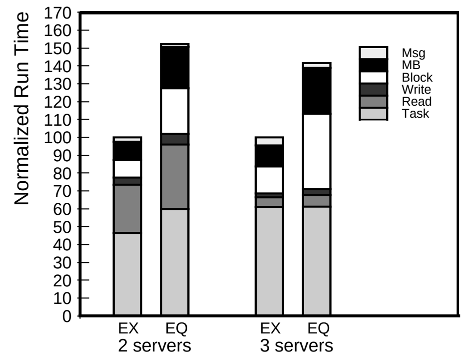

Figure 5: Time breakdowns for runs of DSS-1.

The first column gives the times when the query is run on standard Oracle on a single AlphaServer of our cluster using as many processors as servers. The second column gives times when Oracle is running on SMP-Shasta. All of the database daemons and one of the servers is placed on one AlphaServer, while the second and third servers are placed on the second AlphaServer. However, we use one extra processor on the first AlphaServer so that the most active daemons do not have to context-switch with the server. Finally, the last column gives Oracle running on Shasta across two AlphaServers, but using exactly one processor per server. That is, all the daemons run on the same processor as the first server. Figure 5 gives time breakdowns for two and three server runs when using an extra processor (EX) and equal numbers of servers and processors (EQ). The segments give the time executing the application ("task" time), time spent stalled for reads, time spent stalled for writes, time spent explicitly blocked in *pid_block*, time spent waiting at a memory barriers (for stores to complete), and time spent handling messages while not stalled. The bars are normalized so that the time for the EX runs is 100%.

We note the base Oracle performance scales well with increasing numbers of servers, because the query is fully parallelizable, though the servers do interact somewhat in accessing common database control structures. We also get speedup when we run Oracle on Shasta across the cluster with one extra processor. However, because of the Shasta checking overhead and communication of the server with the daemons, three servers are required to get only slightly better performance than a one-server Oracle run on an SMP. We note from the time breakdowns that the read stall time goes down between two and three server runs, because of sharing via hardware between the second and third servers. When the same number of processors as servers are used, we no longer get a performance improvement when going from one to two servers. Both the time spent blocked and time spent stalled at memory barriers go up significantly, because of the load balancing and message handling limitations described above. The average latency of a read request also goes up, from 244 μs to 323 μs for two servers, and 103 μs to 203 μs for three servers. The majority of read requests have latencies of 20-40 μs, but the average is increased by requests that are delayed from effects due to context switching. The direct downgrade optimization of Section 4.3.4 is very important for all these runs; when it is turned off, all of the runs take so long that we did not measure them.

This workload is not very typical of decision support workloads, because it is so small and because all the data is already cached in memory. In addition, we would expect worse behavior for transaction processing workloads where the database is modified frequently and there is much more sharing between server processes. The main point of these results is to illustrate that speedup is possible for database code running on top of Shasta.

7 Related Work

Shasta's basic approach to checking loads and stores is derived from the Blizzard-S work [15]. However, we have substantially extended the previous work in this area by developing several techniques for reducing the otherwise excessive checking overheads. We have also designed an efficient protocol that exploits a relaxed consistency model, supports multiple coherence granularities in a single application, and executes efficiently on SMP clusters. Finally, we have developed methods to transparently execute unmodified multiprocessor executables.

There are a variety of other software DSM systems that use the virtual memory hardware to detect access to data that is not available locally in the correct state [1, 2, 4, 7, 9, 19]. None of these systems have focused on transparently executing SMP binaries. Most of these page-based systems make use of aggressive protocol optimizations in order to minimize the false sharing problems that can arise because of the large coherence granularity. As described in Section 3, most of these protocol optimizations violate the semantics of memory models for commercial processor architectures. These systems would therefore have to use substantially less efficient protocols in order to correctly execute unmodified SMP executables. However, our solutions for extending system calls across the cluster and for dealing with applications that dynamically create and destroy processes can potentially be adapted for use by other software DSM systems.

There are a number of distributed operating systems that have been developed to make a cluster of machines appear as a single machine with a single operating system [8, 10, 11, 23]. These systems have concentrated on extending nearly all operating system services to achieve identical functionality regardless of where they are invoked in the cluster. These extensions are typically implemented through kernel modifications that add a global communication layer among the nodes and route kernel requests to the appropriate node. The above systems do not typically attempt to support shared memory between processes on different nodes, though a few systems do support sharing via virtual memory mechanisms using a simple protocol similar to Ivy [9].

We have not attempted to support a full distributed operating system in Shasta. We have instead focused on supporting shared memory efficiently between processes, and on supporting the necessary operating system services to run interesting applications such as databases. In addition, our method for extending operating system services is to modify application executables, rather than to modify the kernel. This approach allows us to run on commodity hardware and operating systems.

8 Conclusion

A key goal of the Shasta project is to address the issues that can increase the commercial viability of software DSM systems, namely good performance and a sufficiently large application base. Our previous work on Shasta has explored a large number of optimizations to achieve better performance, including efficient support for fine-grain coherence [14], effective protocol optimizations [12], and effectively exploiting a cluster of SMP nodes [13]. With respect to increasing the application base, we believe that the most promising way of addressing this issue is to support transparent execution of the increasing number of application binaries available for hardware DSM systems.

This paper describes the challenging issues that arise in transparent execution of binaries, which include supporting the full user-level instruction set for a commercial architecture and extending OS services across a cluster to provide a seamless view to an application. We also describe the solutions that we have implemented in the Shasta system. Shasta fully supports the Alpha instruction set architecture, including atomic memory operations and the Alpha memory model. Due to the large amount of effort necessary to extend all OS services, we chose a short-term goal of supporting sufficient functionality to execute a commercial database such as Oracle, which still uses a relatively rich set of OS services. We have extended the necessary system calls for managing processes, shared memory segments, and files, and deal with issues that arise when applications dynamically create and destroy processes. In addition, Shasta validates system call arguments to ensure that the referenced data is available before the system call is made. The solutions we have adopted in Shasta are greatly simplified by our ability to modify binaries, which is also used to support fine-grain coherence.

The Shasta system is fully functional on our cluster of SMPs and can transparently execute SPLASH-2 binaries that run on an Alpha multiprocessor. We can also run Oracle 7.3 across a cluster using Shasta, including runs that are modeled on the TPC-B and TPC-D benchmarks. Our performance results demonstrate that supporting transparent binary execution has performance costs but still allows for good performance. Shasta's ability to support coherence at a fine granularity plays a fundamental role in this result, by allowing for efficient support of the memory models of commercial processor architectures. Our research on transparently executing commercial binaries raises a number of previously unexplored issues in the design of software DSM systems, and in their interaction with OS services, which we hope will interest researchers in both communities.

Acknowledgments

We would like to thank John Heinlein and Anshu Aggarwal for help with the implementation, Luiz Barroso for assistance in setting up and using Oracle, and Marc Viredaz and Drew Kramer for their help in maintaining our cluster of AlphaServers. We also thank the anonymous referees for their comments.

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] 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.

[8] Y. A. Khalidi, J. M. Bernabeu, V. Matena, K. Shirriff, and M. Thadani. Solaris MC: A MultiComputer OS. In *Proceedings of the USENIX 1996 Annual Technical Conference*, pages 191–204, San Diego, CA, Jan. 1996.

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

[10] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch. The Sprite Network Operating System. *IEEE Computer*, Feb. 1988.

[11] G. Popek and B. Walker. *The LOCUS Distributed System Architecture*. MIT Press, 1985.

[12] D. J. Scales and K. Gharachorloo. Design and Performance of the Shasta Distributed Shared Memory Protocol. In *Proceedings of the 11th ACM International Conference on Supercomputing*, July 1997. Extended version available as Western Research Laboratory technical report 97/2 (Feb. 1997).

[13] D. J. Scales, K. Gharachorloo, and A. Aggarwal. Fine-Grain Software Distributed Shared Memory on SMP Clusters. Technical Report 97/3, Western Research Laboratory, Digital Equipment Corporation, Feb. 1997.

[14] 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.

[15] 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.

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

[17] P. Sobalvarro. *Demand-based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors*. PhD thesis, MIT Laboratory for Computer Science, Feb. 1997.

[18] A. Srivastava and A. Eustace. ATOM: A System for Building Customized Program Analysis Tools. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 196–205, June 1994.

[19] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Oct. 1997.

[20] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A Scalable Distributed File System. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Oct. 1997.

[21] 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.

[22] 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.

[23] R. Zajcew et al. An OSF/1 UNIX for Massively Parallel Multicomputers. In *Proceedings of the Winter 1993 USENIX Conference*, pages 449–468, Jan. 1993.