Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory

Daniel J. Scales and **Kourosh Gharachorloo**
Western Research Laboratory
Digital Equipment Corporation

Chandramohan A. Thekkath
Systems Research Center
Digital Equipment Corporation

Abstract

This paper describes Shasta, a system that supports a shared address space in software on clusters of computers with physically distributed memory. 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. In addition, the system allows the coherence granularity to vary across different shared data structures in a single application. Shasta implements the shared address space by transparently rewriting the application executable to intercept loads and stores. For each shared load or store, the inserted code checks to see if the data is available locally and communicates with other processors if necessary. The system uses numerous techniques to reduce the run-time overhead of these checks. Since Shasta is implemented entirely in software, it also provides tremendous flexibility in supporting different types of cache coherence protocols. We have implemented an efficient cache coherence protocol that incorporates a number of optimizations, including support for multiple communication granularities and use of relaxed memory models. This system is fully functional and runs on a cluster of Alpha workstations.

The primary focus of this paper is to describe the techniques used in Shasta to reduce the checking overhead for supporting fine granularity sharing in software. These techniques include careful layout of the shared address space, scheduling the checking code for efficient execution on modern processors, using a simple method that checks loads using only the value loaded, reducing the extra cache misses caused by the checking code, and combining the checks for multiple loads and stores. To characterize the effect of these techniques, we present detailed performance results for the SPLASH-2 applications running on an Alpha processor. Without our optimizations, the checking overheads are excessively high, exceeding 100% for several applications. However, our techniques are effective in reducing these overheads to a range of 5% to 35% for almost all of the applications. We also describe our coherence protocol and present some preliminary results on the parallel performance of several applications running on our workstation cluster. Our experience so far indicates that once the cost of checking memory accesses is reduced using our techniques, the Shasta approach is an attractive software solution for supporting a shared address space with fine-grain access to data.

1 Introduction

There has been much recent interest in supporting a shared address space in software across distributed-memory multiprocessors or workstation clusters. 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. For example, some systems require programmer annotations or explicit calls to access shared data [2, 11]. Another approach, called Shared Virtual Memory (SVM), uses the virtual memory hardware to detect access to data that is not available locally [4, 13, 12]. In most such systems, the granularity at which data is accessed and kept coherent is large, because it is related to the size of an application data structure or the size of a virtual page.

We have developed a system called Shasta to investigate the feasibility of supporting fine-grain sharing of data entirely in software. 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. Shasta uses the basic approach of the Blizzard-S system from Wisconsin [18]. Shasta implements the shared address space by inserting checks in an application executable at loads and stores. This inline code checks if the data is available locally and sends out the necessary messages to other processors if the operation cannot be serviced locally.

Minimizing the cost of the access control checks is critical to making the above approach viable. Without our optimizations, overheads associated with these checks range from 50 to 150% in many applications, even for carefully coded inline checks that consist of seven instructions. To address the above problem, we have implemented a variety of techniques that greatly reduce checking overhead and make it feasible to support shared memory with fine-grain access in software. These techniques include (1) careful layout of the address space, (2) instruction scheduling of the checking code with respect to the application code, (3) using a novel technique to do load checks based on the value loaded, (4) applying a method to reduce extra cache misses caused by the checking code, and (5) "batching" together checks for multiple loads and stores.

The above optimizations are extremely effective in lowering the overheads due to access checks. Our results show these overheads can be reduced to a range of 5% to 35% on a single processor for almost all of the SPLASH-2 applications. These results are encouraging for several reasons. First, there is still much room for more aggressive compiler optimizations, including scheduling application and checking instructions better and extending our techniques to use interprocedural analysis. Second, more of the checking overhead may be hidden on more modern processors with dynamic scheduling capability [10]. Finally, the relative effect of the checking overhead is typically less on the parallel execution time due to the overheads arising from communication and synchronization.

Since the shared address space is supported completely in soft-

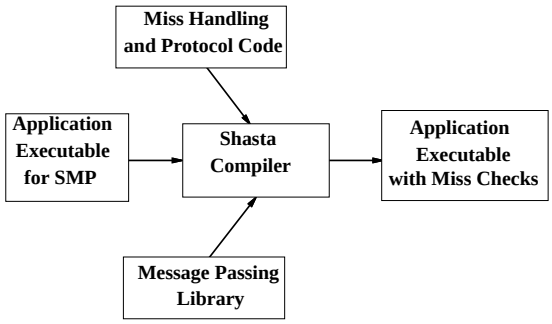

Figure 1: Shasta compilation process.

ware, Shasta provides a flexible framework for experimenting with a variety of cache coherence protocol optimizations to improve parallel performance. One of the unique aspects of the cache coherence protocol that we have developed (relative to other software protocols that transparently support a shared address space) is its ability to support different coherence granularities for different shared data structures within a single application. Our protocol also includes optimizations, such as non-stalling stores, that exploit a relaxed memory model.

We have implemented the Shasta system on clusters of Alpha workstations connected by an ATM network or a Memory Channel network [9]. We use a modified version of ATOM [20] that can insert any Alpha instructions at any point in the executable. As illustrated in Figure 1, our Shasta compiler automatically modifies application executables developed for a hardware shared-memory multiprocessor so that they run on a cluster of workstations.

The following section describes the implementation concepts of the Shasta system, including the basic method of checking the loads and stores. Section 3 describes in detail our techniques for reducing the checking overheads. We discuss optimizations to the basic cache coherence protocol in Section 4. Section 5 analyzes the effectiveness of our techniques by presenting actual checking overhead results for the SPLASH-2 applications. It also provides some preliminary parallel performance results. Finally, we describe related work and conclude.

2 Basic Design

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. In the current Shasta system, we have adopted the memory model of the original SPLASH applications [19]: data that is dynamically allocated is shared, but all static and stack data is private. The use of this kind of model reduces overhead by avoiding checks on loads and stores to private data, such as the stack.

2.1 Cache Coherence Protocol

The Shasta system requires a protocol to ensure coherence among cached copies of shared data. 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.

Data can be read on the local processor if the data is in the shared or exclusive state; data can be written on the local processor only if the data is in the exclusive state. 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*. The checks that Shasta inserts in the application executables at each load and store are shared miss checks on the data being referenced.

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. The size of each block must be a multiple of the fixed line size. The fixed size of lines makes it easy to map from an address to the location of the state table entry for the corresponding line. Furthermore, on each processor, all lines in a given block have the same cache state because data is kept coherent at the block granularity.

In our current implementation, coherence is maintained using a directory-based invalidation protocol. A home node is associated with each virtual page of shared data. The homes for contiguous virtual pages are allocated in a round-robin fashion by default. We also allow the application to explicitly specify the home for individual pages. The protocol supports three types of requests: *read*, *read-exclusive*, and *exclusive* (or *upgrade*). Supporting exclusive requests is an important optimization since it reduces message latency and overhead if the requesting processor already has the line in shared state. We also currently support three types of synchronization primitives in the protocol: locks, barriers, and event flags. These primitives are sufficient for supporting the SPLASH-2 applications.

Each node maintains *directory* information for the shared data pages assigned to it. The protocol maintains the notion of an *owner* node for each line, which corresponds to the last node that maintained an exclusive copy of the line. The directory information consists of two components: (i) a pointer to the current owner node, and (ii) a full bit vector of the nodes that are sharing the data. Our protocol supports *dirty sharing*, which allows the data to be shared without requiring the home node to have an up-to-date copy. A request that arrives at the home is always forwarded to the current owner node; as an optimization, we avoid this forwarding if the home node has a copy of the data. Finally, our protocol depends on point-to-point order for messages sent between any two nodes.

Given the relatively high overheads associated with handling messages in software DSM implementations, we have designed our protocol to minimize extraneous coherence messages. For example, because the current owner node specified by the directory guarantees to service a request that is forwarded to it, we can complete all directory state changes when a request first reaches the home. Our protocol therefore eliminates extra messages (common in hardware shared-memory protocols) sent back to the home to confirm that a forwarded request was satisfied. In addition, the number of invalidation acknowledgements that are expected for an exclusive request is piggybacked on one of the invalidation acknowledgements to the requestor instead of being sent as a separate message.

2.2 Polling

Because of the high cost of handling messages via interrupts, messages from remote nodes are serviced through a polling mechanism. Our current implementation polls for incoming messages whenever the protocol waits for a reply. To ensure reasonable response times, we also insert polls either at every function call or at every loop backedge. On both our ATM cluster and Memory Channel cluster, polling is inexpensive (three instructions), because we have arranged for a single cachable location that can be tested to determine if a message has arrived.

Our use of polling also simplifies the inline miss checks, since the Shasta compiler ensures that there is no handling of messages during a shared miss check. The miss checks are therefore executed atomically with respect to the message handlers and do not require extra instructions to disable message handling. Implementations that handle messages via interrupts may incur a somewhat higher overhead for miss checks due to extra instructions that are needed for ensuring atomicity.

2.3 Deciding Which Loads and Stores to Instrument

We will refer to adding a shared miss check in the executable for a load or store as *instrumenting* the load or store. Shasta does not need to instrument accesses to non-shared (i.e., private) data, which includes all stack and static data in our current implementation. Therefore, we do not instrument a load or store that uses the stack pointer (SP) as its base register. Similarly, since the global pointer (GP) register on the Alpha always points into the static data area, we do not instrument a load or store that uses the GP as its base register. The Shasta compiler does dataflow analysis that tracks the contents of other processor registers during a procedure to determine whether the value of a register was calculated using the value of the SP or GP. Loads and stores that use a base register whose current value was calculated using the contents of the SP or GP are also not instrumented. Since our compiler does not currently do the necessary interprocedural analysis, we conservatively treat a register's contents as undefined if the register is saved to the stack and restored after a procedure call.

2.4 Implementation of the Shared Miss Check

In the simplest implementation of the shared miss check, the inline code must determine the line corresponding to the target address and check the state of that line. As an optimization, the code may first check if the target address is in the shared memory range. The advantage of doing the range check is that the miss check can finish immediately if the address is not in the shared memory range and the system does not need to maintain state information for any of the local, non-shared data.

Figure 2 shows Alpha assembly code that does a store miss check. This code has already been optimized in a number of ways. First, we have chosen to make the exclusive state be represented by a zero, so we can test for that state with a single instruction. Second, the code does not save or restore any registers. The Shasta compiler does live register analysis to determine which registers are unused at the point where it inserts the miss check and uses those registers (labeled `rx` and `ry` in the figure). Although it can insert code to save registers as necessary, we have found that there are sufficient free registers to use for the miss checks in virtually all cases.

Finally, the code in Figure 2 has been simplified by using the address space layout shown in Figure 3. In our current implementation, we place all shared data (and no private data) in each processor's address space above $0x8000000000$. It is therefore possible to check if an address is in the shared memory range by a simple shift. We also place the state table such that shifting the target address by a number of bits corresponding to the line size directly gives us the address of the state table entry. For a line size of 64 bytes, we place the state table at address $0x8000000000/2^6 = 0x200000000$. Alpha's 64-bit address space makes the layout of the address space somewhat easier, but it should be possible to lay out memory in a similar way in a 32-bit address space.

```
1.  lda     rx, offset(base)
2.  srl     rx, 39, ry
3.  beq     ry, nomiss
4.  srl     rx, 6, rx
5.  ldq_u   ry, 0(rx)
6.  extbl   ry, rx, ry
7.  beq     ry, nomiss

8. ... call function to handle store miss

9. nomiss:
10. ... original store instruction ...
```

Figure 2: Optimized code for a store miss check.

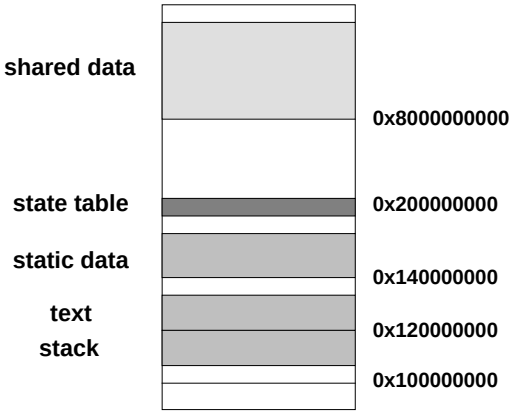

Figure 3: Shasta memory layout on Alphas.

In line 1 of Figure 2, we calculate the target address of the load using the offset and base register of the original load instruction. Line 1 can be eliminated if the offset of the store is zero. In lines 2 and 3, we skip to the non-miss code if the address is not in the shared memory range. Line 4 determines the address of the state table entry for the data being referenced. Lines 5 and 6 are the instructions that load and extract the byte-size state table entry (the Alpha 21064A and 21164 processors do not have a direct byte-load instruction). In the inline miss handling code, we load a register with the target address of the store and call a software handler to serve the miss for that address. The handler always saves all integer registers so as not to interfere with the state of the application; no floating-point operations occur in any of the handlers, so the floating-point registers do not need to be saved.

3 Optimizations to Reduce Check Overhead

In this section, we describe a variety of techniques that we have used to further reduce the overhead of doing the miss checks. We first describe instruction scheduling techniques that reduce the cycle

```
1.  lda     rx, offset(base)
2.  srl     rx, 39, ry
3.  srl     rx, 6, rx

4.  beq     ry, nomiss
5.  ldq_u   ry, 0(rx)
6.  extbl   ry, rx, ry
7.  beq     ry, nomiss

8. ... call function to handle store miss

9. nomiss:
10. ... store instruction ...
```

Figure 4: Rescheduled store miss check code.

overhead of the miss checks. We then describe a novel method of doing the load miss check and a method of reducing cache misses on stores. Finally, we describe a method of combining the checking for multiple loads and stores. Our emphasis is always on reducing the cost of a check when there is no shared data miss, since that is the common case.

3.1 Instruction Scheduling

Instruction scheduling techniques can be used to reduce the overhead of the miss check code. Because most modern processors are pipelined and superscalar, the added code should be arranged to have minimum pipeline delays and maximum potential for multiple issue. For example, on the Alpha 21064A processor, there is a one-cycle delay before the result of a shift operation can be used. Therefore, it is clearly beneficial to change the code to that shown in Figure 4, so that the second shift occupies the delay slot of the first shift and the stall between the second shift and the `ldq_u` is eliminated.

The overhead of the added code can also potentially be reduced by placing the extra instructions at points where there are pipeline stalls or unused issue slots in the original application code. For example, as long as registers `rx` and `ry` remain unused, the first three instructions of the code in Figure 4 can be moved upward from the instrumentation point within a basic block without affecting the application code. These instructions can therefore be moved up among the application code to potentially hide their overhead. We currently do a limited version of this optimization. The current Shasta system requires that the store being instrumented is executed before the store miss routine is called (if there is a miss). Therefore we place the lines 1 through 3 of the miss check before the store (line 10) and the remaining instructions after the store. The advantage of this placement is that there may be a long-latency instruction immediately preceding the store in the application code that calculates the value being stored. The store instruction must stall until this preceding instruction completes, so the overhead of the first three instructions may therefore be partially or completely hidden.

3.2 Improving Load Checks

We use another key technique that dramatically reduces the overhead for the load miss check. Whenever a line on a processor becomes invalid, we store a particular "flag" value in each longword (4 bytes) of the line. Instead of doing the range check and looking at the state table entry, the miss check code for a load can just compare the value loaded with the flag value. If the loaded value is not equal to the flag value, then there is no miss (i.e., the data is 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. The state check distinguishes an actual miss from a "false miss" (i.e., when the application data actually contains the flag value), and simply returns back to the application code in case of a false miss. Since false misses almost never occur in practice, the above technique can greatly reduce the load miss check overhead.

The Alpha 21064A and 21164 support two granularities for loads operations: longwords (4 bytes) and quadwords (8 bytes). By storing the flag value in each longword of the line, we can handle both cases using the code in Figure 5(a) In this code, line 1 is the original load instruction in the application and `rx` is a free register. We chose the value -253 as our "flag value" so that we can determine if the loaded value equals the flag with a single add instruction. Line 2 in the figure is an add-long instruction that drops the top 32 bits of the source register, thus handling the case of a quadword load. We chose the value -253 without any experimentation and have seen almost no matches to the flag value (i.e., false misses) in our applications. It is likely that almost any constant that is not zero and not a small positive integer would also work.

Besides reducing the instruction overhead greatly, the "flag" technique has a number of other advantages. The main advantage is that we have eliminated the load of the state table entry. Therefore, we do not cause any additional data cache misses beyond what would occur in the application code. In addition, the flag technique essentially combines the load of the data and the check of its state into a single atomic event. This property can potentially simplify other Shasta implementations that use interrupts for handling incoming messages or that use shared memory in SMP clusters to allow more efficient sharing among processors on the same node.

The flag technique also applies to floating-point loads. The straightforward option is to do a floating-point compare on the loaded floating-point register. However, the latency of floating-point compares and branches is much greater than the corresponding integer operation (4-6 cycles vs. 1 cycle) in current Alpha processors. Therefore, we instead insert an additional integer load to the same target address as the floating-point load, and apply the flag technique to the integer value, as shown in Figure 5(b) (line 2 is the original floating-point load). Even with the additional load, the flag technique is still much cheaper than a state table check. Other processors and newer versions of the Alpha processor also support byte- and word-level loads. The flag technique can be used in these cases in the same way as for the floating-point loads. We simply add an extra instruction that loads in the longword which is then used for the flag compare.

Instruction scheduling techniques apply to the code in Figure 5(a) (and (b)). In the current Shasta compiler, we attempt to move the entire check (from line 2 on in Figure 5(a)) down in the application code so as to avoid a pipeline stall on the load when the add instruction is executed at line 2 (the processor stalls on use). This optimization is important since the Alpha compiler is typically successful in scheduling the original application code to avoid stalls on loads in the case of a first-level cache hit.

3.3 Improving Store Checks

One potential source of high overhead for the store miss checks are hardware cache misses incurred when loading the state table entry. Applications with good spatial locality will likely not cause many such hardware cache misses, since each byte-size state table entry maps the state of a whole Shasta line. For example, given a 64-byte

```
1.  ldl     r1, offset(base)
 or ldq     r1, offset(base)
2.  addl    r1, 253, rx
3.  bne     rx, nomiss

4. ... call function to handle load miss

5. nomiss:
6. ... continue with application ..
```

(a)

```
1.  ldl     rx, offset(base)
2.  ldt     $f0, offset(base)
3.  addl    rx, 253, rx
4.  bne     rx, nomiss

5. ... call function to handle load miss

6. nomiss:
7. ... continue with application ...
```

(b)

Figure 5: Optimized load miss check code for (a) integer loads and (b) floating-point loads.

Shasta line, the memory for the state table is $1/64$ of the memory of the corresponding lines. However, applications that incur a lot of cache misses due to poor spatial locality of the application data will likely incur a correspondingly large number of cache misses on the state table. To reduce the caches misses caused by the store miss check code, Shasta maintains an additional table, called the *exclusive table*, which has one-bit entries indicating whether the corresponding line is in exclusive state or not. The store miss check code can then check the exclusive table instead of the state table to determine if the store succeeds. For a 64-byte Shasta line, the memory for the exclusive table is $1/512$ of the memory of the corresponding lines. Therefore, the number of cache misses caused by store miss checks using the exclusive table can be as little as $1/8$th of the cache misses that would be caused by store miss checks that use the state table. Note that with the use of the flag technique and the exclusive table, accesses to the state table are completely eliminated from the inline check code for both loads and stores.

3.4 Batching Miss Checks

A very 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 we verify that the two lines that the loads and stores touch are in the correct state, then all of the loads and stores can proceed without further checks. We call this technique *batching* and refer to the set of instructions that contain the loads and stores as the *batched code*. Batching is also useful for eliminating and hiding communication latency in a parallel execution, since it allows load and store misses to the same line to be combined into a single store miss and misses on multiple lines to be serviced at the same time.

More generally, if we have a sequence of loads and stores whose offsets span a range of at most n lines, then all the loads and stores can proceed if we verify that all n or $n + 1$ lines covered by the range are in the correct state. For simplicity, we always choose n to be 2 in our implementation. This restriction does not have much of an impact, since most of the loads and stores relative to the same base register that can be batched have a small range of offsets.

One constraint on the batching technique is that the state of the lines accessed by the batched loads and stores cannot be changed within the batched code after the checks are done. We enforce this restriction by not polling for messages in the batched code, and by requiring that *all* loads and stores in the batched code (that are potentially to shared data) are checked in the initial miss checks for the batch. It is straightforward to do checks on multiple base register ranges, if necessary, to cover all the loads and stores in the batched code. Another more restrictive constraint is that a base register cannot be modified during the batched code, since then we may not have checked the correct lines during the initial batch miss check. Our current system relaxes this restriction slightly by allowing changes to a base register by a constant, since we can still statically determine the range of the loads and stores.

The batching technique is always useful in reducing miss checking overhead, but we find that it is especially useful for code where the compiler has done loop unrolling. In this case, the resulting loop body typically has many loads and stores with very little overhead for managing the loop. Fortunately, the loads and stores are usually all in a small range of offsets relative to a few base registers that are not modified during the loop body. Therefore, the batching technique nearly always applies and is very effective.

3.4.1 Choosing Loads and Stores to Batch

The Shasta compiler allows batching of loads and stores across basic blocks, but within a single procedure. There are typically many ways to batch loads and stores in large procedures with multiple basic blocks The Shasta compiler currently uses a simple greedy algorithm. Instructions are scanned in execution order while attempting to include as many loads and stores as possible in a batch. When a conditional branch instruction that is not a backedge for a loop is encountered, the algorithm continues its scan on each of the two possible paths. The scanning of two paths is merged again if the execution of the two paths merge. If scanning along one of the merging execution paths has already been terminated, then scanning along the other path is also stopped. Similarly, scanning along an execution path is terminated if it merges with an execution path that is not part of the scan. The scanning along a given path is terminated whenever an instruction is encountered that satisfies one of the following conditions:

- a load or store instruction that uses a base register that has been modified by a non-constant since the beginning of the batch, or
- a load or store instruction that makes the load/store range of a base register more than the Shasta line size, or
- a procedure call instruction, a branch that causes a loop, or a procedure return instruction, or
- a store instruction which is in one of the execution paths but not in a parallel path.

The reason for the last condition is a requirement of our cache coherence protocol that the batch miss handling code know exactly

```
1.  ldl     rx, 0(r1)
2.  ldl     ry, 44(r1)
3.  addl    rx, 253, rx
4.  addl    ry, 253, ry
5.  beq     rx, miss
6.  bne     ry, nomiss

7. miss:
8. ... load arguments for batch ranges and
   call function to handle batch miss

9. nomiss:
10 ... continue with the batched code ...
```

Figure 6: Optimized miss check code for batched loads.

which stores will be will be executed during the batched code. The algorithm terminates when scanning along all execution paths has been terminated.

The compiler then generates the proper checking code for all base registers which have loads or stores in the batch. The normal shared miss checks are used if there is only a single load or store for each base register, since batching can actually increase overhead in this case. The scanning process starts again with the earliest instruction in the current procedure that has not yet been scanned.

3.4.2 Code for Batch Miss Checks

As described in the previous section, our current implementation restricts batching to loads and stores via a particular base register that span at most two cache lines. One convenient way to check both lines is to do a normal shared miss check on the beginning address and ending address of the range. Fortunately, the checking code for the two endpoints can be interleaved effectively to eliminate pipeline stalls; therefore, the cycle count is less than double the cycle count of normal checks (see Table 1). For example, Figure 6 shows sample code for checking a range of quadword loads via base register `r1` with offsets 0 to 40 (`rx` and `ry` are free registers). We generate store check code for a range if any of the memory operations in that range are stores. When there are multiple base registers in the batch, the checks for the registers can also be scheduled in an interleaved manner to further reduce the total cycle count. The check is simplified in the case of loads and stores via a particular base register that are to the same location, since only a single cache line needs to be checked. The batch miss code is executed if there is a miss on any line of any base register range. There is possible aliasing between loads and stores via different base registers; however, the batch checking code correctly detects a miss even if different base register ranges have overlapping lines. We discuss the inline batch miss code and the handling of a batch miss in Section 4.3.

4 Protocol Optimizations

This section describes a number of the optimizations in the Shasta coherence protocol that attempt to reduce the effect of the long latencies and large message overheads that are typical in software DSM systems. The optimizations include exploiting relaxed memory models, supporting coherence and communication at multiple granularities, and batching together requests for multiple misses.

4.1 Exploiting Relaxed Memory Models

Our protocol aggressively exploits the release consistency model [8] by emulating the behavior of a processor with non-blocking loads and stores and a lockup-free cache. Because of our non-blocking load and store operations, a line may be in one of two pending states, *pending-invalid* and *pending-shared*. The pending-invalid state corresponds to an outstanding read or read-exclusive request on that line; pending-shared signifies an outstanding exclusive request. The protocol supports non-blocking stores by simply issuing a read-exclusive or exclusive request, recording where the store occurred, and continuing. This information allows the protocol to appropriately merge the reply data with the newly written data that is already in memory. Our protocol also exhibits non-blocking load behavior due to the batching optimization, since batching can lead to multiple outstanding loads (as described in Section 4.3).

We also support 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 recording the location of the stores in the miss handler (invoked due to the pending state). Loads from a line in pending-shared state are allowed to proceed immediately, since the node already has a copy of the data. Loads from a line in pending-invalid state are also allowed to proceed as long as the load is from a valid section of the line. The above two cases are well-suited to the "flag" check for loads since this technique can efficiently detect a "hit" in both cases without actually checking the state for the line. Finally, in the case of a pending read-exclusive request, we allow the requesting processor to use the reply data as soon as it arrives (by immediately setting the local state to exclusive), even though requests from other processors are delayed until all pending invalidations are acknowledged.

4.2 Multiple Coherence Granularity

The most novel aspect of our protocol is its ability to support multiple granularities for communication and coherence, even within a single application. The fact that the basic granularity for access control is software configurable already gives us the ability to use different granularities for different applications. Nevertheless, the ability to further vary the communication and coherence granularity within a single application can provide a significant performance boost in a software DSM system, since data with good spatial locality can be communicated at a coarse grain to amortize large communication overheads, while data prone to false sharing can use a finer sharing granularity.

Our current implementation automatically chooses a block size based on the allocated size of a data structure. Our basic heuristic is to choose a block size equal to the object size up to a certain threshold; the block size for objects larger than a given threshold is simply set to the base Shasta line size. The rationale for the heuristic is that small objects should be transferred as a single unit, while larger objects (e.g., large arrays) should be communicated at a fine granularity to avoid false sharing. We also allow the programmer to override this heuristic by providing a special version of `malloc` that takes a block size as an argument. Since the choice of the block size does not affect the correctness of the program, the programmer can freely experiment with various block sizes (for the key data structures) to tune the performance of an application. Controlling the coherence granularity in this manner is simpler than approaches adopted by object- or region-based DSM systems [2, 11, 14, 16], since the latter approaches can affect correctness and typically require a more substantial change to the application.

We currently associate different granularities to different virtual

pages and place newly allocated data on the appropriate page. The block size for each page is communicated to all the nodes at the time the pool of shared pages are allocated. To determine the block size of data at a particular address, a requesting processor simply checks the block size for the associated page. The above mechanism is simple yet effective for most applications.

We are also working on a more general mechanism that maintains the block size information permanently only at the home along with the directory information for each line. Other processors maintain the block size for a piece of data only while they are caching parts of the data. When a request for a particular address is served by the home node, the home obtains the block size information for the data as part of the directory lookup. The request is then processed for all the lines in the associated block. For example, for a read request to a block with four lines, the home sends back the appropriate four lines of data to the requesting processor. The block size of the data is transmitted with all protocol messages except the initial request, so all protocol operations affect the entire block. This method allows different block sizes for data on the same page. In addition, it is possible to dynamically change the granularity of a data structure by invalidating all copies of the data (except at the home) and then changing the block size information at the home.

The mechanism described above introduces some complexities in the protocol, because a node is not aware of the block size for a piece of data when it is requesting the data. For example, if a node issues several (non-stalling) stores to different lines of the same invalid block, it may issue duplicate read-exclusive requests because it does not know that the stores are to the same block. Such duplicate requests can cause incorrect protocol behavior if they are not detected. One solution is for the requester to communicate its set of outstanding misses to "nearby" lines (which will often be empty) with every request; the home can then detect and drop all duplicate requests when any of the outstanding misses fall within the same block boundary. Another approach depends on the fact that the home usually knows from the directory information that it has already serviced a request from a given node, and can therefore detect and ignore duplicate requests in most cases; however, this latter approach requires the protocol to robustly deal with some duplicate requests, since intervening accesses by other nodes can alter the directory information and prevent the home from filtering all duplicates.

4.3 Batching

The batching technique of Section 3.4 can reduce communication overhead by merging load and store misses to the same line and by issuing requests for multiple lines at the same time. Our protocol handles a miss associated with the batching of loads and stores as follows. The batch checking code jumps to the inline batch miss code if there is a miss on any line of any base register range of the batch. The inline code repeatedly calls a routine to record the starting address and length of each base register range; for ranges with stores, it also includes a bitmap indicating where the stores will occur. The inline code then calls a batch miss handler that issues all the necessary miss requests. We implement non-stalling stores by requiring the handler to wait only for outstanding read and read-exclusive replies and not for invalidation acknowledgements. The protocol correctly handles the case where base register ranges overlap, by issuing only one request for a line in more than one range and recognizing that the reply applies to both ranges.

Although the batch miss handler brings in all the necessary lines, it cannot guarantee that all the lines will be in the appropriate state once all the replies have come back. The reason is that while the handler is waiting for the replies, requests from other processes must be served to avoid deadlock; these requests can in turn change the state of the lines within the batch. Even though a line may not be in the right state, loads to the line will still get the correct value (under release consistency) as long as the original contents of the line remain in memory. We therefore delay storing the flag value into memory for invalidated lines until after the end of the batch. After the batch code has been executed, we complete the invalidation of any such lines. We may also have to reissue stores to lines which were not in the exclusive or pending-shared state when we started the batch code. We do these invalidations and reissues at the time of the next entry into the protocol code (due to polls, misses, or synchronization).

The batching optimizations can be done even under sequential consistency, as long as the handler waits for all requests (including exclusive requests) to complete and ensures that all lines are in the correct state. The case when all lines are not in the right state is significantly more complex to handle under sequential consistency. In addition, performance is lower because the handler must wait for all invalidation acknowledgements before proceeding with the batched code.

5 Performance Results

This section presents performance results for our current Shasta implementation. We primarily focus on characterizing the overhead of miss checks by presenting the static overheads for individual miss checks, the dynamic overheads for all of the SPLASH-2 applications [22], and the frequency of instrumented accesses in these applications. In addition, we present preliminary parallel performance results for some of the SPLASH-2 applications running on a cluster of Alpha workstations connected by Digital's Memory Channel network [9].

5.1 Instruction and Cycle Overhead of Miss Checks

Table 1 gives the static overheads for our optimized load and store miss checks, when using the flag technique and exclusive table, in the case that there is no shared data miss. We have shown instruction counts and cycle counts for the Alpha 21064A and 21164 processors. The cycle overheads may be less than shown if our instruction scheduling techniques are successful in placing the extra instructions at points where application instructions stall. The cycle counts on the Alpha 21164 are lower because of fewer pipeline stalls and dual-issue of some of the checking code. The store overheads vary depending on whether the offsets of the stores are zero or not. The overhead for non-batched store checks would be lower (3-4 instructions) if we eliminated the range check and accessed word-sized state table entries. Clearly, because of the use of the flag for loads, store checks are quite a bit more expensive than load checks. The batch load and store overheads are for a single base register range; they are higher than for normal load and store checks because they must do checks on both ends of the range. For comparison, we also show the cycle latency for loads (assuming first-level cache hits), integer operations, and floating point operations on the 21064A and 21164.

The inline miss check code may also have some dynamic overhead. The load of the exclusive table entry during a store miss check may cause extra hardware cache misses, TLB misses, or even page faults. Similarly, because of the additional inline code, the modified executable may have more instruction cache misses or instruction TLB faults than would have occurred in the unmodified application.

	Miss Checks					Processor Operations		
	No Batching			Batching				
	Integer Load	FP Load	Integer/FP Store	Batch Load	Batch Store	Integer/FP Load	Integer Op.	FP Op.
Instructions	2	3	6-7	6	10-12	1	1	1
Cycles (21064A)	4	4	10-11	7	11-13	3	1-2	6
Cycles (21164)	2-3	2	6	4	7-8	2	1	4

Table 1: Instruction and cycle counts for miss checks.

Finally, the branches in the miss check code may cause additional delays if they are mispredicted, but the processor's branch prediction should be fairly effective in the common case when there is no shared data miss.

5.2 Miss Check Overhead for SPLASH-2 Applications

For our measurements, we compile the SPLASH-2 codes with full optimization and with their synchronization macros converted to calls that Shasta can recognize.[1] The Shasta compiler then automatically transforms the application executable into a new executable that includes all the proper load and store checks and can run on a cluster of workstations. This section analyzes the overhead due to the checking code on a single processor (hence, with no shared misses). We report on the parallel performance of some of these applications in Section 5.4.

Table 2 reports the factor by which run time increases with respect to the original sequential executable when the miss checks (and polling) are added. The measured run time is the wall-clock time for each application to complete after the initialization period defined in each SPLASH-2 application. We show the changes in the overheads as various techniques for reducing the overhead are added. Our results are for a 275 MHz 21064A dual-issue processor, which has a 16 Kbyte on-chip instruction cache, a 16 Kbyte on-chip data cache, and a 4 Mbyte second-level off-chip data cache. We use the standard SPLASH-2 problem sizes for each of the applications [22], except for Radiosity and Raytrace, where we use smaller test sets so as to avoid running out of swap space on some of our systems. The line size of the Shasta system is configured as 64 bytes.

The fourth column in Table 2 reports the overhead for the basic inline checks shown in Figure 2 and only instrumenting loads and stores that are not known to be to the stack or static data. Obviously, the basic overhead is quite high, even though the inline code has already been well optimized by laying out memory properly and making use of free registers. Barnes and Volrend have lower overheads than the other applications, because much of the frequently executed code uses data that is temporarily stored in the stack or static area. The next column gives the results when scheduling techniques are used to reduce the overhead for store checks. The sixth column gives the overheads when the flag technique is used for loads. The flag technique is highly effective in reducing the overhead of load miss checks. The seventh column gives the overheads when the exclusive table is used for store checks. The overhead in Radix goes down significantly, because Radix has very poor spatial locality and therefore causes a lot of cache misses on state table lookups as well. The eighth (bold) column gives the overheads when batching is used. Batching across basic blocks is particularly effective in Raytrace, because it has many conditionals in its most frequently executed functions; batching within basic blocks only reduces the overhead to 1.39.

The overall decrease in overhead from the fourth column to the eighth column is quite dramatic, ranging from a three- to six-fold decrease for all applications. As shown by the final overhead numbers in the eighth column, the techniques we have described are successful in reducing the overhead of the Shasta system to a reasonable level. It is important to note that the effect on parallel performance will not be as large as the effect shown for a single processor. In a parallel execution, each processor's time is spent doing useful work plus doing things related to the parallel execution, such as sending messages, waiting for replies, or waiting at synchronization points. The overheads measured in Table 2 only increase the time of the "useful" work component. The effective overhead in a parallel run is therefore roughly equal to the overhead effect on one processor multiplied by the parallel efficiency of the application run (the average percentage of each processor's time spent doing useful work). For example, consider an application run that achieves a parallel efficiency of 50% without any checking overhead. Given a checking overhead of 20% on a uniprocessor run, the effective overhead on the parallel execution time is reduced to 10%. The overhead may also be reduced because each processor in a parallel execution touches only a part of the whole data set, so the number of data cache misses on the state table (or exclusive table) is smaller.

In the above results, polling is done only when the local process has a miss. The ninth and tenth columns indicate the increase in overhead when polling is also done either at the entry to each function or at each loop backedge, respectively. Loop polling is usually more expensive than polling at each function entry, but the use of loop polling should cause an application to poll more regularly and potentially reduce the average latency for servicing messages. Polls involve three instructions: setup of the polling address, load of the poll location, and a conditional branch. For loop polling, the Shasta compiler can often reduce the polling overhead from three to two instructions by hoisting the address setup outside of the loop. The Shasta compiler also attempts to schedule the polling instructions so as to avoid pipeline stalls on the load of the polling location. In addition, no polling is inserted for "small" loops (loops that have no function calls and do not execute more than 15 instructions per iteration). Except for Raytrace, the overhead of polling by either method is 5% or less.

The last column shows the overhead when range checks are not used (but all other optimizations as well as loop polling are used). The overhead increases for Barnes, Radiosity, and Water because there are many loads and stores in these applications which are instrumented, but are actually to non-shared data. The quick range check reduces the dynamic overhead for the store check in this case. In all the other applications, the removal of the range checks tends to reduce the overhead slightly. We have chosen to use the range check in the basic Shasta system, since it can significantly reduce

[1] We currently make a one-line change to the CREATE macro that makes it simpler for Shasta to initialize static data properly when a new process is spawned.

	Problem Size	Seq. Time (secs)	Inline Checks (Sec. 2.4)	Split Store Checks (Sec. 3.1)	Flag Load Checks (Sec. 3.2)	Exclusive Store Checks (Sec. 3.3)	Batch Checks (Sec. 3.4)	Func. Polling	Loop Polling	No Range Checks
Barnes	16K part.	13.78s	1.26	1.26	1.14	1.14	**1.08**	1.10	1.08	1.12
Cholesky	tk15.O	2.173s	2.66	2.56	1.80	1.78	**1.41**	1.41	1.41	1.41
FFT	64K points	0.195s	2.21	2.12	1.71	1.66	**1.29**	1.27	1.34	1.33
FMM	16K part.	9.635s	1.40	1.39	1.17	1.14	**1.11**	1.13	1.15	1.15
LU	512x512	3.607s	2.34	2.17	1.67	1.65	**1.25**	1.26	1.27	1.27
LU-Contig	512x512	3.041s	2.65	2.45	1.70	1.68	**1.29**	1.31	1.32	1.33
Ocean	258x258	4.245s	1.76	1.71	1.25	1.26	**1.12**	1.12	1.14	1.13
Radiosity	test	5.623s	1.73	1.70	1.39	1.30	**1.19**	1.20	1.20	1.26
Radix	1M ints.	0.815s	1.95	1.95	2.02	1.36	**1.33**	1.33	1.37	1.36
Raytrace	teapot	2.954s	2.26	2.25	1.63	1.62	**1.29**	1.38	1.36	1.37
Volrend	head	2.764s	1.19	1.19	1.05	1.05	**1.05**	1.05	1.07	1.08
Water-Nsq	512 molec.	3.292s	1.65	1.57	1.21	1.22	**1.14**	1.16	1.19	1.23
Water-Sp	512 molec.	3.934s	1.54	1.48	1.23	1.23	**1.18**	1.18	1.19	1.22

Table 2: Factor increase in execution time with miss checks over sequential time (on a 275 MHz 21064A Alpha).

the overhead for some applications, while not greatly increasing the overhead for the rest.

We have also measured the Shasta overheads on a number of other Alpha processors. Relative overheads are larger for some applications and smaller for others in comparison to the overheads in Table 2, but overall the results are similar in each case. For example, for a 350 MHz 21164 quad-issue processor (which has 8KByte first-level data and instruction caches, a 96Kbyte combined second-level cache, and a 4 Mbyte third-level off-chip data cache), overheads corresponding to the seventh column range from 1.04 to 1.36, except for radix, whose overhead is 1.54.

5.3 Frequency of Instrumented Accesses

Table 3 shows the code size increase due to instrumentation and the percentage of loads and stores that are checked. The first column shows the increase in the number of instructions when the sequential executable is instrumented, including all optimizations above and loop polling. These numbers could be reduced by optimizing the inline code for calling the miss routines, especially the batch miss routines. The second and third columns are static measures which give the percentages of all loads and stores in the executable (including in all standard library routines) which must be instrumented because they might reference shared data. The next two columns are dynamic figures which give the percentage of loads and stores that are checked during the execution of the application on one processor, when batching is *not* used. In parentheses, we give the percentages when loads and stores that are known to be relative to the GP *are* instrumented. These figures indicate the expected increase in overhead if Shasta used a memory model in which static data was shared as well as dynamically allocated data. The final two columns give the same percentages when batching is used. For these figures, we count the check for each base register range in a batch as a single store or load check, depending on whether the range has any stores or not.

We see that, for many applications, most loads and store operations are checked, because the most frequently executed code mainly accesses shared data. Batching is highly effective in reducing the number of checks in most cases. The applications that have the highest overall overhead tend to be the ones in which a high percentage of the stores are checked, because of the asymmetry in the overheads of the load and store checks. We also see that the dynamic percentage of loads and stores that are checked does not increase much when loads and stores to the static data area are checked. The biggest increase in the percentages occurs for Barnes; however, the actual run-time overhead for Barnes (when range checks are not used) only increases from 1.12 to 1.29. FMM, Radiosity, and Raytrace have significant but smaller run-time increases, while there is almost no increase for the rest of the other applications.

5.4 Preliminary Parallel Performance Results

This section presents results of parallel execution of several SPLASH-2 applications using Shasta on two different platforms. The two platforms differ substantially in both processor and message passing performance. The first platform is a collection of eight 190 MHz 21064 Alpha processors connected by a Memory Channel network. This configuration consists of four AlphaServer 2100 servers each with 2 processors; however, all communication is via Memory Channel, even if processors are located on the same server. The minimum round-trip latency in Shasta for a 2-hop read miss on a 64-byte line is 37 microseconds, with a 21 MB/s incremental bandwidth for larger lines. These numbers degrade to 42 microseconds and 15 MB/s for processors on the same server since they share the same link to the network. The individual processors have ratings of 132 SPECint92 and 161 SPECfp92. Our second platform is an AlphaServer 8400 with eight 300 MHz 21164 Alpha processors on a shared bus. The Shasta implementation on this system is identical to the Memory Channel implementation, except that processes communicate via a message-passing layer implemented on top of the shared memory. This allows us to study the effects of an improved message passing medium. The minimum round-trip latency in Shasta to fetch a 64-byte line is 14 microseconds, with around 40 MB/s incremental bandwidth for larger lines. Finally, the individual processors have ratings of 341 SPECint92 and 513 SPECfp92.

Figure 7 shows speedup curves for both the Memory Channel and the AlphaServer 8400 platforms. The speedups shown are the ratio of the execution time of the application running via Shasta on 1, 2, 4, and 8 processors to the execution time of the original sequential application (with no miss checks). For these results,

	code size increase	static checks		dynamic checks			
				no batching		batching	
		loads	stores	loads	stores	loads	stores
Barnes	74%	39%	34%	33% (64%)	15% (35%)	10% (31%)	6% (16%)
Cholesky	83%	41%	37%	89% (89%)	84% (84%)	24% (24%)	34% (34%)
FFT	64%	33%	33%	93% (96%)	94% (94%)	7% (10%)	26% (26%)
FMM	64%	33%	29%	32% (53%)	20% (20%)	15% (36%)	13% (14%)
LU	68%	32%	31%	88% (90%)	98% (98%)	13% (16%)	25% (25%)
LU-Contig	67%	31%	31%	85% (88%)	98% (98%)	13% (16%)	25% (25%)
Ocean	107%	51%	52%	85% (96%)	98% (98%)	25% (32%)	44% (44%)
Radiosity	76%	40%	37%	57% (69%)	19% (19%)	22% (33%)	11% (11%)
Radix	71%	31%	33%	100% (100%)	100% (100%)	60% (60%)	100% (100%)
Raytrace	70%	40%	35%	73% (81%)	30% (30%)	39% (46%)	20% (20%)
Volrend	78%	36%	32%	14% (38%)	0.1% (0.1%)	14% (36%)	0.1% (0.1%)
Water-Nsq	66%	36%	33%	56% (69%)	53% (53%)	31% (42%)	17% (17%)
Water-Sp	66%	36%	32%	51% (66%)	46% (46%)	27% (38%)	12% (12%)

Table 3: Code size increase and percentage of accesses that are checked (in parentheses, including accesses relative to the GP).

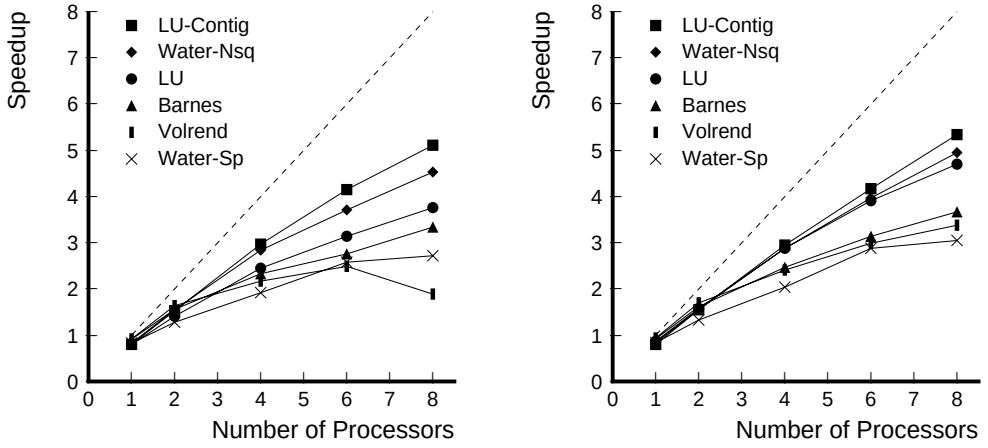

Figure 7: Shasta speedups for SPLASH-2 applications on Memory Channel cluster (left) and AlphaServer 8400 (right).

loop polling is used, and the fixed Shasta line size is 64 bytes. The block size of objects less than 1024 bytes is automatically set to the size of the object. We have additionally made one-line changes (at `malloc` calls) to the applications (except for Water-sp) to explicitly set the block size for one or two important data structures. Table 4 describes the selected data structures along with the block size specified for them. The table also gives the problem sizes (larger than default SPLASH-2 sizes for several applications) and the sequential execution times. Even though the two platforms are significantly different in their underlying processor and messaging performance, the speedups for various applications are roughly comparable. Theoretically, the faster processors on the 8400 platform make it more difficult to get good speedups. However, the higher performance messaging layer and the fact that protocol processing is also sped up by the faster processors seems to compensate for this, leading to better speedups on this platform. The reason for the drop off in performance of Volrend on the Memory Channel platform is the lower communication bandwidth. Table 4 also compares the eight-processor speedups on the AlphaServer 8400 platform when the default block size of 64 bytes is used for the selected data structures and when the specified block size is used. In the following, we discuss the applications and explain the choice of block sizes.

Barnes has three main arrays which hold three different kinds of structures called cells, leaves, and bodies. The cells and leaves are part of an oct-tree that is repeatedly traversed by all the processors without being modified in the main force computation phase. Therefore, cell and leaf arrays can be communicated at a larger granularity, such as 512 bytes, so that contents of the tree are fetched with few messages. On the other hand, the body array can be subject to false sharing during the force computation phase, since the bodies are modified by the individual processors that "own" them. It is therefore better to keep the block size of the body array at the default 64 bytes. The interesting result is that the best performance is obtained when different arrays in the application use different coherence granularities.

In non-contiguous LU, the matrix is allocated as one big array, so the blocks of the matrix that are modified by difference processors are not contiguous. Given matrix blocks of 16x16 floating point numbers, one row of a block is 128 bytes. The largest block size for the matrix array that does not cause false sharing is therefore 128 bytes. In contiguous LU, the data for each matrix block is actually stored contiguously (but still in one big array). Therefore, it makes sense to maintain coherence at the granularity of a matrix block, which is 2048 bytes.

In Water-nsquared, all of the molecules are stored in a single array. Since each processor accesses all molecules at each time step and molecule information is only changed at the end of a time step, it makes sense to have a large coherence granularity for the molecule

	problem size	sequential time		selected data structure(s)	specified block size (bytes)	8-proc. speedup (on 8400)	
		190MHz 21064	300MHz 21164			default block size (64 bytes)	specified block size
Barnes	16K particles	22.55s	9.04s	cell, leaf arrays	512	3.2	3.7
LU	1024x1024 matrix	48.55s	27.45s	matrix array	128	3.8	4.7
LU-Contig	1024x1024 matrix	39.63s	17.75	matrix block	2048	3.1	5.3
Volrend	head	4.22s	1.86s	opacity, normal maps	1024	2.8	3.4
Water-Nsq	1000 molecules	17.20s	7.88s	molecule array	2048	4.4	5.0
Water-Sp	1000 molecules	8.55s	3.91s	molecules, boxes	varies	NA	3.1

Table 4: Effects of varying block sizes on speedups on AlphaServer 8400.

array. We obtain the best performance when the block size for this array is 2048 bytes. Similarly, in Volrend, the opacity and normal tables are read by all processors, and it improves performance to increase the block size of these tables to 1024 bytes. In Water-spatial, all of the molecule and box data structures are allocated separately and are under 1024 bytes. Therefore, our heuristic for choosing block sizes automatically makes the coherence granularity for the molecules and boxes equal to the size of the molecule and box structure, respectively.

In summary, the speedups we have observed are encouraging given the extremely fast processors and the large message latencies. In addition, the ability to choose different coherence granularities for different data structures in the same application appears to be highly beneficial for boosting performance.

6 Related Work

Shasta's basic approach is derived from the Blizzard-S work [18]. However, we have substantially extended the previous work in this area by developing several techniques for reducing the otherwise excessive access control overheads. We have also developed an efficient protocol that provides support for maintaining coherence at variable granularities within a single application. Finally, we have explored the use of relaxed memory models in the context of software protocols that can support coherence at a fine granularity.

We independently developed the "flag" technique of Section 3.2 for use with all types of loads in the 64-bit Alpha architecture. We recently became aware that a form of the "flag" technique has been proposed for use in hardware in the StarT-NG machine [5]. The Blizzard-S project has also recently incorporated the flag technique by adapting this idea from the StarT-NG design [17]. With this optimization, the Blizzard-S overhead is 3 instructions at most loads and 8 instructions at most stores. Run-time overheads on a 66 MHz HyperSPARC processor (with a 8K first-level data cache and 256K second-level data cache) are reported for five applications, of which only one is a SPLASH-2 application [17]. For the Barnes-Hut application, the reported Blizzard-S overhead on the Sparc is 1.6, while the Shasta overhead on the 275 MHz Alpha is 1.08. We have also measured the Shasta overhead for the appbt application. The Blizzard-S overhead is 1.9, while the Shasta overhead is 1.19.

There are several other systems that use compiler-generated checks to aid in implementing a global address space. Olden [3] uses the compiler to insert checks at loads and stores to implement a specialized shared memory protocol across workstations. Split-C [6] uses compiler-inserted checks to implement a shared address space without caching in the context of a parallel language. Midway [2] inserts code at stores to record where writes have occurred in a shared memory block. Some systems that utilize garbage collection record the location of stores in a manner similar to Midway to aid in the process of scavenging for free memory [21].

Object- or region-based DSM systems [1, 2, 11, 14, 16] communicate data at the object level and therefore support coherence at multiple granularities, but these systems require explicit programmer intervention to partition the application data into objects and to identify when objects are accessed through annotations. Midway also allows different regions of memory to have different granularities for detecting writes. Even though a finer granularity of write detection can reduce the amount of communicated data, the access and coherence granularity is still at an object or page level (depending on the consistency model). Similarly, some page-based systems (e.g., Treadmarks [12]) reduce the required bandwidth by only communicating the differences between copies, but the coherence granularity is still a page. Page-based DSM systems implemented on a cluster of shared-memory multiprocessors, such as MGS [23] and SoftFLASH [7], naturally support two coherence granularities – the line size of the multiprocessor hardware and the size of the virtual memory page. However, neither of these granularities can be changed.

A number of systems attempt to use a small amount of extra hardware to support fine-grain access control to shared data more efficiently. Blizzard-E [18] uses ECC bits at the memory level to cause faults on accesses to particular lines; similarly, Typhoon-0 [15] uses hardware at the memory bus to detect an access fault. These schemes require precise memory exceptions on processor reads and writes if the main processor is to execute the code to handle the access faults; however, precise exceptions on processor writes are not supported in many processors (e.g., due to the presence of write buffers). Support for "informing" memory operations has also been proposed as a way of invoking handler code whenever there is a miss in the primary data cache [10]. While "informing" operations may be used to implement coherent shared memory, a major disadvantage of the scheme is that the handler is called whenever referenced data is not in the first-level cache, even if it is in local memory. Another potential mechanism is to extend the functionality of the TLB to support fine-grain access control bits. For example, the RS/6000 provides a single access control bit for every 128-byte segment. This approach has the advantage of leveraging the precise exception and fast trap mechanisms that are already present for handling TLB faults. It is interesting to note that some of the techniques we have developed to reduce access control overheads in software may also be useful in systems with hardware assistance. For example, our flag technique for loads can be used in a hybrid system where access checks for loads are achieved efficiently in software, while the TLB control bits can be used to invoke a trap on a store access fault. Among the above techniques, an *"extended TLB"* with two access control bits (e.g., per 128 bytes) to allow for efficient checks for both loads and stores seems to be the most promising approach if the goal is to handle access control faults on the main processor.

7 Conclusion

Instrumenting loads and stores in an application to check for accesses to remote data is a potentially attractive technique for supporting a fine-grain shared address space in software. We have demonstrated a set of techniques that reduce both the cost and the frequency of the added code in order to decrease the overall overhead to acceptable levels. Our techniques to reduce the overhead of the inline checks include careful memory layout of the program, using effective instruction scheduling between application instructions and checking instructions, reducing cache misses incurred by the instrumentation code, and a novel method for quickly determining load misses. Our techniques to reduce the frequency of instrumentation include using compiler analysis to eliminate unnecessary loads and store checks and batching checks for multiple loads and stores.

Our current implementation of Shasta on Alpha processors incurs instrumentation overheads in the range of 5–35% for almost all of the SPLASH-2 applications. This overhead may be reduced further by interprocedural analysis and better instruction scheduling. We feel that this level of overhead makes it feasible to use our approach to support fine-grain data sharing on clusters of processors connected by high-speed interconnects, especially because the relative effect of the checking overhead on the parallel execution time is reduced due to the addition of other overheads such as communication latency. Since Shasta supports shared memory entirely in software, it provides considerable flexibility in managing coherence granularity and applying protocol optimizations. Our parallel performance results on a cluster of Alpha workstations illustrate the benefits of this flexibility.

Acknowledgments

We would like to thank Amitabh Srivastava for advice in modifying ATOM, John Brosnan, Tim Reddin, and Marc Viredaz for help in using the Memory Channel, and Roland Belanger, Rick Gillett, and David Pimm for providing access to machines. Thanks to Jennifer Anderson, Luiz Barroso, Carl Waldspurger, and the anonymous referees for comments on earlier drafts of the paper.

References

[1] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, Mar. 1992.

[2] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. In *COMPCON 1993*, pages 528–537, Mar. 1993.

[3] M. C. Carlisle and A. Rogers. Software Caching and Computation Migration in Olden. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 29–38, July 1995.

[4] 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.

[5] D. Chiou, B. S. Ang, Arvind, M. J. Becherle, A. Boughton, R. Greiner, J. E. Hicks, and J. C. Hoe. StarT-NG: Delivering Seamless Parallel Computing. In *Proceedings of EURO-PAR '95*, pages 101–116, Aug. 1995.

[6] D. E. Culler et al. Parallel Programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, Nov. 1993.

[7] 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*, Oct. 1996.

[8] 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.

[9] R. Gillett, M. Collins, and D. Pimm. Overview of Memory Channel Network for PCI. In *Proceedings of COMPCON '96*, pages 244–248, Feb. 1996.

[10] M. Horowitz, M. Martonosi, T. C. Mowry, and M. D. Smith. Informing Memory Operations: Providing Memory Performance Feedback in Modern Processors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 260–270, May 1996.

[11] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. In *Proceedings of the Fifteenth Symposium on Operating System Principles*, pages 213–228, Dec. 1995.

[12] 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.

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

[14] R. S. Nikhil. Cid: a Parallel, "Shared-memory" C for Distributed-memory Machines. In *Seventh Workshop on Languages and Compilers for Parallel Computing*, pages 376–390, Aug. 1994.

[15] S. K. Reinhardt, R. W. Pfile, and D. A. Wood. Decoupled Hardware Support for Distributed Shared Memory. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 34–43, May 1996.

[16] D. J. Scales and M. S. Lam. The Design and Evaluation of a Shared Object System for Distributed Memory Machines. In *Proceedings of the First Symposium on Operating System Design and Implementation*, pages 101–114, Nov. 1994.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] P. R. Wilson and T. G. Moher. A Card-marking Scheme for Controlling Intergenerational References in Generation-Based GC on Stock Hardware. *SIGPLAN Notices*, 24(5):87–92, 1989.

[22] 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.

[23] 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 44–55, May 1996.