Software Write Detection for a Distributed Shared Memory

Matthew J. Zekauskas Wayne A. Sawdon
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
{mattz,wsawdon}@cs.cmu.edu

Brian N. Bershad
Department of Computer Science
and Engineering FR-35
University of Washington
Seattle, WA 98185
bershad@cs.washington.edu

Abstract

Most software-based distributed shared memory (DSM) systems rely on the operating system's virtual memory interface to detect writes to shared data. Strategies based on virtual memory page protection create two problems for a DSM system. First, writes can have high overhead since they are detected with a page fault. As a result, a page must be written many times to amortize the cost of that fault. Second, the size of a virtual memory page is too big to serve as a unit of coherency, inducing false sharing. Mechanisms to handle false sharing can increase runtime overhead and may cause data to be unnecessarily communicated between processors.

In this paper, we present a new method for write detection that solves these problems. Our method relies on the compiler and runtime system to detect writes to shared data without invoking the operating system. We measure and compare implementations of a distributed shared memory system using both strategies, virtual memory and compiler/runtime, running a range of applications on a small scale distributed memory multicomputer. We show that the new method has low average write latency and supports fine-grained sharing with low overhead. Further, we show that the dominant cost of write detection with either strategy is due to the mechanism used to handle fine-grain sharing.

1 Introduction

A distributed shared memory (DSM) system provides the abstraction of a shared memory multiprocessor to a program running on multiple processors that do not physically share memory. Software DSM systems implement this abstraction entirely in software, relying on an explicit message-passing network for communication [Li & Hudak 89, Carter et al. 91, Bershad et al. 93, Keleher et al. 94]. The primary service provided by these systems is *cache management*, which enables processors to cache recently accessed data items in local memory. Cache management requires that the DSM system detect writes to shared memory so that a consistency protocol can be activated.

In this paper, we examine the performance implications of two strategies – one relying on virtual memory page protection (VM-DSM) and one relying on the compiler and runtime (RT-DSM) – to detect and collect writes to shared data in a software-based DSM system. We show that RT-DSM has lower overhead than VM-DSM for three reasons. First, RT-DSM tends to have lower average update latency because it can avoid the operating system altogether. Second, RT-DSM directly supports variable sized objects eliminating *false-sharing* and the overhead necessary to accommodate it. Third, RT-DSM efficiently provides a detailed update history, which allows it to minimize the data transferred to maintain consistent memory.

1.1 Motivation

Most DSM systems use virtual memory page protection to detect writes to shared memory. The runtime relies on virtual memory write faults to detect updates to a program's address space. VM-DSM systems are attractive because they allow an application to use shared memory in a transparent fashion using native compilers and libraries [Li & Hudak 89]. There are, however, some disadvantages. Virtual memory operations, even in an optimized system, have a high cost [Appel & Li 91, Thekkath & Levy 94] relative to that of an instrumented write, as we show in this paper. More importantly, the large virtual memory page size creates problems with false sharing [Bennett et al. 90]. To support sharing at the sub-page level, an additional mechanism such as page "diffing" is required [Carter et al. 91]. Such mechanisms are generally memory reference intensive, and

This research was sponsored in part by the Advanced Research Projects Agency under the title "Software System Support for High Performance Multicomputing", contract number DABT63-93-C-0054. Sawdon was partially supported by a grant from the International Business Machines Corporation.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of ARPA, IBM, or the U.S. Government.

therefore expensive. Moreover, mechanisms that deal with false sharing at the page level do not efficiently reveal a fine-grained history of updates to shared data, resulting in either the transfer of excess data or a more complicated data consistency algorithm.

In contrast to VM-DSM, RT-DSM relies on the compiler and runtime to detect writes entirely in software at user-level. No virtual memory operations are required. Each memory store operation is instrumented by the compiler to update a locally maintained *dirtybit* associated with the stored address. This approach offers low overhead and supports fine-grain sharing since the unit of coherency is not tied to the virtual memory page size. Fine-grain dirtybits can be used directly to record the history of updates to a particular line of shared memory. This history enables the RT-DSM system to perform an update at a processor exactly once, thereby minimizing the data communicated between processors.

A potential disadvantage with the RT-DSM approach is that the overhead of write detection occurs on every write. Multiple stores to the same item incur the same overhead each time. In contrast, with VM-DSM, only the first store to a page must be detected with a page fault. Others can proceed at full speed, thereby amortizing the initial overhead. The amortization period lasts until another processor attempts to access the new data or its containing page. The page must then be write protected to detect the next store.

This paper explores the tradeoffs between the two approaches to detecting writes in a DSM system. We base our comparison on the observed behavior of applications running on top of Midway [Bershad et al. 93], a DSM system for a network of workstations that can be configured as either a VM-DSM or an RT-DSM for parallel C programs.

1.2 Related work

Most software DSM work has focused on improving the performance of page-based DSM systems [Minnich & Farber 89, Bryant et al. 91, Carter et al. 91, Keleher et al. 94]. A few parallel programming environments support shared memory, but do not rely on hardware page protection, such as Orca, Amber and Emerald. Orca [Bal et al. 92] implements an update-based protocol based on language-specific objects. The protocol is implemented as part of the compiler for the Orca language. Data is globally replicated and updates are broadcast as they occur, so write detection and consistency management occur as a single operation. Emerald [Jul et al. 88] and Amber [Chase et al. 89] are object-oriented systems that detect writes by way of runtime support. However, these systems only support data migration, not replication, so write detection and consistency management are combined.

There are a few efforts to distill shared memory into its basic operations. Tempest [Reinhardt et al. 94] is an interface definition for a hardware DSM that could be realized in software. Tempest requires both write *and* read detection. A software implementation, Blizzard-S [Schoinas et al. 94], has been done for the CM-5. However, Blizzard-S more closely emulates a hardware DSM and uses a more complex scheme for fine-grain detection than the scheme presented here. SAM [Scales & Lam 94] provides shared objects, not general shared memory, with new kinds of data called *accumulators* and *values*. A compiler can issue operations on these kinds of data much like our compiler instrumenting writes.

The languages community has explored the tradeoffs between runtime and page-based support for access detection within a single-node program [Hosking & Moss 93, Appel & Li 91]. The majority of these studies have concentrated on the use of write detection to implement efficient garbage collection in object-oriented and type-safe languages. Recently, software write detection has been investigated for other purposes, such as debugger breakpoints [Wahbe 92] and fault isolation [Wahbe et al. 93].

Our experiments, which are based on parallel programs running on a DSM system, differ from previous work in the choice of language used, whether or not the language is compiled or interpreted, style of programming (parallel vs. sequential), the existence of sharing and synchronization patterns, communication (bandwidth and latency) overheads, and the presence of fine-grained sharing.

The rest of this paper

In the next section we discuss the role of write detection in software-based DSM systems, and provide more details on the tradeoffs between RT-DSM and VM-DSM. In Section 3 we describe our implementation of both detection mechanisms in Midway. In Section 4 we compare the performance of a suite of benchmark applications using the two methods. Finally, in Section 5 we present our conclusions.

2 Reference detection and consistency protocols

Low-latency reads and writes represent the primary goal for any DSM system. To provide the abstraction of a single address space, software-based DSM systems must detect updates to shared memory and propagate the changes. Whether reads must also be detected depends on the consistency protocol. Read latency is decreased to local memory latency if an update protocol is used, since there are no read misses. An update-based protocol can increase the amount and frequency of communication between processors [Dwarkadas et al. 93]. Two techniques can be used to counter this effect. First, an update can be tied to a program's use of synchronization, enabling the updates to be coalesced and piggybacked on top of the necessary synchronization messages. The linking of synchronization to updates is accomplished by basing the DSM system on a weakly consistent memory model [Adve & Hill 93, Mosberger 93] as implemented in several systems [Carter et al. 91, Bershad et al. 93, Keleher et al. 94]. Second, the propagation of updates can be restricted to those processors that require the new data, rather than all processors, by taking advantage of the causal communication ordering revealed by a program's synchronization pattern [Keleher et al. 92, Bershad et al. 93, Keleher et al. 94]. Given that read latency can be minimized by an efficient update protocol, the problem becomes one of minimizing write latency, that is, detecting writes.

It might appear that write detection in an RT-DSM system is not as efficient as in a VM-DSM system because there is a constant overhead for each write. However, there are a number of factors to consider:

- The relative cost of a page fault is much greater than that of an individual write [Appel & Li 91, Hosking & Moss 93, Wahbe et al. 93, Thekkath & Levy 94]. In contrast, a write to shared memory can be instrumented with an overhead of just a few instructions.
- The virtual memory page size is too large for many applications, resulting in *false sharing* [Bennett et al. 90]. False sharing occurs when two data items share the same memory block, but are updated independently by two different processors, causing the memory block to be repeatedly transferred between the two processors. RT-DSM systems do not suffer from false sharing because the size of the unit of coherency can be set to meet the needs of the application.
- The number of instrumented writes in an RT-DSM system can be decreased by observing that shared memory parallel programs often have regions of memory dedicated to a particular processor, with just a few key data structures and variables shared among the processors. There is no need to instrument writes to memory that will not be referenced by other processors.

Whether RT-DSM or VM-DSM systems offer lower average write latency depends on a program's reference and synchronization pattern. For a coarse-grained program that exhibits little actual sharing, a VM-DSM system may be advantageous. In contrast, for a program that synchronizes (communicates) frequently, an RT-DSM system may be better. We consider these aspects of performance in Section 4.

3 Implementing write detection

In this section we describe the implementation of two write detection facilities, one using runtime support and one using virtual memory page protection. We have implemented both facilities in the context of Midway, a distributed shared memory system for a network of workstations to run parallel C programs. Our algorithms for page management under the VM-based strategy follow those described in the literature for other systems [Li & Hudak 89, Carter et al. 91, Keleher et al. 94].

Midway is a DSM system that provides the programmer with a weakly-consistent memory model called *entry consistency* [Bershad et al. 93]. Using entry consistency, processes synchronize explicitly via locks or barriers. Locks may be acquired in exclusive (for writing) or non-exclusive mode (for reading). When a processor acquires a lock that was last acquired on another processor, the first processor (the requester) must send a message to the second processor (the releaser) explicitly requesting the lock. A reply to this message represents a point of synchronization between the processors. The programmer provides the association between a lock or barrier and the data that the lock or barrier protects. At a synchronization point, entry consistency ensures that the data protected by the requested synchronization object reflects the most recent write to that data at the requesting processor. In practice, this requires that the DSM system determine the set of updates which have been performed at all other processors that have yet to be applied at the requesting site.

There are two aspects of each write detection scheme to consider. First, a write to shared memory must be trapped to ensure that the write can be later propagated to other processors. Second, at each synchronization point, a DSM system must collect the set of writes to be performed at the requesting processor. This set includes those actually issued on the releasing processor, and those writes issued on processors that had previously released the synchronization object. In the next two sections, we discuss in detail these two aspects, called *write trapping* and *write collection*, for both implementations.

3.1 Write trapping in RT-DSM

In Midway's RT-DSM implementation, write trapping is done entirely in user-level software. Every shared address cached on a processor has a dirtybit elsewhere in that processor's memory that reflects whether or not the address has been written. After each write to a shared address, the program sets a dirtybit associated with the modified address.

The strategy for write trapping relies on the careful arrangement of shared data and dirtybits in a processor's virtual memory. The application's virtual address space is partitioned into large, fixed size *regions*, as is illustrated in Figure 1. Data within a single region is either shared between all processors or private to each processor. The data within a shared region is divided into software *cache lines* and each cache line has a single dirtybit per processor. All cache lines in a region are the same size, although different regions may have different cache line sizes. The first page of each region contains a code template that updates the dirtybits for all data within that region. This code is tailored specifically for each region to contain the cache line size and location of the dirtybits as constants. The code template is generated when memory within the region is first allocated and write protected to prevent inadvertent modification.

To set a dirtybit, the compiler generates code that zeroes the low-order bits of the memory address being modified to produce the address of the dirtybit update code template. The generated code then calls the template passing the offset of the modified data within the region. The dirtybit update code computes the address of the dirtybit corresponding to the modified data, sets the dirtybit and returns. The compiler treats the call as a built-in and does not clobber temporary registers. Using common code for a region limits the inline code expansion and minimizes the perturbation in the processor's instruction cache. The code template also enables library code, such as *bcopy* and *scanf*, to be compiled only once and used on both shared and private memory.

Midway uses a modified version of the GCC compiler to emit the call to set the dirtybit after each write to shared memory. The compiler classifies program memory as either *shared* or *private*. By default, the program's static data is classified as shared and data stored on the program's stack is private. Memory referenced via a pointer is, by default, also classified as shared. All writable memory has a dirtybit template at the base of its region. The template for private memory returns to the caller without side-effects, although on our platform there is a six instruction penalty associated with misclassifying memory. The programmer can annotate the code with explicit `shared` or `private` attributes to change the classification of a type, a variable or a single assignment statement. These attributes are type-checked by the compiler and warnings are generated for mismatched assignments. In practice, it is relatively easy to accurately classify all memory, with misclassifications occurring quite infrequently.

We have found that the most frequently occurring type of write involves a doubleword store to a doubleword cache line. On a DECstation 5000/200 (MIPS R3000 [Kane 87]), this common case requires nine instructions (360 nanoseconds), of which four are generated inline by the compiler and five are in the dirtybit update code stored in the region's template. There are no load instructions which could miss in the cache and only one store instruction, which does not stall the processor given a sufficiently deep writebuffer. Instruction sequences for the dirtybit update code on a MIPS R3000 are shown in Appendix A.

3.2 Write collection in RT-DSM

Midway's RT-DSM implementation uses the dirtybits to collect the data to transfer at each synchronization point. There are two aspects to write collection: determining the data that the local processor has modified, and determining the data that has been previously modified, but has not been updated at the requesting processor. RT-DSM uses the dirtybits for both functions. A dirtybit is actually a timestamp that records the time of the most recent modification to each cache line. When a processor modifies a shared data item, it sets the associated dirtybit to the processor's local time.[1] The processor's local time is maintained as a *Lamport clock* [Lamport 78] to provide an ordering on the updates to an individual cache line.

The dirtybits allow RT-DSM to send the minimum number of updates necessary to maintain consistent memory. Updates are never performed more than once at a processor. The algorithm for collecting the minimal set of prior modifications is as follows. The first time a processor, P_1, acquires a lock (or crosses a barrier) it obtains all data bound to the lock, all dirtybits associated with the data, and the current logical time of the releasing processor. When the lock is subsequently transferred to another processor, P_1 updates the dirtybit timestamps for the data it modified to its current logical time. This time is also recorded by P_1 as the logical time at which the lock's data was consistent in the local cache. P_1 includes this time in a subsequent request to reacquire the lock from a remote processor, P_2. To make P_1's local cache consistent again, all data bound to the requested lock that has changed since P_1 last held the lock must be included in the reply message. The releasing processor, P_2, collects this set of data by scanning its dirtybits associated with the requested lock. Any dirtybit value greater than P_1's last timestamp indicates P_1's version of the associated data is inconsistent and must be updated. The new value and its timestamp are included in the lock reply message.

3.3 Write trapping in VM-DSM

Midway's VM-DSM implementation is similar to that of many systems using hardware page faults to detect writes to shared memory. VM-DSM uses Mach's external pager [Young et al. 87] to receive page fault notifications. The application's shared address space is mapped to allow a Midway runtime thread to service the application's paging requests. Initially all shared pages are mapped for read-only access and marked as clean. On the first store to each page, a write fault occurs and Midway creates a *twin* by saving a copy of the page. The page is marked as dirty, and write access is granted. The application's store is then completed and the faulting thread resumed. Subsequent writes to the page are not trapped and incur no additional overhead.

3.4 Write collection in VM-DSM

A reply message to a synchronization request must include any changes made by the releasing processor, as well as earlier changes by other processors that have not been performed at the requester. Each time a lock or barrier is transferred, its *incarnation number* is incremented and an

[1] In practice, the dirtybit timestamp is zeroed and lazily set to the processor's local time when the guarding synchronization object is transferred to another processor.

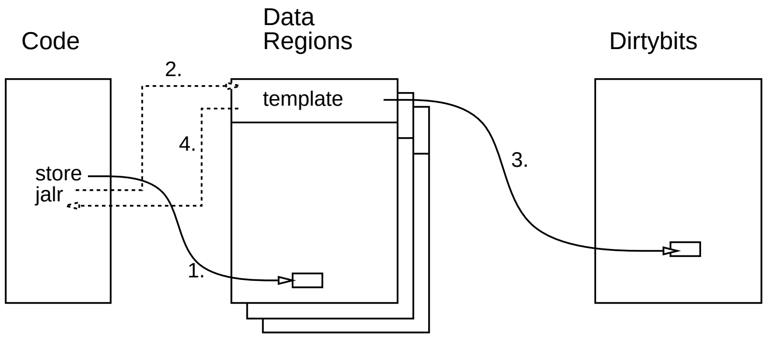

Figure 1: *This figure schematically represents a shared memory write. First the write is done. Based on the address of the modified data, a jump is made to the template for that region. The template computes the dirtybit address and stores a zero to mark the cacheline as dirty.*

update is created to encapsulate the modifications made to the data during that incarnation. The incarnation number imposes an ordering on the updates to the data bound to each synchronization object. In this sense, the incarnation number is similar to the cache line timestamps in the RT-DSM implementation. Each synchronization request includes the associated incarnation number at the last time it was held by the requester. Only updates with an incarnation number larger than the one indicated by the requester are transferred.

Midway's VM-DSM collects the data modified by the local processor by using the entry consistency binding information to locate the pages containing data bound to the requested lock or barrier. If any data on the page has been modified, the page will have been marked as dirty and a twin allocated. The contents of the page are compared against the original contents stored in the twin to create a *diff*. The diff is a succinct description of all modifications to the page. Modified data that is bound to the requested lock or barrier is collected into an update and shipped to the requesting processor. The diff created for each page is saved and may be reused to transfer other data on the page bound to different synchronization objects. When all modified data on the page has been shipped to other processors, the page is considered clean and its diff and twin deallocated. At the site which requested the lock, the incoming updates are applied to the shared data and the updates are saved. If the page being updated is currently marked as dirty, the update is applied to both the primary page and to the twin. Updating the twin ensures that the update will not be treated as a new modification by the local processor.

Updates from prior incarnations may either be combined by the releasing processor, or sent in their entirety and applied in incarnation order at the requester. The releasing processor has available the complete set of prior updates, because it saves the updates it receives when acquiring each lock. When subsequently releasing the lock, all prior updates required by the requesting processor are included in the reply message. There are no additional messages to other processors to obtain prior updates. However, Midway's implementation of VM-DSM does not save all the updates, nor does it combine updates from different incarnations. For example, if a memory location is written during incarnation 7 and then again during incarnation 8, any processor requesting a lock with a "last seen" incarnation number less than 7 would receive both writes. If the size of the concatenated updates exceeds the size of the data, a new update containing all of the bound data is sent instead. Combining updates requires that each address in the combined set reflect the most recent incarnation at which the address was written. We could maintain a finer-grained history by associating an incarnation number with each update to every shared address in the VM-DSM (as we do in RT-DSM), but have chosen not to do so. This scheme would incur at least the same data collection overhead as the RT-DSM (scan the incarnation numbers) and it would incur the additional overhead of trapping and detection for VM-DSM (write fault, twin, and diff).

3.5 Alternative approaches

The strategies described above represent points in the possible design space of write detection and collection mechanisms. There are alternate approaches that require neither software dirtybits nor virtual memory page faults to ensure consistency. For example, we could implement entry consistency by simply "blasting" all data associated with a synchronization object during interprocessor synchronization. There would be no need to detect writes at all (either with dirty bits or with page faults), since the guarantees of the consistency model could be provided through a conservative update protocol. This approach, although simple and having no immediate write overhead, would transfer data unnecessarily when synchronization objects guard large data objects being sparsely written.

A twinning and differencing algorithm, similar to our implementation for VM-DSM, could be used to reduce the amount of data transferred by the simple "blasting" approach described above. As data arrived on a processor, the processor would keep a second copy of each data item. At each synchronization point, all data bound to the synchronization object would be compared with its copy to determine which addresses have been modified. This approach avoids the cost of write detection, but increases the storage requirements (every shared data item must be twinned on any processor which writes it), and the synchronization overhead of the consistency mechanism (to diff unmodified data and to maintain the twin). Moreover, this approach would still require management of the update incarnations to ensure that a chain of processor updates are correctly propagated. As we show in the next section, data twinning and write collection constitute a significant portion of system overhead compared to the actual page fault time. Strategies that reduce the number of page faults by increasing the amount of data diffed cannot minimize the total cost of write detection.

Other memory models

A *targetted* memory consistency model delimits the addresses that must be made consistent whenever processors communicate. In contrast, an *untargetted* consistency model requires that the entire shared address space be made consistent. Entry consistency is a targetted model because only the data explicitly bound to the synchronization object is made consistent when processors synchronize using that object. Other consistency models, such as release consistency [Gharachorloo et al. 89], are untargetted. The consequence of an untargetted model is that a different style of bookkeeping is necessary during write trapping to delimit the scope of subsequent consistency operations. For example, Midway's RT-DSM strategy, when used with an untargetted memory model, would require scanning the dirtybit for every line cached in the processor's local memory. Implementations of write detection for untargetted models are more likely to rely on data structures better suited for sparseness.

There are two straightforward ways that Midway's RT-DSM strategy could be extended to support untargetted memory models. Both approaches slightly increase the write trapping overhead to reduce the subsequent write collection overhead. One approach is to use an update queue rather than a dirtybit array. Many updates are sequential, allowing a simple heuristic to substantially reduce the queue size. Although an update queue roughly triples the cost of write trapping, it keeps the cost of write detection proportional to the amount of dirty data, rather than the amount of shared data. An alternative approach is to represent dirtybits in a two level structure. Each dirtybit at the first level can "cover" a large number of dirtybits at the second. On a write to a shared address, the dirtybit at both levels associated with the written address is set. When synchronizing between processors, the first level dirtybits are scanned. If the first level bit has not been set, then no data in the second level is dirty and scanning of the second level bits can be avoided. Two-level dirtybits can be implemented with one additional store instruction in the write detection path, increasing its length by about 10%. Virtual memory page protection could also be used to implement the first level dirtybits. The second level dirtybits could be write protected so that a write to any one of them causes a page fault. The fault would result in the associated first level dirtybit being set and the page on which the second level bits are stored being made writable. As we show in the next section, write collection, and not write trapping, is the more expensive operation, therefore this seems like a practical approach.

Other memory consistency protocols invalidate cached memory rather than updating the data values. For applications that exhibit little write sharing, an invalidation protocol may reduce the total amount of data communicated. A consequence of invalidated memory is that reads, as well as writes, to shared data must be detected. Software read detection incurs a higher overhead than write detection, since read detection is more complicated and generally invoked more often than write detection. Applications with little write sharing also exhibit infrequent accesses to the invalidated data. By minimizing the cost of the common case and using the hardware to detect the exceptional events, we expect a VM-DSM system to incur lower overhead than an RT-DSM system. Of course, if the VM-DSM system exhibits a large number of read misses, than an update-based protocol is in order.

4 Experiments

In this section we describe the results of experiments using implementations of RT-DSM and VM-DSM described in the previous section. All experiments were run on a cluster of eight DECstation 5000/200's connected through Fore Systems TCA-100 TURBOchannel interface cards to a 140 MBit/sec ForeRunner ASX-100 ATM switch. The workstations run Mach 3.0 [Accetta et al. 86], have 25MHz MIPS R3000 processors, a 4KB page size, and 64 MBs of memory (no paging occurs during program execution). Our communication protocols use the ATM AAL3/4 adaptation layer directly, bypassing the Unix server [Brustoloni 94].

The experiments are intended to answer three questions:

1. Do programs using RT-DSM execute more quickly than when run under VM-DSM?
2. How does the cost of write trapping using RT-DSM compare to that using VM-DSM?
3. How much overhead is required during processor synchronization to collect the set of modified data that must be transferred?

We selected and ran five applications to answer these questions. These applications have sharing and computation patterns that are representative of a range of parallel programs. The applications were modified to use entry consistency and to run on both the RT-DSM and VM-DSM systems.

The first application, `water`, is a N-body molecular dynamics simulation from the Splash application suite [Singh et al. 92]. The program evaluates forces and potentials for a system of 343 water molecules in a liquid state for 5 steps. It exhibits medium-grained sharing. Our version of `water` has the optimization suggested in [Singh et al. 92], which collects changes to the molecules in private memory during a time step, updating the shared molecules only at the end of each time step.

The second application, `quicksort`, is one of the TreadMarks applications [Keleher et al. 94]. It sorts an array of $250,000$ integers using a parallel quicksort algorithm until the partition size is less than a threshold of 1000 elements and then sorts locally using a bubblesort algorithm. This program exhibits medium to coarse-grain sharing, but does little computation between writes to shared memory (the inner loop does a compare and swap of adjacent elements in the array). The array is partitioned dynamically, so the lock binding the data to the task queue element is rebound to a new range of addresses for every task created.

The `matrix-multiply` application exhibits coarse grain sharing with a high computation to communication ratio. This program multiplies two 512×512 floating point matrices. The `matrix-multiply` program is of interest because its data is partitioned to minimize the amount of sharing and because it writes every word on every page of the result matrix. The large number of writes to each page helps the VM-DSM system best amortize the cost of the initial page fault. By minimizing the amount of fine-grain sharing, the program also minimizes the cost of twinning and diffing. This represents the expected best case for VM-DSM, and the worst case for RT-DSM. For the RT-DSM system, every write incurs the overhead to set the associated dirtybit.

The application `sor` implements a red-black successive over-relaxation algorithm to calculate the steady state temperature of a two-dimensional rectangular plate, given a fixed temperature at each edge. The program iteratively computes new values for each element in a 1000×1000 matrix of floating point values. The matrix data is laid out with red and black elements adjacent in memory and is not partitioned to match the peculiarities of the memory system. Only data at the edges of each partition are shared between processors. The interior elements are initialized to random values to maximize the changed elements per iteration. The program runs for 25 iterations and exhibits medium-grain sharing.

The final application, `cholesky`, is from the Splash application suite and performs a parallel Cholesky factorization of a sparse matrix. Given a positive definite matrix A, the program finds a lower triangular matrix L, such that $A = LL^T$. This program exhibits fine-grain sharing.

Overall performance

The graphs in Figure 2 show the execution time and the amount of data transferred for each application under both RT-DSM and VM-DSM. The graphs show that VM-DSM dominates RT-DSM only for `quicksort`. The other programs run either faster, or no slower when using run time detection. `quicksort` frequently changes the association between locks and data. When a lock binding changes under RT-DSM, the dirtybits must still be scanned for every lock transfer. Under VM-DSM, the incarnation number is incremented which causes all data bound to the lock to be sent without performing a diff. Since this application modifies nearly all of the bound data on every transfer, the VM-DSM approach is better. Although the `matrix-multiply` program has expected worst-case performance for RT-DSM and best-case performance for VM-DSM, the graphs show only a minor difference between the two approaches. The medium and fine-grained programs, `water`, `sor`, and `cholesky`, all exhibit lower execution times and transmit less data under RT-DSM than under VM-DSM.

The graphs in Figure 2 also show the execution time for a standalone, uniprocessor version of each application. The standalone version contains no code for write detection or synchronization, which is included in both the RT-DSM and VM-DSM versions, and thus executes faster than either version. The uniprocessor times for `water` under RT-DSM, VM-DSM and a standalone system are 110.1, 109.1, and 104.2 seconds, respectively. The execution time for the uniprocessor RT-DSM version is highest since it pays the entire cost for write detection. The VM-DSM version pays for a single write fault on each shared page. It never diffs or write protects a page, since the data is never transferred. Thus its overhead is lower than the RT-DSM version. Once the application runs in parallel, the cost of write collection becomes important and we see RT-DSM outperform VM-DSM.

4.1 The cost of write trapping

The previous section presented overall program performance. In this and the next section, we discuss the performance of a write detection mechanism's two components: write trapping and write collection. We determine the cost of write trapping by examining the frequency and cost of the primitives required for its implementation under RT-DSM and VM-DSM. The primitive operations for RT-DSM are (1) set a dirty bit after a shared write and (2) set a dirty bit after a private write. For VM-DSM, the primitive operations are (1) field a write fault, (2) twin a page, and (3) set the page protection. The measured cost of each primitive operation is given in Table 1. Table 2 summarizes the invocation counts for each application as a per processor

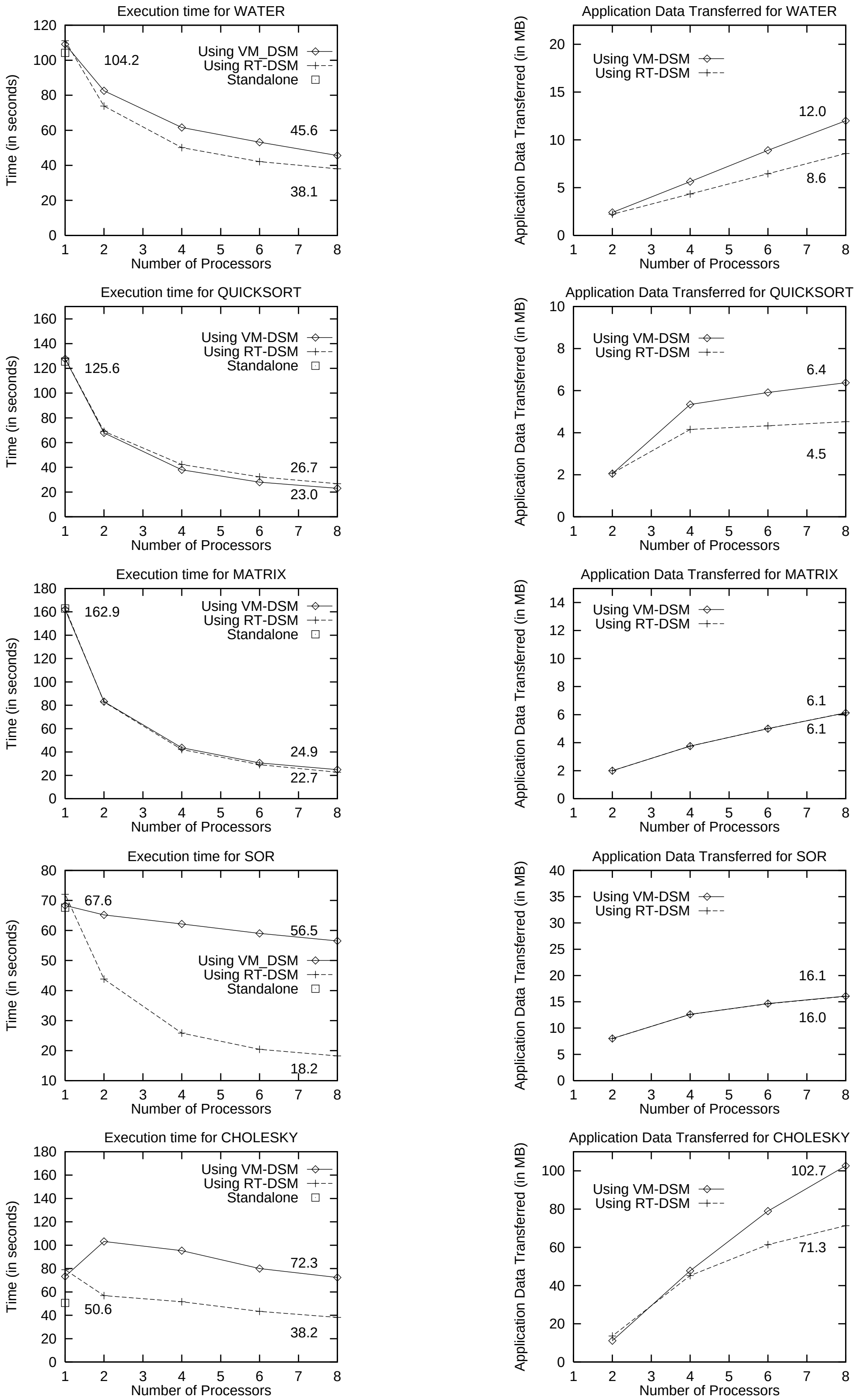

Figure 2: *Comparison of execution times (in seconds) and total data transferred (in MB) for our applications. The graphs represent an average over 5 runs of each application. The average value of the standalone execution time, eight processor execution time, and data transferred in an eight processor execution are also shown.*

System	Primitive Operation	Time (μsecs)	Cycles
RT-DSM	dirtybit set time		
	for a word write	0.360	9
	for a doubleword write	0.360	9
	for writing to private memory	0.240	6
	dirtybit read time		
	for each "clean" dirtybit read	0.217	5
	for each "dirty" dirtybit read	0.187	4
	dirtybit write time		
	for each dirtybit updated	0.067	2
VM-DSM	page write fault		
	includes page copy & protection call	1,200	30,000
	page diff time		
	if none or all of the data changed	260	7,000
	if every other word changed	1,870	46,750
	page protection call		
	to allow read-write access	125	3,125
	to allow read-only access	127	3,175
	memory block copy time		
	for each KB of data in a cold cache	84	2,100
	for each KB of data in a warm cache	26	650

Table 1: *Execution times for primitive operations on a 25MHz MIPS R3000 processor running Mach 3.0. The page size is 4 KB. All times are measured averages and are given in microseconds.*

		Applications				
System	Operation	Water	Quicksort	Matrix	SOR	Cholesky
RT-DSM	dirtybits set	43,180	220,804	98,311	348,516	1,284,004
	dirtybits misclassified	0	124	11	1	28
	clean dirtybits read	48,552	98,190	135,776	19,185	2,568,269
	dirty dirtybits read	11,280	108,939	94,217	261,097	739,625
	dirtybits updated	35,676	147,896	200,849	262,987	1,132,009
	data transferred (in KB)	1,096	579	784	2,053	9,128
	percent dirty data	55.7	62.7	87.4	98.1	29.3
VM-DSM	write faults	258	156	74	468	2,916
	pages diffed	253	27	120	674	3,107
	pages write protected	253	27	120	674	3,107
	data updated in twins (in KB)	976	418	15	47	5,114
	data transferred (in KB)	1,543	816	784	2,058	13,144

Table 2: *Per processor invocation counts of the primitive operations. All numbers are averages for all processors in an 8-way run. The "data transferred" number measures only application data, it does not include protocol overhead.*

average. By multiplying the counts in Table 2 by the time for each primitive operation from Table 1, we can estimate the cost for write trapping in each system. A summary of the write trapping costs for each applications is presented in Table 3. For example, we compute the write trapping cost for `water` as follows: under RT-DSM, each processor set 43,180 dirtybits (none were misclassified), requiring 0.360 μsecs each for a total time of 16 msecs; under VM-DSM, each processor incurred 258 write faults, requiring 1200 μsecs each to handle the fault, make a twin and set the page protection, for a total time of 310 msecs. Times for the other programs can be similarly determined.

The impact of fast exceptions

Midway's VM-DSM implementation uses Mach's external pager, which is relatively expensive, to trap writes. A write exception, though, can be delivered in as few as 18 μsecs on the hardware used by Midway [Thekkath & Levy 94]. Despite this, the VM-DSM fault handler must still copy the 4KB page to its twin, increasing the best-case time to service a write fault to 122 μsecs. We can estimate the contributed cost of write trapping when using a fast exception handler. The graph in Figure 3 shows this cost for a range of write trapping overheads, comparing against the write trapping cost for an RT-DSM system. Each application is represented by a horizontal line showing the effect of varying the page fault service time between 122 μsecs and 1200 μsecs. The

System	Operation	Applications				
		Water	Quicksort	Matrix	SOR	Cholesky
RT-DSM	write trapping time	15.6	79.5	35.4	125.5	485.3
VM-DSM	write trapping time	309.6	187.2	88.8	561.6	3,499.2
RT-DSM trapping advantage		**294.0**	**107.7**	**53.4**	**436.1**	**3,103.9**

Table 3: *Summary of the time for write trapping. All times are in milliseconds and are computed by measuring the costs of the primitive operations and multiplying by the average per-processor number of invocations for each application.*

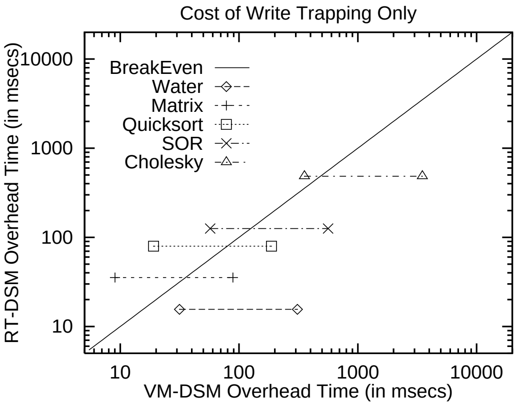

Figure 3: *The effect of varying page fault cost on write detection. Each application is a line with the left end represents the costs incurred by VM-DSM with a time to service a page fault of 122 μsecs using Thekkath and Levy's fast exception handler. The right end represents the costs with a page fault service time of 1200 μsecs using Mach's external pager. The diagonal line represents the break even point between VM-DSM and RT-DSM. Points beneath the line show a lower cost for RT-DSM.*

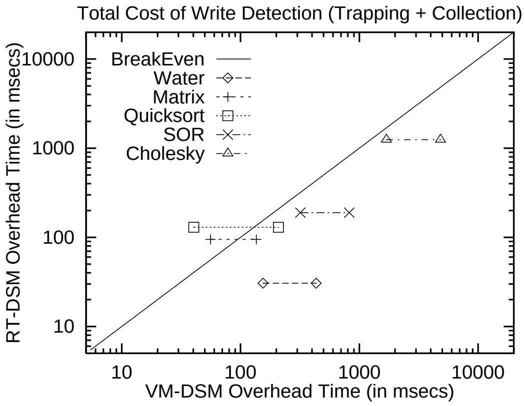

Figure 4: *The effect of varying page fault cost on the total cost of write detection, including both trapping and collection. The graph is interpreted as in Figure 3. The break even cost for handling a page fault is 650 μsecs for* `matrix-multiply` *and 696 μsecs for* `quicksort`.

break even line for the cost of write detection is shown as the diagonal line. All points beneath the diagonal line indicate that write trapping overhead will be lower when using RT-DSM as opposed to VM-DSM. If the line for the application crosses the break even point, write trapping for that application will be less expensive when using a VM-DSM system with a fast exception mechanism.

The graph shows that most applications span the break even point. This indicates the cost of write trapping under VM-DSM is strongly dependent on the cost of the platform's exception handler. For the coarse-grained applications, `matrix-multiply` and `quicksort`, the actual overhead is small and neither system outperforms the other. For the medium and fine-grained applications, the costs are higher and tend to favor the RT-DSM approach.

4.2 The cost of write collection

The cost of trapping writes is only one part of the total cost for write detection. We must also account for the cost of collecting modified data at each synchronization point. We can determine this cost by examining the frequency and cost of the primitive operations required for its implementation under RT-DSM and VM-DSM. The primitives for RT-DSM are (1) read a dirtybit when collecting data, (2) reset the dirtybit for modified data, and (3) update a dirtybit with a new timestamp value for data updated at the requesting processor. For VM-DSM, the primitive operations are (1) compare a page during diffing, (2) change the page protection, and (3) update twinned data at the requesting processor. The measured cost of each primitive operation is given in Table 1.

The count of invocations of each primitive operation is summarized in Table 2. By multiplying the counts in Table 2 by the time for each primitive operation from Table 1, we can estimate the average cost per processor for write collection in each system. The results are summarized in Table 4. We see that, with the exception of `quicksort` where diffing is not performed due to the lock rebinding, write collection with VM-DSM is more expensive than with RT-DSM. Furthermore, the cost of write collection increases with the amount of write sharing; fine-grained applications incur a higher percentage overhead than coarse-grained ones.

The break even point between VM-DSM and RT-DSM for write collection is a function of the percentage of data that has been modified by the local processor. It does not depend on page fault overhead. If none of the data bound to the requested synchronization object has been modified, nor has any of the data on the containing pages, then VM-DSM can avoid the cost of diffing. The RT-DSM system,

System	Operation	Applications: Water	Quicksort	Matrix	SOR	Cholesky
RT-DSM	clean dirtybits read	10.5	21.3	29.5	0.5	557.3
	dirty dirtybits read	2.0	19.2	16.6	46.0	138.3
	dirtybits updated	2.4	9.9	13.5	17.6	75.8
	Total	14.9	50.4	59.6	64.1	771.4
VM-DSM	pages diffed	65.8	7.0	31.2	175.2	807.8
	pages write protected	32.1	3.4	15.2	85.6	394.6
	data updated in twins	25.4	10.9	0.4	1.2	133.0
	Total	123.3	21.3	46.8	262.0	1,335.4
RT-DSM collection advantage		**108.4**	**-29.1**	**-12.8**	**197.9**	**564.0**

Table 4: *Summary of the cost for write collection as a per-processor average. All times are in milliseconds and are computed by measuring the costs of the primitive operations and multiplying by the average number of per-processor invocations for each application.*

however, must still scan all of the dirtybits bound to the requested object. In practice, we found that often much of the bound data is modified, nullifying this possible advantage. Table 2 shows the percentage of dirty data bound to the synchronization object for each of the applications.

The graph in Figure 4 shows the accumulated cost of write trapping and write collection for each application across a range of page fault times. Like Figure 3, the break even line for the total cost is shown as the diagonal line. All points beneath the diagonal indicate a lower overhead when running under RT-DSM than under VM-DSM. If the application's line crosses the break even line, the application will have a lower total cost using a VM-DSM system with the fast exception mechanism, but a higher cost using VM-DSM on our system. The coarse-grained applications do not exhibit sufficient sharing to be strongly affected by either approach and are again near the break even line. The `quicksort` application favors VM-DSM since its calls to rebind data to locks allow VM-DSM to avoid the overhead of diffing. The medium and fine-grain applications have moved substantially to the right. This indicates that the cost of write collection is significant, and that even with an optimized exception handler RT-DSM dominates VM-DSM.

Memory reference activity

Collecting writes is memory intensive. We can partially account for the cost differential between RT-DSM and VM-DSM by observing the number of additional memory references required with each approach. For example, under VM-DSM the number of memory references is equal to twice the number of words per twinned page (read the original, write the twin), plus twice the number of words per diffed page (read the original, read the twin), plus the number of words applied to each twin during update. Under RT-DSM, it is the sum of the dirtybits set, clean dirtybits read, dirty dirtybits read (times 2 to store the timestamp in the dirtybit value), and the dirtybits updated to set the timestamps at the requesting processor. We can compare the number of memory references incurred by each approach using the primitive operation counts in Table 2. Table 5 shows that for medium and fine-grain applications, RT-DSM incurs substantially fewer memory references, primarily because it avoids the page twin and diff operations. As memory reference times continue to increase relative to CPU cycle time, we expect the impact of this difference will grow.

Data transferred

The graphs and figures show that Midway's implementation of RT-DSM transfers less data than VM-DSM, although both use the same consistency protocol. As previously mentioned, the difference stems from how the two implementations maintain update histories. An exact history, like that made possible by the RT-DSM's dirtybit timestamps, is necessary to minimize the total data transferred. The amount of redundant data transferred can be substantial. For example, `water` sent an extra 441 KB/processor (40%), and `cholesky` sent an extra 4015 KB/processor (44%). As network bandwidth increases, the impact of the additional data load on the network may decrease, however it will still remain necessary to construct the packets containing the excess data, and to needlessly reapply the updates at the requesting processor's local cache.

4.3 Summary

Our analysis divides the cost of write detection into two components, write trapping and write collection. For the VM-DSM system, the cost of write trapping depends on the platform's cost for exception handling. Even for systems with fast exceptions, an RT-DSM system has comparable or lower write trapping overhead for most applications. The total cost of write trapping is relatively low, and write collection (which is memory intensive) is the dominant component of write detection. The dirtybit scheme used by RT-DSM has low fixed overheads and naturally accommodates fine-grain sharing. The RT-DSM dirtybit scheme also provides an update history, reducing both the overhead and the data sent at each synchronization point.

System	Operation	Applications				
		Water	Quicksort	Matrix	SOR	Cholesky
RT-DSM	write trapping	43	221	98	349	1,349
	write collection	96	355	431	526	4,440
	Total	139	576	529	875	5,788
VM-DSM	write trapping	510	358	262	1,264	5,767
	write collection	768	162	250	1,392	7,672
	Total	1,278	521	512	2,656	13,439
RT-DSM memory reference advantage		**1,139**	**-55**	**-17**	**1,781**	**7,651**

Table 5: *Total memory references incurred for write detection under RT-DSM and VM-DSM. All counts are in units of* 1000 *and are per-processor averages.*

5 Conclusions

We have presented a new technique for detecting writes in a distributed shared memory system, and have compared its performance to a page-based strategy. A system using runtime write detection in conjunction with an update-based protocol can exhibit lower overhead when compared to a page-based system. The base cost of servicing a page fault is only one factor in the total overhead, the dominant cost is the mechanism to support fine-grain sharing. The overhead incurred using runtime write detection does not depend on the granularity of sharing, allowing runtime detection to more efficiently support fine-grained applications. Our compiler-based detection mechanisms are currently being used in the Midway distributed shared memory system at Carnegie Mellon University.

Acknowledgments

José Brustoloni implemented the network drivers for our Mach 3.0 Fore ATM interfaces. Dave McKeown has been invaluable in obtaining the resources we used, as well as providing a user's perspective. Finally, Willy Zwaenepoel helped us clarify much of this paper's presentation.

References

[Accetta et al. 86] Accetta, M. J., Baron, R. V., Bolosky, W., Golub, D. B., Rashid, R. F., Avadis Tevanian, J., and Young, M. W. Mach: A New Kernel Foundation for Unix Development. In *Proceedings of the 1986 Summer USENIX Conference*, pages 93–113, July 1986.

[Adve & Hill 93] Adve, S. V. and Hill, M. D. A Unified Formalization of Four Shared-Memory Models. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):613–624, June 1993.

[Appel & Li 91] Appel, A. W. and Li, K. Virtual Memory Primitives for User Programs. In *Proceedings of the Fourth ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 96–107, April 1991.

[Bal et al. 92] Bal, H. E., Kaashoek, M. F., and Tanenbaum, A. S. Orca: A Languange For Parallel Programming of Distributed Systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[Bennett et al. 90] Bennett, J. K., Carter, J. B., and Zwaenepoel, W. Adaptive Software Cache Management for Distributed Shared Memory Architectures. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 125–134, May 1990.

[Bershad et al. 93] Bershad, B. N., Zekauskas, M. J., and Sawdon, W. A. The Midway Distributed Shared Memory System. In *Proceedings of the 1993 IEEE CompCon Conference*, pages 528–537, February 1993.

[Brustoloni 94] Brustoloni, J. C. Exposed Buffering and Sub-Datagram Flow Control for ATM LANs. In *Proceedings of the 19th Annual Conference on Local Computer Networks*. IEEE Computer Society, September 1994.

[Bryant et al. 91] Bryant, R., Carini, P., Chang, H.-Y., and Rosenburg, B. Supporting Structured Shared Virtual Memory under Mach. In *Proceedings of the Second Mach USENIX Symposium*, pages 59–76, November 1991.

[Carter et al. 91] Carter, J., Bennett, J., and Zwaenepoel, W. Implementation and Performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[Chase et al. 89] Chase, J. S., Amador, F. G., Lazowska, E. D., Levy, H. M., and Littlefield, R. J. The Amber System: Parallel Programming on a Network of Multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 147–158, December 1989.

[Dwarkadas et al. 93] Dwarkadas, S., Keleher, P., Cox, A. L., and Zwaenepoel, W. Evaluation of Release Consistent Software Distributed Shared Memory on Emerging Network Technology. In *Proceedings of the 20th Annual Symposium on Computer Architecture*, pages 144–155, May 1993.

[Gharachorloo et al. 89] Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., and Hennessy, J. Memory Consistency and Event Ordering in Scalable Shared Memory Multiprocessors. In *Proceedings of the 16th Annual Symposium on Computer Architecture*, pages 15–26, May 1989.

[Hosking & Moss 93] Hosking, A. L. and Moss, J. E. B. Protection Traps and Alternatives for Memory Management of an Object-Oriented Language. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 106–119, December 1993.

[Jul et al. 88] Jul, E., Levy, H., Hutchinson, N., and Black, A. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[Kane 87] Kane, G. *MIPS R2000 RISC Architecture*. Prentice Hall, 1987.

[Keleher et al. 92] Keleher, P., Cox, A. L., and Zwaenepoel, W. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, pages 13–21, May 1992.

[Keleher et al. 94] Keleher, P., Cox, A. L., Dwarkadas, S., and Zwaenepoel, W. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the 1994 Winter USENIX Conference*, pages 115–132, January 1994.

[Lamport 78] Lamport, L. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[Li & Hudak 89] Li, K. and Hudak, P. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[Minnich & Farber 89] Minnich, R. and Farber, D. The Mether System: A Distributed Shared Memory for SunOS 4.0. In *Proceedings of the 1989 Summer USENIX Conference*, June 1989.

[Mosberger 93] Mosberger, D. Memory Consistency Models. Technical Report TR 93/11, Department of Computer Science, University of Arizona, 1993.

[Reinhardt et al. 94] Reinhardt, S. K., Larus, J. R., and Wood, D. A. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 325–336, April 1994.

[Scales & Lam 94] Scales, D. J. and Lam, M. S. The Design and Evaluation of a Shared Object System for Distributed Memory Machines. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, November 1994.

[Schoinas et al. 94] Schoinas, I., Falsafi, B., Lebeck, A. R., Reinhardt, S. K., Larus, J. R., and Wood, D. A. Fine Grained Access Control for Distributed Shared Memory. In *Proceedings of the Sixth ACM Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.

[Singh et al. 92] Singh, J. P., Weber, W.-D., and Gupta, A. SPLASH: Stanford Parallel Applications for Shared Memory. *Computer Architecture News*, 20(1):2–12, March 1992.

[Thekkath & Levy 94] Thekkath, C. A. and Levy, H. M. Hardware and Software Support for Efficient Exception Handling. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.

[Wahbe 92] Wahbe, R. Efficient Data Breakpoints. In *Proceedings of the Fifth ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 200–212, September 1992.

[Wahbe et al. 93] Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 203–216, December 1993.

[Young et al. 87] Young, M., Tevanian, A., Rashid, R., Golub, D., Eppinger, J., Chew, J., Bolosky, W., Black, D., and Baron, R. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *The Proceedings of the 11th Symposium on Operating System Principles*, November 1987.

Appendix A: Dirtybit Update Code

Every dirtybit update is handled by two separate code sequences. The compiler generates the first sequence of code inline, immediately following the store to shared memory. The exact instruction sequence depends on the alignment of data and the amount of data written (eg. byte, word, doubleword). The instructions compute the address of the second code sequence, stored at the base of the region of memory being modified. The second code sequence actually sets the dirtybit to mark the cache line as "dirty."

All writable memory is logically divided into regions. The first page of the region is write protected and contains the code to update the dirtybits for data within that region. When the region is first allocated, a template of the dirtybit update code is copied to the base of the region and is modified to include as constants in the instruction stream the cache line size and the location of the dirtybits. Each template contains five entry points to set the dirtybit for a store of a byte, halfword, word, doubleword and an arbitrary sized range of bytes, called an "area". For shared regions, the template depends on the cache line size, allowing an optimization for word and doubleword size cache lines. For private regions, each entry point in the template simply returns.

For floating point applications, the common case is a doubleword store to doubleword size cache line. On the MIPS R3000 this requires 9 instructions, 4 generated inline by the compiler and 5 in the dirtybit update template. The exact sequence is shown in Figure 5. The mask for computing the base of the region is a constant and the compiler may allocate a register to it and avoid reloading it for subsequent stores. The compiler treats the dirtybit update call as a built-in and will only clobber registers a0, a1, and ra.

For integer applications, the common case is a single word store to a word size cache line. This is shown in Figure 6. The compiler must generate one extra instruction inline to compute the address of the entry point within the template for the dirtybit code for word writes. The code in the template saves an instruction, since the offset in the data region equals the offset in the dirtybits.

The most expensive case is when writing an unaligned object or for area operations, such as structure assignments. The compiler must generate an extra instruction to pass the size of the object modified as an argument to the dirtybit code. The inline code is shown in Figure 7. The code stored in the template allocates a stack frame, saves the temporary registers, and then calls a higher-level routine to perform the actual work. This sequence is rarely invoked. It was called only once by one of the applications presented.

If the compiler misclassifies a write to private memory, it will generates a call to the dirtybit update code for that region. The code for all entry points for private memory simply returns to the caller and is shown in Figure 8.

```
#
# Code generated inline by the compiler for a doubleword write
#
        sw      <val_lo>, 0(rx)         # original store instruction of low order word
        sw      <val_hi>, 4(rx)         # original store instruction of high order word
        lui     a0, <mask_for_template> # load mask for start of region address
        and     at, a0, rx              # generate addr for dirtybit template
        jalr    at                      # jump to dirtybit update code
        sub     a0, rx, a0              #  compute offset w/in region (jump delay slot)
#
# Code stored in template for a doubleword write to a doubleword size cache line
#
        lui     at,<dbit_address>       # load addr of start of dbits for region
        srl     a1, a0, 1               # divide offset by 2 to get dbit offset
        addu    at, a1, at              # generate address of dbit
        jr      ra                      # and return
        sw      zero, 0(at)             #  zero dbit to mark as ``dirty''
```

Figure 5: *Doubleword write to a doubleword size cache line.*

```
#
# Code generated inline by the compiler for a word write
#
        sw      <val>, 0(rx)            # original store instruction of word
        lui     at, <mask_for_template> # load mask for start of region addr
        and     a0, at, rx              # generate addr for dirtybit template
        or      at, a0, <entryW_offset> # gen addr for entry point w/in template
        jalr    at                      # jump to dirtybit update code
        sub     a0, rx, a0              #  compute offset within region
#
# Code stored in template for a word writes to a word size cache line
#
        lui     at,<dbit_address>       # load addr of start of dbits for region
        addu    at, a1, at              # generate address of dbit
        jr      ra                      # and return
        sw      zero, 0(at)             #  zero dbit to mark as ``dirty''
```

Figure 6: *Word write to a word size cache line.*

```
        sw      <val>, 0(rx)            # example of original store instuction
        lui     at, <mask_for_template> # load mask for start of region address
        and     a0, at, rx              # generate addr for dirtybit template
        or      at, a0, <entryA_offset> # generate addr for entry point w/in template
        addi    a1, zero, <object_size> # arg1 is size of the object written
        jalr    at                      # jump to dirtybit update code
        sub     a0, rx, a0              #  arg0 is the offset w/in region (jump delay slot)
```

Figure 7: *Code generated inline by the compiler, following an unaligned write or a write to an object larger than a doubleword.*

```
        jr      ra                      # simply return to caller
        nop                             #  fill jump delay slot
```

Figure 8: *Code stored in template for all writes to unshared memory.*