Accepted for publication in IEEE Transactions on Parallel and Distributed Systems (TPDS), accepted July 1998.

A Protocol-Centric Approach to On-The-Fly Race Detection

Dejan Perković and Peter J. Keleher

Department of Computer Science, University of Maryland

College Park, MD 20742-3255

(dejanp|keleher)@cs.umd.edu

Index terms: data races, on-the-fly, DSM, shared memory

Abstract

We present the design and evaluation of a new data-race-detection technique. Our technique executes at runtime rather than post-mortem, and handles unmodified shared-memory applications that run on top of CVM, a software distributed shared memory system. We do not assume explicit associations between synchronization and shared data, and require neither compiler support nor program source. Instead, we use a binary code re-writer to instrument instructions that may access shared memory.

The most novel aspect of our system is that we are able to use information from the underlying memory system implementation in order to reduce the number of comparisons made at run time.

We present an experimental evaluation of our techniques by using our system to look for data races in five common shared-memory programs. We quantify the effect of several optimizations to the basic technique: data flow analysis, instrumentation batching, runtime code modification, and instrumentation inlining. Our system correctly found races in three of the five programs, including two from a standard benchmark suite. The slowdown of this debugging technique averages less than 2.5 for our applications.

1 Introduction

Despite the potential savings in time and effort, data-race detection techniques are not yet an accepted tool of builders of parallel and distributed systems. Part of the problem is surely the restricted domain in which most such mechanisms operate, i.e., parallelizing compilers. Compiler support is usually deemed necessary because race-detection is generally NP-complete [19].

This paper presents the design and evaluation of an on-the-fly race-detection technique for explicitly parallel shared-memory applications. This technique is applicable to shared memory programs written for the lazy-release-consistent (LRC) [11] (see Section 3.1) memory model. Our work differs from previous work [3, 4, 7, 9, 18, 17] in that data-race detection is performed both on-the-fly *and* without compiler support. In common with other dynamic systems, we address only the problem of detecting data races that occur in a given execution, not the more general problem of detecting all races allowed by program semantics [19, 25]. Earlier work [21] introduced this approach by demonstrating its use on a less complex single-writer protocol. This paper extends this earlier work through the use of a more advanced multi-writer protocol, and through a series of optimizations to the basic technique.

We find data races by running applications on a modified version of the Coherent Virtual Memory (CVM) [13, 14] software distributed shared memory (DSM) system. DSMs support the abstraction of shared memory for parallel applications running on CPUs connected by general-purpose interconnects, such as networks of workstations or distributed memory machines like the IBM SP-2. The key intuition of this work is the following:

> *LRC implementations already maintain enough ordering information to make a constant-time determination of whether any two accesses are concurrent.*

In addition to the LRC information, we track individual shared accesses through binary instrumentation, and run a simple race-detection algorithm at existing global synchronization points. This last task is made much easier precisely because of the synchronization ordering information maintained by LRC. The system can automatically generate global synchronization points in long-running applications if there are none originally.

We used this technique to check for data races in implementations of five common parallel applications. Our system correctly found races in three. Water-Nsquared and Spatial, from the Splash2 [27] benchmark suite, had data races that constituted real bugs. These bugs have been reported to the Splash authors and fixed in their current version. While the races could affect correctness, they were unlikely to occur on the platforms for which SPLASH was originally intended. Barnes, on the other hand, had been modified locally in order to eliminate unnecessary synchronizations. The races introduced by these modifications did not affect the correctness of the application.

Since overhead is still potentially exponential, we describe a variety of techniques that greatly reduce the number of comparisons that need to be made. Those portions of the race-detection procedure that have the largest theoretical complexity turn out to be only the third or fourth-most expensive component of the overall overhead.

Specifically, we show that i) we can statically eliminate over 99% of all load and store instructions as potential race participants, ii) we eliminate over 80% potential comparisons at runtime through use of LRC ordering information, and iii) the average slowdown from use of our techniques is currently less than 2.8 on our applications, and could be reduced even further with support for inlining of instrumentation code. While this overhead is still too high for the system to be used all of the time, it is low enough for use when traditional debugging techniques are insufficient, or even to be a part of standard debugging toolbox for parallel programs.

2 Problem Definition

We paraphrase Adve's [1] terminology to define the data races detected by our system.

Definition 1 *We define the* happened-before-1 *partial order, denoted* $\xrightarrow{\text{hb1}}$*, over shared accesses and synchronization* acquires *and* releases *as follows:*

1. *If a and b are ordinary shared memory accesses, releases, or acquires on the same processor, and a occurs before b in program order, then $a \xrightarrow{\text{hb1}} b$.*
2. *If a is a release on processor p_1, and b is the corresponding acquire on processor p_2, then $a \xrightarrow{\text{hb1}} b$. For a lock, an acquire corresponds to a release if there are no intervening acquires or releases of that lock. For a barrier, an acquire corresponds to a release if the acquire is a departure from and the release is an arrival to the same instance of the barrier.*
3. *If $a \xrightarrow{\text{hb1}} b$ and $b \xrightarrow{\text{hb1}} c$, then $a \xrightarrow{\text{hb1}} c$.*

Given Definition 1, we define data races as follows:

Definition 2 *Shared accesses a and b constitute a* data race *if and only if:*

1. *a and b both access the same word of shared data, and at least one is a write, and*
2. *Neither $a \xrightarrow{\text{hb1}} b$ nor $b \xrightarrow{\text{hb1}} a$ is true.*

Definition 2 approximates the notion of *actual* data races defined by Netzer [20].

In common with most other implemented systems, both with and without compiler support, we make no claim to detect all data races allowed by the semantics of the program (the *feasible* races discussed by Netzer [20]). As such, a program running to completion on our system without data races is not a guarantee that subsequent executions will be free of data races as well. However, we do detect all races that occur in any given execution.

Figure 1 shows two possible executions of code in which processes access shared variable x and synchronize through synchronization variable L. The access pair $w_1(x) - r_1(x)$ in the execution on the left does not constitute a data race because $w_1(x) \xrightarrow{\text{hb1}} r_1(x)$. However, if program semantics do not enforce an ordering on lock acquisitions, the execution might instead have happened as shown in 1(b). In this case, $r_1(x)$ is not ordered with respect to $w_1(x)$, and the two therefore constitute a race.

Note that not all data races cause incorrect results to be generated. "Correct" results will be generated even for the execution on the right if r_1 completes before w_1 is issued.

In order for the system to distinguish between the accesses in Figure 1, the system must be able to detect and understand the semantics of all synchronization used by the programs. In practice, this requirement means that programs must use only system-provided synchronization. Any synchronization implemented on top of the shared memory abstraction is invisible to the system, and could result in spurious race warnings.

However, the above requirement is no stricter than that of the underlying DSM system. Programs must use system-visible synchronization in order to run on any release-consistent system. Our data-race detection system imposes no additional consistency or synchronization constraints.

3 Lazy Release Consistency and Data Races

P_1	P_2
$w_1(x)$	
Acquire(L)	
$w_2(x)$	
Release(L)	
	Acquire(L)
	$r_1(x)$
	Release(L)

(a)

P_1	P_2
	Acquire(L)
	$r_1(x)$
	Release(L)
$w_1(x)$	
Acquire(L)	
$w_2(x)$	
Release(L)	

(b)

Figure 1. If the ordering of accesses to lock L is non-deterministic, either (a) or (b) is a possible ordering of events. The ordering given in (a) is not a data race because there is a *release-acquire* sequence between each pair of conflicting accesses. (b) has a race between $r_1(x)$ and $w_1(x)$.

3.1 Lazy Release Consistency

Lazy release consistency [11] is a variant of *eager* release consistency (ERC) [8], a relaxed memory consistency that allows the effects of shared memory accesses to be delayed until selected synchronization accesses occur. Simplifying matters somewhat, shared memory accesses are labeled either as *ordinary* or as *synchronization* accesses, with the latter category further divided into *acquire* and *release* accesses. Acquires and releases may be thought of as conventional synchronization operations on a lock, but other synchronization mechanisms can be mapped on to this model as well. Essentially, ERC requires ordinary shared memory accesses to be performed before the next release by the same processor. ERC implementations can delay the effects of shared memory accesses as long as they meet this constraint.

Under LRC protocols, processors further delay performing modifications remotely until subsequent acquires by other processors, *and* the modifications are only performed at the other processor that performed the acquire. The central intuition of LRC is that competing accesses to shared locations in correct programs will be separated by synchronization. By deferring coherence operations until synchronization is acquired, consistency information can be piggy-backed on existing synchronization messages.

To do so, LRC divides the execution of each process into *intervals*, each identified by an *interval index*. Figure 2 shows an execution of two processors, each of which has two intervals. The second interval of P_1, for example, is denoted σ_1^2.

Each time a process executes a release or an acquire, a new interval begins and the current interval index is incremented. We can relate intervals of different processes through a *happens-before-1* partial ordering similar to that defined above for shared accesses:

1. intervals on a single processor are totally ordered by program order,
2. interval $\sigma_p^i \xrightarrow{\text{hb1}}$ interval σ_q^j if σ_q^j begins with the acquire corresponding to the release that concluded interval σ_p^i, and

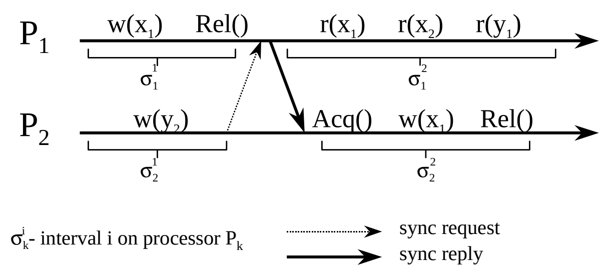

Figure 2. Shared data x_1 and x_2 are assumed to be on the same page, and y_1 and y_2 are co-located on another page. A page-based comparison of concurrent intervals would flag concurrent interval pairs σ_2^1-σ_1^2 and σ_1^2-σ_2^2 as containing possibly conflicting references to common pages. Comparison of the associated bitmaps would reveal that the former has only false sharing, while the latter has a true race in $r(x_1)$ from σ_1^2 and $w(x_1)$ from σ_2^2.

3. the transitive closure of the above.

LRC protocols append consistency information to all synchronization messages. This information consists of structures describing intervals seen by the releaser, together with enough information to reconstruct the $\xrightarrow{hb1}$ ordering on all visible intervals. For example, the message granting the lock to P_2 in Figure 2 contains information about all intervals seen by P_1 at the time of the release that had not yet been seen by P_2, i.e., σ_1^1. The system also records the fact that $\sigma_1^1 \xrightarrow{hb1} \sigma_2^2$.

While we discuss only locks and barriers in this paper, the notion of synchronization acquires and releases can be easily mapped to other synchronization models as well.

3.2 Data-Race Detection in an LRC System

Intuitively, a data race is a pair of accesses that do not have intervening synchronization, such that at least one of the accesses is a write. In Figure 2, the read of x_1 by P_1 and the write of x_1 by P_2 constitute a data race, because intervals σ_1^2 and σ_2^2 are concurrent (not ordered).

Detecting data races generally requires comparing each shared access against every other shared access. With an LRC system, as with any other system based on $\xrightarrow{hb1}$ partial ordering, we can limit comparisons only to accesses in pairs of concurrent intervals. For example, interval pair σ_1^1-σ_2^2 in Figure 2 is not concurrent, and so we do not have to check further in order to determine if there is a data race formed by accesses in those intervals. We only perform word-level comparisons if we have first verified that the pages accessed by the two intervals overlap.

For example, assume that y_1 and y_2 of Figure 2 reside on the same page. A comparison of pages accessed by concurrent intervals σ_1^2 and σ_2^1 would reveal that they access overlapping pages, i.e. the page containing y_1 and y_2. We would therefore need to perform a bitmap comparison in order to determine if the accesses constitute false sharing or true sharing (i.e., a data race). In this case, the answer would be false sharing because the accesses are to distinct locations. However, if P_2's first write were to z, a variable on a completely different page, our comparison of pages accessed by the two intervals would reveal no overlap. No bitmap comparison would be performed, even though the intervals are concurrent.

4 Implementation

4.1 System and its Changes

We implemented our race-detection system on top of CVM [13, 14], a software DSM that supports multiple protocols and consistency models. Like commercially available systems such as TreadMarks [12], CVM is written entirely as a user-level library and runs on most UNIX-like systems. Unlike TreadMarks, CVM was created specifically as a platform for protocol experimentation.

The system is written in C++, and opaque interfaces are strictly enforced between different functional units of the system whenever possible. The base system provides a set of classes that implement a generic protocol, lightweight threads, and network communication. The latter functionality consists of efficient, end-to-end protocols built on top of UDP.

New shared memory protocols are created by deriving classes from the base `Page` and `Protocol` classes. Only those methods that differ from the base class's methods need to be defined in the derived class. The underlying system calls protocol hooks before and after page faults, synchronization, and I/O events take place. Since many of the methods are inlined, the resulting system is able to perform within a few percent of a severely optimized system, TreadMarks, running a similar protocol. CVM was also designed to take advantage of generalized synchronization interfaces, as well as to use multi-threading for latency toleration. Our detection mechanism is based on CVM's multi-writer LRC protocol. This protocol propagates modifications in the form of *diffs*, which are run-length encodings of modifications to a single page [12]. Diffs are created through word-by-word comparisons of the current contents of a page with a copy of the page saved before any modifications were made.

We made only three modifications to the basic CVM implementation: (i) we added instrumentation to collect read and write access information, (ii) we added lists of pages read (*read notices*) to message types that already carry analogous information about pages written, and (iii) we potentially add an extra message round at barriers in order to retrieve word-level access information, if necessary.

4.2 Instrumentation

We use the ATOM [26] code-rewriter to instrument shared accesses with calls to analysis routines. ATOM allows executable binaries to be analyzed and modified. We use ATOM to identify and instrument all loads and stores that may access shared memory. Although ATOM is currently available only for DEC Alpha systems, a port is currently underway to Intel's x86 architecture. Moreover, tools that provide similar support for other architectures are becoming more common. Examples are EEL [16] (SPARC and MIPS), Shade [5] (SPARC), and Etch [22] (x86).

The base instrumentation consists of a procedure call to an analysis routine that checks if the instruction accesses shared memory. If so, the routine sets bits corresponding to the access's page and position in the page in order to indicate that the page and word have been accessed. The analysis routine consists of 30 instructions, including 10 instructions for saving registers to and restoring registers from the stack. The actual call to the analysis routine, together with instructions that save some registers outside the call, consume 7 or 8 more instructions. A small number of additional instructions are needed for the batching and runtime code modification optimizations.

Information about which pages were accessed, together with the bitmaps themselves, is placed in known locations for

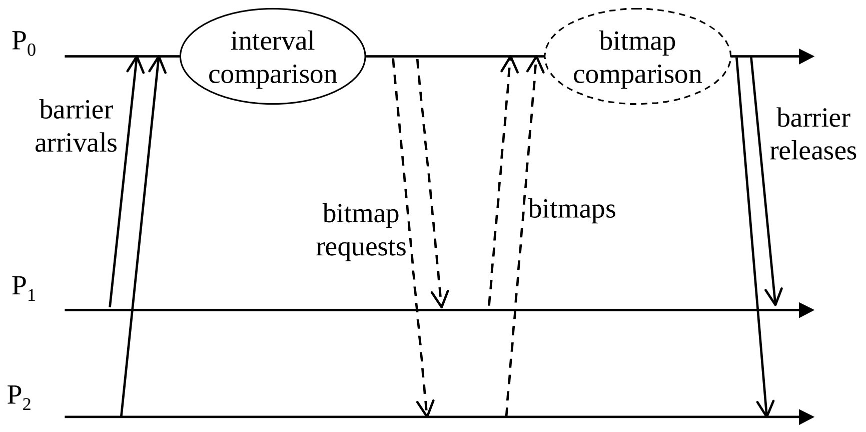

Figure 3. The barrier algorithm has the following steps: 1) read and write notices are sent to the barrier master, 2) the barrier master identifies concurrent intervals with overlapping page access lists, 3) bitmaps are requested for the overlapping pages from step 2, and 4) bitmap comparisons are used to identify data races. Dotted lines indicate events that occur only if sharing is detected in step 2.

CVM for use during the execution of the application. All data structures, including bitmaps, are statically allocated in order to reduce runtime cost. Shared memory and bitmaps are allocated at fixed memory locations in order to decrease the cost of the instrumentation code.

4.3 Algorithm

The overall procedure for detecting data races, illustrated in Figure 3, is the following:

1. We use ATOM to instrument all shared loads and stores in the application binary.

   The problem here lies in determining which references are shared and which are not. In the base case, we simply instrument any reference that does not use the stack pointer, the global variables pointer[1], or any register that has been loaded with one of these values in the current basic block. We also skip library references, as we know from inspection that our applications make no library calls that modify shared memory. In the absence of guarantees to the contrary, we can easily instrument non-CVM libraries as well. Such instrumentation would not have affect our slowdown because our applications spend time in libraries only during initialization. Section 4.5 describes several extensions to this basic technique that either eliminate more memory accesses as candidates for instrumentation, or decrease the cost of the resulting instrumentation.

2. Most synchronization messages in the base CVM protocol carry consistency information in the form of interval structures. Each interval structure contains one or more *write notices* that enumerate pages written during that interval. In CVM, we augmented these interval structures to also carry *read notices*, or lists of pages read during that interval.

[1]CVM assumes that all shared data is allocated dynamically.

Interval structures also contain version vectors that identify the logical time associated with the interval, and permit checks for concurrency.

3. Worker processes in any LRC system append interval structures (together with other consistency information) to barrier arrival messages. At each barrier, therefore, the barrier master has complete and current information on all intervals in the entire system. This information is sufficient for the barrier master to locally determine the set of all pairs of concurrent intervals. Although the algorithm must potentially compare the version vector of each interval of a given processor with the version vector of each interval of every other processor, exploiting synchronization and program order allows many of the comparisons to be omitted.

4. For each pair of concurrent intervals, the read and write notices are checked for overlap. A data race might exist on any page that is either written in two concurrent intervals, or read in one interval and written in the other. Such interval pairs, together with a list of overlapping pages, are placed on the *check list*.

   Steps 5 and 6 are performed only if this check list is non-empty, i.e., there is a data-race or there is false sharing (see Figure 3).

5. Barrier release messages are augmented to carry requests for bitmaps corresponding to accesses covered by the check list. Each read or write notice has a corresponding bitmap that describes precisely which words of the page were accessed during that interval. Hence, each pair of concurrent intervals has up to four bitmaps (read and write for each interval) that might be needed in order to detect races. These bitmaps are returned to the barrier master for each interval pair on the check list.

6. The barrier master compares bitmaps from overlapping pages in concurrent intervals. A single bitmap comparison is a constant time process, dependent only on page size. In the case of a read-write or write-write overlap, the algorithm has determined that a data race exists, and prints the address of the offending race.

We currently use a very simple interval comparison algorithm to find pairs of concurrent intervals, primarily because the major system overhead is elsewhere. The upper bound on the number of intervals per processor pair that the comparison algorithm must compare is $O(i^2)$, where i is the maximum number of intervals of a single processor since the last barrier. However, the algorithm needs only to examine intervals created during the last *barrier epoch*, where a barrier epoch is the interval of time between two successive barriers. By definition, these intervals are separated from intervals in previous epochs by synchronization, and are therefore ordered with respect to them. Since each interval potentially needs to be compared against every other interval of another process in the current epoch, the total comparison time per barrier is bounded by $O(i^2p^2)$, where p is the number of processes and i is the maximum number of intervals of any process in the current epoch.

In practice, however, the number of such comparisons is usually quite small. Applications that use only barriers have one interval per process per barrier epoch. More than one interval per barrier is only created through additional peer-to-peer synchronization, such as exclusive locks. However, peer-to-peer synchronization also imposes ordering on intervals of the synchronizing processes. For example, a lock release and subsequent acquire order intervals prior to the release with respect to those subsequent to the acquire. Since an ordered pair of intervals is by definition not concurrent, the same act that creates intervals also removes many interval pairs from consideration for data races. Hence, programs with many intervals between barriers usually also have ordering constraints that reduce the number of concurrent intervals.

Note that this technique does not require frequent barriers. We can accommodate long-running, barrier-free applications by forcing global synchronization to occur when system buffers are filled.

4.4 Example

We illustrate the use of this technique through an example based on Figure 2. Figure 2 shows a portion of the execution of two processes, together with their synchronization and memory accesses. Only memory accesses that were not statically identified as non-shared are shown. Further, data items x_1 and x_2 are located on the same page, and y_1 and y_2 are on another. If we assume that barriers occur immediately before and after the accesses in this figure, then the events of the figure correspond to a single barrier epoch. Barrier arrival messages from P_1 and P_2 will therefore contain information about four intervals: σ_1^1, σ_1^2, σ_2^1, and σ_2^2. Interval structures σ_1^1, σ_2^1, and σ_2^2 each contain a single write notice, while σ_1^2 contains two read notices. The reads of x_1 and x_2 are represented by a single read notice because they are located on the same page.

Upon the arrival of all processes at the second barrier, there are six possible interval pairs. We can eliminate σ_1^1-σ_1^2 and σ_2^1-σ_2^2 because of program order, and σ_1^1-σ_2^2 because of synchronization order. Finally, σ_1^1-σ_2^1 can be eliminated because these intervals access no pages in common.

This leaves σ_1^2-σ_2^1 and σ_1^2-σ_2^2 as possible causes of races or false sharing. In the following, we use the notation $\sigma_i^j r(x)$ to refer to the read bitmap of page x during interval σ_i^j and similar notation for writes. Barrier release messages will include requests for bitmaps $\sigma_1^2 r(y)$ and $\sigma_2^1 w(y)$ in order to judge the first pair, and $\sigma_1^2 r(x)$ and $\sigma_2^2 w(x)$ for the second pair.

The comparison of $\sigma_1^2 r(y)$ and $\sigma_2^1 w(y)$ will only reveal false sharing the intervals access different data items that just happen to be located on the same page. By contrast, the comparison of $\sigma_1^2 r(x)$ and $\sigma_2^2 w(x)$ will show that a data race exists because x_1 is accessed in both intervals, and one of the accesses is a write.

4.5 Optimizations

This section describes three enhancements to our basic technique.

4.5.1 Dataflow analysis

We use a limited form of iterative interprocedural register dataflow analysis in order to identify additional non-shared memory accesses. Our technique consists of creating a data-flow graph and associating incoming and outgoing sets of registers with each basic block. The registers in each set define registers known not to be pointers into shared memory. During each iteration of the analysis, the outgoing set is defined as the incoming set minus registers that are loaded in that block. Incoming sets are then redefined as the intersection of the incoming set for the previous iteration and the outgoing sets of all *preceding* blocks. The procedure continues until all incoming register sets stabilize.

Any values left in registers of the incoming register sets are known not to be pointers into shared space. Memory accesses using such registers do not need to be instrumented.

We made two main assumptions here. First, we simplify interprocedural analysis by exploiting the fact that function arguments are usually passed through registers. Tracking parameters that are not passed in this manner entails tracking the order of stack accesses in the caller and callee blocks. We conservatively assume that parameters passed by any other method might name pointers to shared data. Second, we assume that there are no function calls through register pointers. Such calls complicate data flow analysis because the destination of the calls can not be identified statically. The system could easily be modified to disable data-flow optimization when such calls are detected.

4.5.2 Batching

Calls to instrumentation routines can be *batched* by combining the accesses checks for multiple instructions into a single procedure call. We implemented three different types of batching for accesses within a single basic blocks:

- batching of accesses to the same memory location with the same reference type (either both load or both store)
- batching of accesses to the same memory location with different reference types
- batching of accesses to consecutive memory location with the same reference type

The largest performance improvement is provided by the first method, i.e, batching of accesses to the same memory location with the same reference type. Instrumentation for all but the first such access can be eliminated because we care only that the data *is* accessed, we do not care how many times it is accessed. Duplicated loads or stores to the same memory location might occur within a basic block because of register pressure or aliasing. An example of the latter case is a pair of loads through one register, sandwiched around a store through another register. The compiler's static analysis generally has no way of determining whether the loads and the store access the same, or distinct locations in memory. Hence, the second load is left in the basic block.

The other batching methods are less useful because no instrumentation is eliminated. However, the instrumentation that remains is less costly than without batching. Both the second and third methods avoid procedure calls by consolidating the instrumentation for multiple accesses into a single routine. The resulting instrumentation also needs to check whether the accesses are shared only once. This is valid even for accesses to consecutive memory locations because we assume that shared and non-shared regions are not located contiguously in the address space.

Instrumentation generated by the second method has the additional advantage of being able to use the same bitmap offset calculation for all accesses.

4.5.3 Runtime code modification

We use self-modifying code to remove instrumentation from instructions that turn out to reference private data. Memory reference instrumentation consists of a check to distinguish private and shared references, and code to record the access if it references shared memory. With runtime code modification, we overwrite the instrumentation with no-op instructions if the instruction references private data. The advantage is that subsequent executions of the instrumented instruction are delayed only by cost of executing no-op instructions, rather that the cost of executing instrumentation code that includes additional memory references.

Modifying code at runtime requires that the text segment be writable. We unprotect the entire text segment at the beginning of an application's execution using ATOM routines to obtain the size of the application's instrumented text segment. The

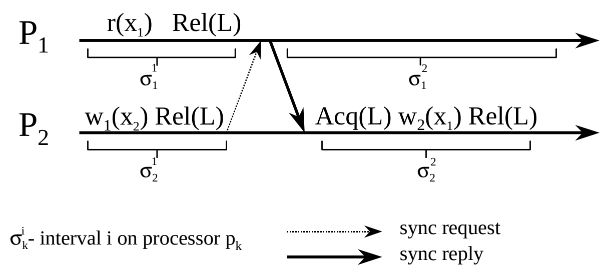

Figure 4. A single diff describes all modifications to page x from both σ_2^1 and σ_2^2 because of *lazy diffing.*

primary complication is caused by the separation of the data and instruction caches. We use data stores to overwrite instrumentation code. The new instructions are seen as data by the system and can be stalled in the (write-back) data cache. Problems remain even after the new instructions are written to memory, because stale copies might remain in the instruction cache. We solve this problem by issuing a special PAL IMB Alpha instruction that makes the caches coherent.

A second complication is that naively overwriting the entire instrumentation call *while inside the call* causes the stack to become corrupted. We get around this problem by merely saving an indication that the affected instrumentation calls should be deleted, rather than performing the deletion immediately. The instrumentation calls are actually deleted by code at subsequent synchronization points.

This technique is applicable only if we assume that each memory access instruction exclusively references either private or shared data. We modified our system to detect instructions that access both shared and non-shared data at runtime. This information is only anecdotal in that it provides no guarantees of behavior during other runs. Nonetheless, this technique can be useful if applied with caution. We used our modified version of CVM to detect instructions that access both shared and non-shared data in two of our applications. We eliminated the offending instructions by manually cloning [6] the routines that contained them.

4.5.4 Diffs

One optimization that we do not exploit is the use of *diffs* to capture write behavior. Diffs are summaries of changes made to a single page during an interval. They are created by comparing the current copy of a page with a *twin*, which is a copy saved before the modifications were begun. Hence, this diff seemingly has the same information as the write bitmaps, and use of the diffs could allow us to dispense with instrumentation of write accesses. However, diffs are created lazily, meaning that shared writes might be assigned to the wrong portion of a process's execution. For example, consider the process in Figure 4. We assume that x_1 and x_2 are on the same page. Lazy diff creation means that the diff describing P_2's first write is not immediately created at the end of interval σ_2^1. The problem is that the subsequent write to x_1 is then folded into the same diff, which is associated with the earlier interval. This merging does not violate consistency guarantees because LRC systems require applications to be free of data races. However, the merging will cause the system to incorrectly believe that a data-race exists between σ_1^1 and σ_2^1.

Another disadvantage of this approach is that the use of diffs would slightly weaken our race detection technique. Diffs contain only modifications to shared data. Locations that are overwritten with the same value do not appear in diffs, even

Apps	Input Set	Sync.	Memory (kbytes)	Intervals / Barrier	Slowdown (8 Proc)	Intervals Used	Bitmaps Used	Msg Ohead
Barnes		barrier	32768	1	2.42	4%	47%	11%
FFT	64 x 64 x 16	barrier	3088	1	1.36	4%	1%	<1%
SOR	1024x1024	barrier	32768	1	6.24	0%	0%	<1%
Spatial	512 mols, 5 iters	lock, barrier	824	16	5.93	86%	66%	26%
Water	512 mols, 5 iters	lock, barrier	344	84	3.18	5%	19%	31%

Table 1. Application Characteristics

though their use might constitute a race.

5 Performance

We evaluated the performance of our prototype by searching for data races in five common shared-memory applications: Barnes (Barnes-Hut algorithm the from Splash2 [27] benchmark suite) FFT (Fast Fourier Transform), SOR (Jacobi relaxation), Water (a molecular dynamics simulation; from the Splash2 suite, and Spatial (the same problem as Water, but different algorithm; also from Splash2, but optimized for reduced synchronization). All applications were run on DECstations with four 275 MHz Alpha processors, connected by a 155 MBit ATM. All performance numbers are measured on data-race free applications, i.e., we first detected, identified, and removed data-races from Water, Barnes, and Spatial, and then measured the numbers to be shown.

Table 1 summarizes the application inputs and runtime characteristics. “Memory size” is the size of the shared data segment. “Intervals / Barrier” is the average number of intervals created between barriers. As the number of interval comparisons is potentially proportional to the square of the number of intervals, this metric gives an approximate idea of the worst-case cost of running the comparison algorithm. Roughly speaking, a new interval is created for each synchronization acquire. Hence, barrier-only applications will have only a single interval per barrier epoch.

“Slowdown” is the runtime slowdown for each of the applications *without* any of the optimizations described in Section 4.5, compared with an uninstrumented version of the application running on an unaltered version of CVM. The first iteration of each application is not timed because we are interested in the steady-state behavior of long-running applications. However, slowdowns would be even smaller if the first iteration were counted. Over the five applications, non-optimized execution time slows only by an average factor of 3.8. This number compares quite favorably even with systems that exploit extensive compiler analysis [17, 7]. The last three columns are discussed in Section 5.2.

Figure 5 breaks down the application slowdown into five categories (again, without the optimizations described in Section 4.5). “CVM Mods” is the overhead added by the modifications to CVM, primarily setting up the data structures necessary for proper data-race detection and the additional bandwidth used by the read and write notices. “Bitmaps” describes the overhead of the extra barrier round required to retrieve bitmaps, together with the cost of the bitmap comparisons. “Intervals” refers to the time spent using the interval comparison algorithm to identify concurrent interval pairs with

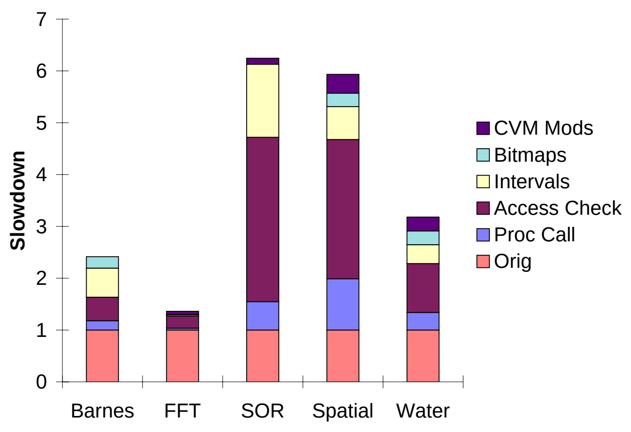

Figure 5. Breakdown of overhead for unoptimized instrumentation techniques.

overlapping page accesses. "Access" is the time spent inside the instrumentation's procedure calls determining whether accesses are to shared memory, and setting the proper bits if so. "Proc Call" is the procedure call overhead for our instrumentation. The base version of ATOM does not currently inline instrumentation; only procedure calls can be inserted into existing code. Section 5.5 describes the performance impact of an experimental version of ATOM that can inline instrumentation. "Orig" refers to the original running time.

Access check time dominates the overhead of the two applications that slow the most: SOR and Spatial. Neither application has significant false sharing, or frequent synchronization. There are therefore few interval creations, and few opportunities to run the interval comparison algorithm.

Barnes has the highest proportion of overhead spent in the interval comparison algorithm. The reason is that the process of determining whether a given pair of intervals access the same pages is expensive. Each Barnes process accesses a significant fraction of the entire shared address space during each interval. Our current representation of read and write notices as lists of pages is not efficient for large numbers of pages. This overhead could be reduced by changing the representation to bitmaps for intervals with many notices.

The following subsections describe the above overheads in more detail.

5.1 Instrumentation Costs

We instrumented each load and store that could potentially be involved in a data race. The instrumentation consists of a procedure call to an analysis routine, and hence adds "Proc Call" and "Access Check" overheads. By summing these columns from Figure 5, we can see that instrumentation accounts for an average of 64.6% of the total race-detection overhead.

This overhead can be reduced by instrumenting fewer instructions. This goal is difficult because shared and private data are all accessed using the same addressing modes, and sometimes even share the same base registers. However, we eliminate most stack accesses by checking for use of the stack pointer as a base register. The fact that all shared data in our system is dynamically allocated allows us to eliminate instructions that access data through the the "base register", which points to the start of the statically-allocated data segment.

Finally, we do not instrument any instructions in shared libraries because none of our applications pass segment pointers

App	Load and Store Instructions				
	Stack	Static	Library	CVM	Inst.
Barnes	558	320	118057	15759	933
FFT	308	207	118057	15759	358
SOR	100	83	61057	15759	210
Spatial	758	506	118057	15782	1043
Water	613	503	118057	15759	940

Table 2. Categorization of memory access instructions. "Inst" shows the number of instructions that are actually instrumented.

to any libraries. This is the case with the majority of the scientific programs where data race detection is the most important. We can, however, easily instrument "dirty" library functions, if necessary.

Table 2 breaks down load and store instructions into the categories that we are able to statically distinguish for the base case, i.e., without optimizations applied. The first five columns show the number of loads and stores that are not instrumented because they access the stack, statically-allocated data, or are in library routines, including CVM itself.

The sixth column shows the remainder. These instructions could not be eliminated, and are therefore possible data-race participants. We use ATOM to instrument each such access with a procedure call to an access check routine that is executed whenever the instruction is executed.

On average, we are able to statically determine that over 99% of the loads and stores in our applications are to non-shared data. As an example, the FFT binary contains 134993 load and store instructions. Of these, 118057 instructions are in libraries. A further 308 instructions access data through the stack pointer, and hence reference stack data. Another 15759 are in the CVM system itself. Finally, 207 instructions access data through the global pointer, a register pointing to the base of statically allocated global memory. We can eliminate these instructions as well, since CVM allocates all shared memory dynamically. In the entire binary, there remain only 358 memory access instructions that could possibly reference shared memory, and hence might be a part of a data race.

Nonetheless, Section 5.3 will show that the majority of run-time calls to our analysis routines are for private, not shared, data.

5.2 The Cost of the Comparison Algorithm

The comparison algorithm has three tasks. First, the set of concurrent interval pairs must be found. Second, this list must be reduced to those interval pairs that access at least one page in common (e.g., one interval has a read notice for page x and the other interval has a write notice for page x). Each such pair of concurrent intervals exhibits unsynchronized sharing. However, the sharing may be either false sharing, i.e., the loads and stores to the page x reference different locations in x (not a data race), or true sharing, when the loads and stores reference at least one common location at the page x (data race).

The column labeled "Intervals Used" in Table 1 shows the percentage of intervals that are involved in at least one such concurrent interval pair. This number ranges from zero for SOR, where there is no unsynchronized sharing (true or false), to 86% for Spatial, where there is a large amount of both true and false sharing. Note that the number of possible interval pairs is quadratic with respect to the number of intervals, so even if this stage eliminates only 14% of all intervals, as we do for Spatial, we may be eliminating a much higher percentage of interval pairs.

The column labeled "Bitmaps Used" shows that an average of only 27% of all bitmaps must be retrieved from constituent processors in order to identify data races by distinguishing false from true sharing. As page access lists of concurrent intervals will only overlap in cases of false sharing or actual data races, the percentage of intervals and bitmaps involved in comparisons is fairly small.

Note, however, the effect of bitmap and interval comparisons on Barnes. Although the absolute amount of overhead added by the comparisons is not large, it is larger relative to the rest of the overhead than for any other application. There is also a seeming disparity between the utilization of intervals and bitmaps for Barnes. Only 4% of intervals are used, but 47% of the bitmaps are used. This implies that the majority of the shared pages are accessed in only a small number of intervals, probably one or two phases of a timestep loop that has many phases.

In fact, this is exactly the case for our version of Barnes. Most of the work is done in the force and position computation phases, which are separated by a barrier. However, each processor accesses the same bodies during both phases, and the sets of bodies accessed by different processors are disjoint. Hence, there is no true sharing between processors across the barrier for these computations. Since the bodies assigned to each processor during any iteration are scattered throughout the address space, a large amount of false sharing occurs. The barrier serves to double the effect of the false sharing by splitting each of the intervals in half, causing bitmaps to be requested twice for each page instead of once. Note that the barrier can not be removed without a slight reorganization of code because it synchronizes updates to a few scalar global variables.

We tested this interpretation of the results by implementing the above reorganization. Removal of the barrier effectively reduced the interval and bitmap overheads in half, reducing the overall overhead by approximately 30%.

The final column of Table 1 shows the amount of additional data needed by the race detection technique compared with the uninstrumented system.

5.3 The Effect of optimizations

Table 3 shows the effect of our optimizations on the number of instructions actually instrumented. "No Opt." refers to the base case with no optimizations, "DF" is dataflow, "Batching" is self-explanatory, "Code Mod" refers to dynamic code modification, and "All" includes all three. "DF+Batching" is included to show the synergy between dataflow and batching in the absence of code modification. Code modification actually *increases* the number of instrumented sites in FFT and Water because of cloning.

The last three columns of Table 3 show the number of times we were able to apply each batching method. Here, "2-3" represents combining two or three instructions of the same type with consecutive addresses. "Same" represents combining instructions of the same type and address. Finally, "Mix" is combining instructions with the same address, but different access types. These numbers imply that batching is most applicable for applications with complex data structures and access patterns.

Apps	Instrumented Instructions						Batching		
	No Opt.	Data Flow	Batching	Code Mod	DF+Batching	All	2-3	Same	Mix
Barnes	933	854	730	933	655	**655**	63	26	76
FFT	358	280	337	444	276	**355**	1	7	1
SOR	210	199	189	210	179	**179**	9	0	3
Spatial	1043	844	904	1043	725	**725**	39	41	32
Water	940	776	747	1156	604	**733**	39	38	58

Table 3. Static instrumentation statistics.

Apps	Millions of Instrumented Instructions					
	No Opt	Data Flow	Batching	Code Mod	DF+Batching	All
Barnes	435.8	415.0	434.3	88.5	413.5	**87.0**
FFT	5.8	5.3	5.8	1.5	5.3	**1.5**
SOR	38.5	38.5	38.5	27.7	38.5	**27.7**
Spatial	108.3	39.5	88.2	28.5	22.1	**20.0**
Water	124.9	51.8	105.0	21.6	32.6	**7.5**

Table 4. Dynamic Optimization Statistics

Table 4 shows the effect of our optimizations on the number of instrumented instructions executed at runtime. Although data flow analysis and batching together eliminate 29.8% and 22.9% of local reference instrumentations for Barnes and FFT, respectively, this accounts for only 5.1% and 8.8% of instrumented references at runtime. On the other hand, only 30.5% of Spatial's instrumentations and 35.8% of Water's instrumentations are eliminated. Yet the elimination of these instrumentations accounts for 80.0% and 73.9% of runtime references, respectively. Clearly the effectiveness of these optimizations is heavily application-dependent.

Figure 6 shows the effect of these optimizations on the overall slowdown of the applications. The average slowdown when all three optimizations are applied is 2.8, which is an improvement of 26%. FFT has the lowest overhead at 22%. Figure 6 also includes bars for the inlining optimization discussed in Section 5.5.

5.4 The Cost of CVM Modifications

Figure 5 shows that almost 15.8% of our overhead comes from "CVM Mods", or modifications made to the CVM system in order to support the race-detection algorithm. This overhead consists of the cost of setting up additional data structures for data-race detection and the cost of the additional bandwidth consumed by read notices.

The last column of Table 1 shows the bandwidth overhead of adding read and additional write notices to synchronization messages. Individual read and write notices are the same size, but there are typically at least five times as many reads as writes, and read notices consume a proportionally larger amount of space than write notices. Additional write notices are needed because even the notices are no longer created lazily, even though diffs still are.

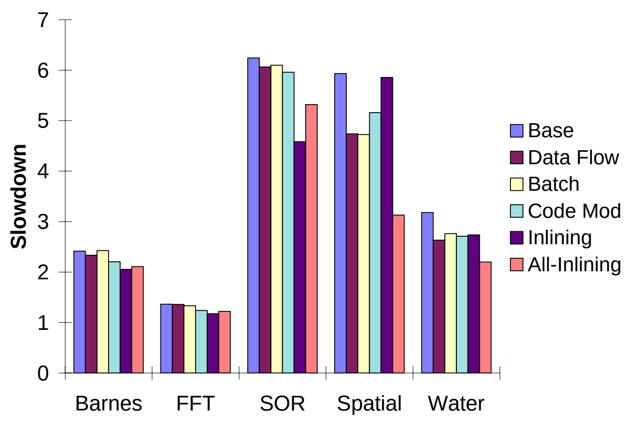

Figure 6. Optimizations

The bandwidth overhead for Water is quite large because the fine-grained synchronization means that many intervals and notices are created. By contrast, the primary cost for Spatial is that false sharing is quite prevalent, leading to a large number of bitmap requests.

5.5 Inlining

To verify our assumption that the procedure call and access check overhead can be significantly reduced by inlining, we used an unreleased version of ATOM, called XATOM, to inline read and write access checks. Our implementation decreases the cost of the inlined code fragments by using register liveness analysis to identify dead registers. Dead registers are used whenever possible to avoid spilling the contents of registers needed by the instrumentation code.

Table 5 shows the effect of inlining on overall performance, relative to the base case with no optimizations. The column labeled "Runtime" shows the effect as a percentage of overall running time, while "Overhead" shows the same quantity as a percentage of instrumentation overhead. The "Static" column shows the percent of inlined instructions eliminated by register liveness analysis (mostly load and store instructions), and "Dynamic" shows the corresponding dynamic quantity. Elimination of these instructions is very useful because the majority are memory access instructions, and hence relatively expensive.

The improvements roughly correlate with the procedure call overhead shown in Figure 5, where an average of 13.7% of total overhead is caused by procedure calls. However, inlining can also eliminate some of the access check overhead because the liveness analysis can reduce register spillage. This is particularly important in the case of SOR, where most of the overhead is in access checks.

An important question that we have not answered is how effective the other optimizations are in combination with inlining. Inlining certainly decreases the potential of the other techniques because all of them work by decreasing the cost or number of access checks. However, they should still be effective in combination with inlining because the remaining overhead is still significant. Runtime code modification, in particular, would still be useful because inlining has no effect on the total number of instructions instrumented. However, the code modification mechanism would probably need to change slightly in order to address the fact that the inserted instrumentation is no longer just a few bytes. Inserting unconditional branches to the end of

Apps	Improvement		Register Liveness	
	Runtime	Overhead	Static	Dynamic
Barnes	15.0%	28.5%	21.7%	35.5%
FFT	13.8%	23.5%	16.8%	15.8%
SOR	26.6%	32.8%	10.8%	9.0%
Spatial	1.3%	1.9%	13.5%	7.3%
Water	14.0%	25.7%	17.8%	24.4%

Table 5. Inlining

the instrumentation code might be more effective than overwriting the inlined instrumentation with a large number of no-op instructions.

6 Discussion

6.1 Reference Identification

The system currently prints the shared segment address together with the interval indexes for each detected race condition. In combination with symbol tables, this information can be used to identify the exact variable and synchronization context.

Identifying the specific instructions involved in a race is more difficult because it requires retaining program counter information for shared accesses. This information is available at runtime, but such a scheme would require saving program counters for each shared access until a future barrier analysis phase determined that the access was not involved in a race. The storage requirements would generally be prohibitive, and would also add runtime overhead.

A second approach is to use the conflicting address and corresponding barrier epoch from an initial run of the program as input to a second run. During the second run, program counter information can be gathered for only those accesses to the conflicted address that originate in the barrier epoch determined to involve the data race.

While runtime overhead and storage requirements can thereby be drastically reduced, the data race must occur in the second run exactly as in the first. This will happen if the application has no *general races* [20], i.e., synchronization order is deterministic. This is not the case in Water, the application for which we found data races. A solution is to modify CVM so as to save synchronization ordering information from the first run, and to enforce the same ordering in the second run. This is done in the work on execution replay in TreadMarks, a similar DSM. The approach of the Reconstruction of Lamport Timestamps (ROLT) [23] technique keeps track of minimal ordering information saved during an initial run to enforce exactly the same interleaving of shared accesses and synchronization in a second run. During the second run, a complete address trace can be saved for post-mortem analysis, although the authors do not discuss race detection in detail. The advantage of this approach is that the initial run incurs minimal overhead, ensuring that the tracing mechanism does not perturb the normal interleaving of shared accesses.

The ROLT approach is complementary to the techniques described in this paper. Our system could be augmented to include an initial synchronization-tracing phase, allowing us to eliminate our perturbation of the parallel computation in the second, i.e., race-reference identification phase.

Currently, we use the shared page of the variable involved in the data-race from the initial run as the target page for which we save program counters during the identification run.

6.2 Global Synchronization

The interval comparison algorithm is run only at global synchronization operations, i.e., barriers. The applications and input sets in this study use barriers frequently enough, or otherwise synchronize infrequently enough, that the number of intervals to be compared at barriers is quite manageable. Nonetheless, there certainly exist applications for which global synchronization is not frequent enough to keep the number of interval comparisons to a small number. Ideally, the system would be able to incrementally discard data races without global cooperation, but such mechanisms would increase the complexity of the underlying consistency protocol [10]. If global synchronization is either not used, or not used often enough, we can exploit CVM routines that allow global state to be consolidated between synchronizations. Currently, this mechanism is only used in CVM for garbage collection of consistency information in long-running, barrier-free programs.

6.3 Accuracy

Adve [2] discusses three potential problems in the accuracy of race detection schemes in concert with *weak memory* systems, or systems that support memory models such as lazy release consistency.

The first is whether to return all data races, or only “first” data races [18, 2]. First races are essentially those that are not caused or affected by any prior race. Determining whether a given race is affected by any other effectively consists of deciding whether the operations of any other race precede (via $\xrightarrow{hb1}$) the operations of the race in question. Our system currently reports all data races. However, we could easily capture an approximation of first races by turning off reporting of any races for which both accesses occur after (via $\xrightarrow{hb1}$) the accesses of other data races.

The second problem with the accuracy of dynamic race-detection algorithms is the reliability of information in the presence of races. Race conditions could cause wild accesses to random memory locations, potentially corrupting interval ordering information or access bitmaps. This problem exists in any dynamic race-detection algorithm, but we expect it to occur infrequently.

A final accuracy problem identified by Adve is that of systems that attempt to minimize space overhead by buffering only limited trace information, possibly resulting in some races remaining undetected. Our system only discards trace information when it has been checked for races, and hence does not suffer this limitation.

6.4 Limitations

We expect this technique to be applicable for a large class of applications. The applications in our test suite range from SOR, which is bandwidth-limited, to Water, which both synchronizes and modifies data at a fine granularity. These applications stress different portions of the race-detection technique: the raw cost of access instrumentation for SOR, and the complexity and cost of dealing with large numbers of intervals for Water.

> Furthermore, our applications (with the exception of SOR), have not been > modified in order to reduce false sharing. > Multi-writer LRC tolerates false sharing much better than most other > protocols. > Applications that have been tuned for LRC tend not to have false sharing > removed. > Water, Spatial, and Barnes all have large amounts of false sharing.

Nonetheless, our methodology is clearly not applicable in all situations. Chaotic algorithms, for example, tolerate races as a means of eliminating synchronization and improving performance. The false positives caused by tolerated races can obscure unintended races, rendering this class of applications ill-suited for our techniques.

Similarly, protocol-specific optimization techniques may cause spurious races to be reported. An earlier version of Barnes had a barrier that enforced only anti-dependences. This barrier has been removed in our version of Barnes. The application is still correct because LRC delays the propagation of both consistency information and data. However, these anti-dependences are flagged by our system as data races, and can interfere with the normal operation of our technique.

The techniques that we discuss in this paper are not necessarily limited to LRC systems and applications. Our approach is essentially to use existing synchronization-ordering information to reduce the number of comparisons that have to be made at runtime. This information could easily be collected in other distributed systems and memory models by putting appropriate *wrappers* around synchronization calls, and appending a small amount of additional information to synchronization messages. Multiprocessors could be supported by using wrappers and appending a small amount of data to synchronization state. Application of the rest of the techniques should be straightforward.

6.5 Further Performance Enhancements

Performance of the underlying protocol could be improved by using our write instrumentation to create diffs, rather than using the twin and page comparison method. We did not investigate this option because of its complexity. Integrating this mechanism with the runtime code modification, for example, would be non-trivial.

Compiler techniques could be used to expose more opportunities for batching. Loop unrolling and trace scheduling would be particularly effective for applications such as SOR, the application with the largest overhead in our application testbed.

Finally, the interval comparison algorithm could be improved significantly. While the overhead added by the comparison algorithm was relatively small for our applications, better worst-case bounds would be desirable for a production system. One promising approach is the use of hierarchical comparison algorithms. For example, if two processes create a large number of intervals through exclusively pairwise synchronization, the number of intervals to be compared with other processes could be reduced by first aggregating the intervals that were created in isolation, and then using these aggregations to compare with intervals of other processes.

7 Related Work

There has been a great deal of published work in the area of data race detection. However, most prior work has dealt with applications and systems in more specialized domains. Bitmaps have been used to track shared accesses before [7], but we know of no other language independent implementation of on-the-fly data-race detection for explicitly-parallel, shared-memory programs.

We previously [21] described the performance of a preliminary form of our race-detection scheme that ran on top of CVM's single-writer LRC protocol [14]. This paper describes the performance of our race-detection scheme on top of CVM's multi-writer protocol. This protocol is a more challenging target because it usually outperforms the single-writer protocol significantly, making it more difficult to hide the race-detection overheads. Additionally, the work described in this paper includes several optimizations to the basic system (i.e., batching, data-flow analysis, runtime code modification, and inlining).

Our work is closely related to work already alluded to in Section 6.3, a technique described (but not implemented) by Adve et al. [2]. The authors describe a post-mortem technique that creates trace logs containing synchronization events, information allowing their relative execution order to be derived, and computation events. Computation events correspond roughly to CVM's intervals. Computation events also have READ and WRITE attributes that are analogous to the read and write page lists and bitmaps that describe the shared accesses of an interval. These trace files are used off-line to perform essentially the same operations as in our system. We differ in that our minimally-modified system leverages off of the LRC memory model in order to abstract this synchronization ordering information *on-the-fly*. We are therefore able to perform all of the analysis on-the-fly as well, and do away with trace logs, post-mortem analysis, and much of the overhead.

Work on execution replay in TreadMarks could be used to implement race-detection schemes. The approach of the Reconstruction of Lamport Timestamps (ROLT) [23] technique is similar to the technique we described in Section 6.1 for identifying the instructions involved in races. Minimal ordering information saved during an initial run is used to enforce exactly the same interleaving of shared accesses and synchronization in the second run. During the second run, a complete address trace can be saved for post-mortem analysis, although the authors do not discuss race detection in detail. The advantage of this approach is that the initial run incurs minimal overhead, ensuring that the tracing mechanism does not perturb the normal interleaving of shared accesses.

The ROLT approach is complementary to the techniques described in this paper. The primary thrust of our work is in using the underlying consistency mechanism to prune enough information *on-the-fly* so that post-mortem analysis is not necessary. As such, our techniques could be used to improve the performance of the second phase of the ROLT approach. Similarly, our system could be augmented to include an initial synchronization-tracing phase, allowing us to reduce perturbations of the parallel computation.

Recently, work on Eraser [25] used verification of lock discipline to detect races in multi-threaded programs. Eraser's main advantage is that it can detect races that do not actually occur in the instrumented execution. However, it does not guarantee race-free behavior if no data-races are found, and can return false positives. Furthermore, the system does not support distributed execution. Finally, the overhead of Eraser's approach is an order of magnitude higher than ours.

Work on detecting data-race detection for non-distributed multi-threaded programs has also been done for RecPlay [24], a Record/Replay system for multi-threaded programs. This work is similar to ROLT approach discussed here, but applied to multi-threaded programs. This work uses the happens-before relation to reconstruct and replay the execution of the initial run, and then in the second run perform access checks.

8 Conclusions

This paper has presented the design and performance of a new methodology for detecting data races in explicitly-parallel, shared-memory programs. Our technique abstracts synchronization ordering from consistency information already maintained by multiple-writer lazy-release-consistent DSM systems. We are able to use this information to eliminate most access comparisons, and to perform the entire data-race detection on-the-fly.

We used our system to analyze five shared-memory programs, finding data races in three of them. Two of those data races, in standard benchmark programs, were bugs.

The primary costs of data-race detection in our system are in tracking shared data accesses. We were able to significantly reduce these costs by using three optimization techniques: register data-flow analysis, batching, and inlining. Nonetheless, the majority of the runtime calls to our library are for non-shared accesses. We therefore used runtime code-modification to dynamically rewrite our instrumentation in order to eliminate access checks for instructions that accessed only non-shared data. By combining all of the optimizations except inlining, we were able to reduce the average slowdown for our applications to approximately 2.8, and to only 1.2 for one application. We expect that combining inlining with the other optimizations would reduce the slowdown even further.

While the implementation described above is specific to LRC, our general approach is not. Our system exploits synchronization ordering to eliminate the majority of shared accesses without explicit comparison. Fine-grained comparisons are made only where the coarse-grained comparisons fail to rule data races out. This approach could be used on systems supporting other programming models by using “wrappers” around synchronization accesses to track synchronization ordering.

We believe that the utility of our techniques, in combination with the generality of the approach that we present, can help data-race detection to become more widely used.

References

[1] S. V. Adve and M. D. Hill. A unified formalization of four shared-memory models. *IEEE Transactions on Parallel and Distributed Systems,* 4(6):613–624, June 1993.

[2] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer. Detecting data races on weak memory systems. In *Proceedings of the 18th Annual International Symposium on Computer Architecture,* pages 234–243, May 1991.

[3] T. R. Allen and D. A. Padua. Debugging fortran on a shared memory machine. In *International Conference on Parallel Processing,* pages 721–727, August 1987.

[4] J. Choi and S. L. Min. Race frontier: Reproducing data races in parallel program debugging. In *Proceedings of the 1991 Conference on the Principles and Practice of Parallel Programming*, April 1991.

[5] Robert F. Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. Technical Report TR-93-12, Sun Microsystems Lab, July 1993.

[6] K. Cooper, M. W. Hall, and K. Kennedy. A methodology for procedure cloning. *Computer Languages,* 19(2):105–117, February 1993.

[7] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Proceedings of the 1990 Conference on the Principles and Practice of Parallel Programming*, pages 1–10, March 1990.

[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] Robert Hood, Ken Kennedy, and John Mellor-Crummey. Parallel program debugging with on-the-fly anomaly detection. In *Proceedings Supercomputing '90*, pages 15–26, May 1990.

[10] P. Keleher. *Distributed Shared Memory Using Lazy Release Consistency*. PhD thesis, Rice University, 1994.

[11] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, May 1992.

[12] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the 1994 Winter Usenix Conference*, pages 115–131, January 1994.

[13] Pete Keleher. The Coherent Virtual Machine. Technical Report Maryland TR93-215, Department of Computer Science, University of Maryland, September 1995.

[14] Pete Keleher. The relative importance of concurrent writers and weak consistency models. To appear in *The Proceedings of the 16th International Conference on Distributed Computing Systems*, May 1996.

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

[16] James R. Larus and Eric Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, June 1995.

[17] John Mellor-Crummey. Compile-time support for efficient data race detection in shared-memory parallel programs. Technical Report CRPC-TR92232, Rice University, September 1992.

[18] R. H. B. Netzer and B. P. Miller. Improving the accuracy of data race detection. In *Proceedings of the 1991 Conference on the Principles and Practice of Parallel Programming*, April 1991.

[19] Robert H. B. Netzer and Barton P. Miller. On the complexity of event ordering for shared-memory parallel program executions. In *1990 International Conference on Parallel Processing*, pages 93–97, August 1990.

[20] Robert H. B. Netzer and Barton P. Miller. What are race conditions? In *ACM Letters on Programming Languages and Systems*. ACM, March 1992.

[21] Dejan Perković and Peter J. Keleher. Online data-race detection via coherency guarantees. In *Proceedings of the Second USENIX Symposium on Operating System Design and Implementation(OSDI'96)*, pages 47–58, October 1996.

[22] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, and Brian Bershad. Instrumentation and optimization of win32/intel executables using etch. In *USENIX Windows NT Workshop*, 1997.

[23] M. A. Ronsse and W. Zwaenepoel. Execution replay for TreadMarks. Submitted for publication, 1996.

[24] Michiel Ronsse and Koen De Bosschere. Work in progress: An on-the-fly data race detector for recplay, a record/replay system for parallel programs. In *16th ACM Symposium on Operating Systems Principles (Work in progress)*, October 1997.

[25] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multi-threaded programs. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997.

[26] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, May 1994.

[27] 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 Annual International Symposium on Computer Architecture*, pages 24–37, June 1995.