In The Second Symposium on Operating Systems Design and Implementation (OSDI '96), 47-57, October 1996.

Online Data-Race Detection via Coherency Guarantees

Dejan Perković and Peter J. Keleher
Department of Computer Science
University of Maryland
College Park, MD 20742-3255
(keleher|dejanp)@cs.umd.edu

Abstract

We present the design and evaluation of an on-the-fly data-race-detection technique that handles applications written for the lazy release consistent (LRC) shared memory model. We require no explicit association between synchronization and shared memory. Hence, shared accesses have to be tracked and compared at the minimum granularity of data accesses, which is typically a single word.

The novel aspect of this system is that we are able to leverage information used to support the underlying memory abstraction to perform on-the-fly data-race detection, without *compiler support. Our system consists of a minimally modified version of the CVM distributed shared memory system, and instrumentation code inserted by the ATOM code re-writer.*

We present an experimental evaluation of our technique by using our system to look for data races in four unaltered programs. Our system correctly found read-write data races in a program that allows unsynchronized read access to a global tour bound, and a write-write race in a program from a standard benchmark suite. Overall, our mechanism reduced program performance by approximately a factor of two.

1 Introduction

While potentially very useful, data-race detection mechanisms have yet to become widespread. Part of the problem is surely the restricted domain in which most such mechanisms operate, i.e. parallelizing compilers. Such restrictions are deemed necessary because data-race detection is generally NP-complete [17], and exponential searches over a domain the size of the number of shared accesses in a program execution are prohibitively expensive.

This paper presents the design and evaluation of an online data-race detection technique for explicitly parallel shared-memory applications. This technique is applicable for shared memory programs written for the lazy-release-consistent (LRC) [9] memory model. Our work differs from previous work [3, 4, 5, 7, 16, 15] 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 [17].

Our general approach is to run applications on a modified version of the Coherent Virtual Memory (CVM) [11, 12] system, a distributed shared memory (DSM) system that supports LRC. 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.

Hence, a DSM that implements LRC can perform the entire process on-the-fly with acceptable overhead.

Modifying CVM to implement data-race detection consisted of (i) adding instrumentation to detect read accesses, (ii) integrating this information into existing CVM structures that already contain analogous information about write accesses, and (iii) running a simple race-detection algorithm at existing global synchronization points. The task of this last point is made much easier by leveraging off of ordering information already maintained to support consistency guarantees.

We used this technique to check for data races in implementations of four common parallel applications. Our system correctly found races in two. TSP, a program that solves the Traveling Salesman Problem, has a large number of data races that result from unsynchronized read accesses to a global tour bound. The reads are left unsynchronized to improve performance; out-of-date tour bounds may cause redundant work to be performed, but do not violate correctness. Water-Nsquared, of the Splash2 [22] benchmark suite, had a data race that constituted a real bug. This bug has been reported to the Splash authors and fixed in their current version.

While overhead is still potentially exponential, we describe a variety of techniques that greatly reduce the number of comparisons that have to be made. Those portions of the race-detection procedure that have the largest theoretical complexity are only the third or fourth-most expensive portion of the overall technique for the applications that we studied. Specifically, we show that i) we can statically eliminate over 99% of all load and store instructions as potential race participants, ii) we dynamically eliminate over 70% of all program execution from consideration by using LRC ordering information, and iii) the slowdown from using data-race detection in our system is approximately a factor of two for the applications studied. While this overhead is clearly too high for the system to be used all of the time, it is low enough for use when traditional debugging techniques are insufficient.

2 Problem Definition

The goal of this work is to create a system that detects race conditions online. Since our strategy relies on LRC consistency, our system is clearly applicable only for applications that will run properly on release-consistent systems, i.e. properly-labeled [6] or DRF1 [1] applications. The following definitions are assumed throughout the rest of the paper.

Definition 1 *A* `data race` *is defined as a pair of memory accesses in some execution, such that:*

1. *Both access the same shared variable,*
2. *At least one is a write,*
3. *The accesses are not ordered by system-visible synchronization or program order.*

In the sense discussed by Netzer [18], the races found by our system are *actual* data races, i.e. they are races that occur while the program is running on our system. In common with most other implemented systems, both with and without compiler support, we make no claim to detect all *feasible* data races, i.e. all data races allowed by the semantics of the program. 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. In practice, however, we expect most data races to reveal themselves when given an appropriate input set.

Figure 1 shows a portion of a single execution in which two processes access shared variable x, and synchronize through synchronization variable L. Both w_1-r_1 and w_1-r_2 are *feasible* data races, assuming we have no knowledge of the value of `flag`. However, if `flag` is equal to zero during an execution, w_1-r_2 is the only *actual* data race, the only data race that occurs and would therefore be caught by our system. The access pair w_1-r_2 is still potentially a bug for some other execution, but would not be flagged by CVM during this execution. Shared accesses w_1 and r_3 do not constitute a data race, as they are ordered by P_1's unlock and P_2's lock.

```
   P1               P2

Lock(L)
w1(x)
Unlock(L)         if (flag) {
                      r1(x)
                  }
                  r2(x)
                  Lock(L)
                  r3(x)
                  Unlock(L)
```

Figure 1. Race Conditions

In order for our system to distinguish between w_1-r_1 and w_1-r_3 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.

Given the above definition for data races, our system will detect all data races that occur during a given execution.

3 Lazy Release Consistency and Data Races

3.1 Lazy Release Consistency

Lazy release consistency [9] is a variant of *eager* release consistency (ERC) [6], 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

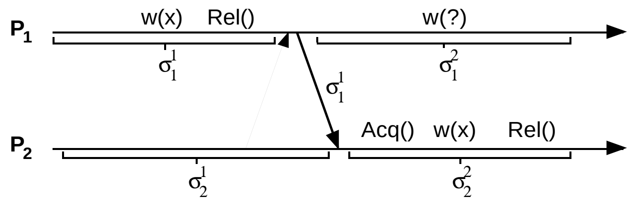

Figure 2. Process Intervals

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 only when a subsequent release by the same processor is performed. 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 piggybacked on existing synchronization messages.

To do so, LRC divides the execution of each process into *intervals*, each identified by an *interval index*. For example, Figure 2 shows an execution of two processors, each of which have two intervals. Interval 1 of P_1, denoted σ_1^1, contains a release synchronization access and a write to shared variable x. Each time a process executes a release or an acquire, a new interval begins and the current interval index is incremented. Intervals of different processes are related by a *happens-before-1* partial ordering [1]:

1. intervals on a single processor are totally ordered by program order,
2. interval σ_p^i precedes interval σ_q^j if σ_q^j begins with the acquire corresponding to the release that concluded interval σ_p^i, and
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 but not the acquirer. 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.

3.2 Data Race Detection in an LRC System

The *happens-before-1* relation orders intervals, and by implication, accesses within intervals. Since *happens-before-1* is a combination of synchronization order (the release by P_1 precedes the acquire by P_2), and program order, it is clear that the write to x in σ_1^1 of Figure 2 precedes (via the *happens-before-1* relation) the write in σ_2^2 (interval 2 of P_2).

We can now re-define Definition 1 as follows:

Definition 2 *A* `data race` *is defined as a pair of memory accesses in some execution, such that:*

1. *Both access the same shared variable,*
2. *At least one is a write,*
3. *The accesses are not ordered with respect to* happens-before-1.

More informally, 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, if the second write of P_1 were to variable x, it would constitute a data race with the access in σ_2^2, because intervals σ_1^2 and σ_2^2 are concurrent (not ordered).

In general, detecting data races requires comparing each access against every other access. With an LRC system, however, 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 (among others), and so we do not have to check further in order to determine if there is a data race formed by accesses of these intervals. Furthermore, for each concurrent interval pair, we only perform word-level comparisons if we have first verified that the pages accessed by the two intervals overlap.

For example, assume that the second write by P_1 in Figure 2 is to a variable y that is located on the same page as x. A comparison of pages accessed by concurrent intervals σ_1^2 and σ_2^2 would reveal that they access overlapping pages, and hence we would 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 different locations. However, if P_1's second 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

We implemented our data-race detection on top of CVM [11, 12], a software DSM that supports multiple protocols and consistency models. Like commercially available

	Input Set	Synchronization	Memory Size (kbytes)	Intervals Per Barrier	Slowdown (8 Proc)
FFT	64 x 64 x 16	barrier	3088	2	2.08
SOR	512x512	barrier	8208	2	1.83
TSP	19 cities	lock	792	177	2.51
Water	216 mols, 5 iters	lock, barrier	152	46	2.31

Table 1. Application Characteristics

systems such as TreadMarks [10], 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. However, CVM was designed to take advantage of generalized synchronization interfaces, as well as to use multi-threading for latency toleration. We therefore expect the performance of the fully functional system to improve over the existing base. In order to simplify the comparison process, however, we do not use either of these techniques in this study.

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 added an extra message round at barriers in order to retrieve word-level access information, if necessary.

We use the ATOM [21] 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 available only for DEC Alpha systems, similar tools are becoming more common. EEL [13] provides similar support for Sparc and MIPS systems, and several machine vendors are working on such tools as well.

The actual instrumentation consists of a procedure call to an analysis routine that sets a bit in a per-page bitmap if the instruction accesses shared memory. Information about which pages were accessed, together with the bitmaps themselves, is placed in known locations for CVM to use during the execution of the application. All data structures, including bitmaps, are statically allocated in order to reduce runtime cost.

The overall procedure for detecting data races is the following:

1. CVM synchronization messages carry information about process intervals. Each interval contains one or more *write notices* that specify pages written during that interval. We augmented interval structures to also carry *read notices*, or lists of pages read during that interval. Interval structures also contain version vectors that identify the logical time associated with the interval, and permit checks for concurrency.

2. Worker processes in any LRC system append consistency information describing all local intervals 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 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 vector of each interval of every other processor, synchronization and program order allow many of the comparisons to be bypassed. Version vector comparison is a constant time process, requiring only two integer comparisons.

3. 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*.

4. Barrier release messages carry the check list to all system processes. Each read or write notice has a corresponding bitmap that describes precisely which words of the page were accessed. These bitmaps are returned

to the barrier master for each page and interval on the check list.

5. The barrier master compares bitmaps from overlapping pages in concurrent intervals. Bitmap comparison is a constant time process, dependent 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 affected variable.

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. The algorithm needs only to examine intervals created during the last barrier epoch. 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. In practice, however, the number of comparisons is usually quite small.

Applications that use only barriers have two intervals per process per barrier epoch. More than two intervals per barrier are 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 can not be 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.

5 Performance

We evaluated the performance of our prototype by searching for data races in implementations of four common shared-memory applications: FFT (Fast Fourier Transform), SOR (Jacobi relaxation), TSP (branch and bound traveling salesman problem), and Water (a molecular dynamics simulation from the Splash2 [22] benchmark suite. All applications were run on DECstations with four 250 Mhz Alpha processors, connected by a 155 MBit ATM. We used only a single processor per machine in order to avoid bus contention.

Table 1 summarizes the application inputs, synchronization types, the number of intervals per barrier, and the overall

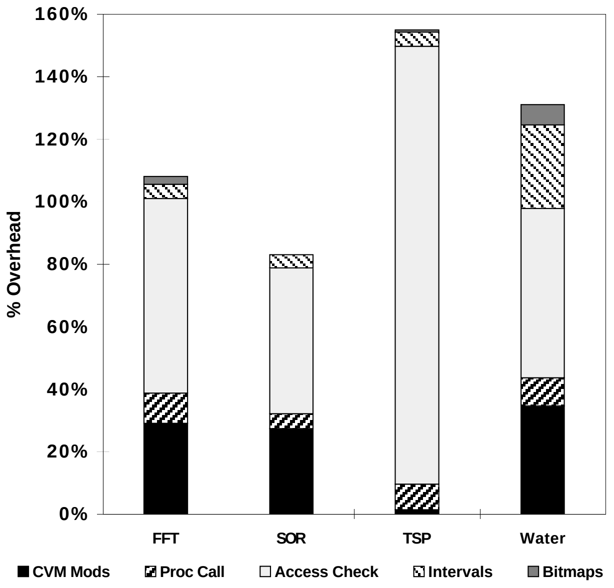

Figure 3. Overhead Breakdown

slowdown for eight-processor runs. “Memory size” is the size of the shared data segment. “Intervals Per Barrier” is the average number of intervals created between barriers. As the number of interval comparisons is potentially proportional to the number of intervals squared, this metric gives a rough idea of the worst-case cost of running the comparison algorithm. This number is greater than 1 for FFT and SOR because our barrier implementation requires two interval structures per barrier. Other synchronization mechanisms require only a single interval per synchronization. As the next section will show, the comparison algorithm is at most only the third most costly form of overhead in our applications.

“Slowdown” is the runtime slowdown for each of the applications, compared with an uninstrumented version of the application running on an unaltered version of CVM. Over the four applications, execution time slows only by an average factor of 2.2. This number compares quite favorably even with systems that exploit extensive compiler analysis.

Figure 3 shows the overhead added by the race-detection mechanism relative to the running time of the unaltered binary, for each application. For example, the execution time of the instrumented FFT binary is 108% longer than that of the uninstrumented binary. “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 notices. “Proc Call” is the procedure call overhead for our instrumentation. ATOM will not currently inline instrumentation; only procedure calls can be inserted into existing code. The ATOM team is working to eliminate this restriction, and the “Proc

Call" column shows how much of the total overhead could be eliminated as a result. "Access Check" is the additional time spent inside the procedure call determining whether an access is to shared memory, and setting the proper bit if so. "Intervals" refers to the time spent using the interval comparison algorithm to identify concurrent interval pairs with overlapping page accesses. "Bitmaps" describes the overhead of the extra barrier round required to retrieve bitmaps, together with the cost of the bitmap comparisons.

The two largest components of the overhead are the access checks and modifications to CVM. The overheads of the interval comparison algorithm and the bitmap checks are usually fairly small. As we will see in the next section, TSP has a higher rate of calls to the runtime analysis routines than the other applications, hence the higher instrumentation overhead. The comparison algorithm adds more overhead for Water than the other applications because of the large degree of fine-grained synchronization.

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 3, we can see that instrumentation accounts for an average of 68% 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 even share some base registers. However, we eliminate most stack accesses by checking for use of the frame pointer as a base register. The fact that all shared data in our system is dynamically allocated allows us to eliminate any instructions that access private data by indirection through the base register that points to statically allocated data. Finally, we do not instrument any instructions in shared libraries because none of our applications pass segment pointers to any libraries. This is the case with the majority of the scientific programs where data race detection is most important. However, we can 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. The first four columns show the number of loads and stores that are not instrumented because they access the stack or statically-allocated data, or are in library routines, including CVM itself. The fifth column shows the remainder. These instructions could not be eliminated as possible data-race participants and are therefore instrumented by ATOM to make a procedure call to an analysis routine each time the memory access is executed.

App	Load and Store Instructions				
	Stack	Static	Library	CVM	Inst.
FFT	1285	1496	124716	3910	261
SOR	342	1304	48717	3910	126
TSP	244	1213	48717	3910	350
Water	649	1919	124716	3910	528

Table 2. Instrumentation Statistics

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 contained 131,668 load and store instructions. Of these, 124,716 instructions are in libraries. A further 1285 instructions access data through the frame pointer, and hence reference stack data. Another 3910 are in the CVM system itself. Finally, 1496 instructions access data through 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, only 261 memory access instructions remain that could possibly reference shared memory, and hence form part of a data race.

Nonetheless, the last two columns of Table 3 show that the majority of run-time calls to our analysis routines are for private, not shared, data. "Inst. Accesses Per Second" refers to the number of instrumented loads and stores executed per second, and the number of these calls to our instrumentation routines that turn out to be for shared or private data. The high rate of instrumented accesses for TSP explains the large "Access Check" overhead for TSP in Figure 3. Accesses to shared data are distinguished from accesses to private data by comparing the address with that of the shared data segments.

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 winnowed down to those interval pairs for which an overlap of pages is found (i.e. one interval of a pair reads from page x and the other interval in the pair writes to 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 page x are to different locations in x (not a data race), or true sharing, in which the loads and stores overlap at least one location (data race).

The first column of Table 3 shows the percentage of intervals that are involved in at least one such concurrent interval pair. This number ranges from zero for SOR, where

	Intervals Used	Bitmaps Used	Msg Ohead	Inst. Accesses Per Second Shared	Inst. Accesses Per Second Private
FFT	15%	1%	0.4%	311079	924226
SOR	0%	0%	1.6%	483310	251200
TSP	93%	13%	1.3%	737159	2195510
Water	13%	11%	48.3%	145095	982965

Table 3. Dynamic Metrics

this is no unsynchronized sharing (true or false), to 93% for TSP, 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 7% of all intervals, as we do for TSP, we may be eliminating a much higher percentage of interval pairs.

The second column of Table 3 shows that an average of only 6% 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.

5.3 The Cost of CVM Modifications

Figure 3 shows that almost 22% 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 third column of Table 3 shows the bandwidth overhead of adding read 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. The bandwidth overhead for Water is much larger than for the other applications because of the fine-grained synchronization, and hence the large number of intervals.

The bandwidth consumed by read notices prevents us from running larger input sets because current message sizes are already at system maximums. Our communication code can, and eventually will, be modified in order to accommodate larger messages.

6 Discussion

6.1 Reference Identification

The system currently prints the shared segment address for each detected race condition, together with the interval indexes. 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 each shared memory access. 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 for retaining this information 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* [18], i.e. synchronization order is deterministic. This is not the case in either of the two applications 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.

6.2 Scalability

Figure 4 shows runtime slowdown versus number of processors. Slowdown actually decreases as we increase the number of processors. This seemingly anomalous result has two causes. First, interval and bitmap comparison overhead is serialized at the master process, and hence *observable* overhead from these sources remains constant as system

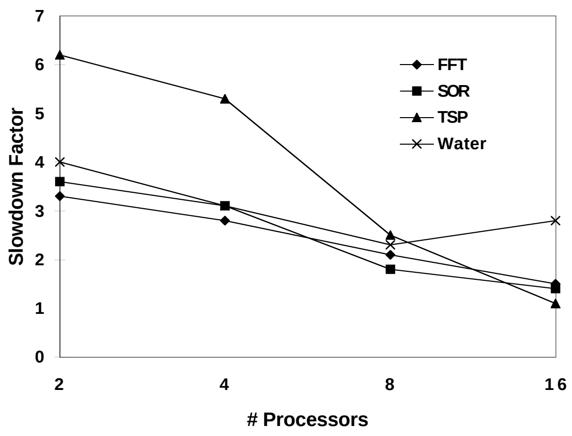

Figure 4. Slowdown Factor versus Number of Processors

size is increased. Instrumentation costs, however, occur in parallel with the shared accesses. As system size increases, therefore, per-process computation and observable instrumentation overhead decreases.

Second, the combination of modest problem sizes, fast processors, and the large page size of the DECstations result in our applications getting very modest speedups even with the unmodified version of the single-writer protocol used in this study. Hence, at least some of the overhead of the race detection algorithm is probably masked by DSM overhead. However, none of these limitations are intrinsic to our approach. Our problem sizes are small because of message size limitations. We are modifying the underlying communication layer to alleviate this problem. The large page size exacerbates the problems of false sharing associated with single-writer protocols [12] protocols. We based our prototype on CVM's single-writer protocol in order to minimize complexity, but our algorithm will work identically with CVM's multi-writer protocol.

Finally, comparison to determine if two intervals are concurrent is a constant-time process, as each interval is marked with a vector timestamp [14, 10]. Comparison of two concurrent intervals to determine whether their page lists overlap is currently $O(n^2)$ in the size of the lists, as they are usually very small (i.e. less than ten). If we encountered applications where these lists grew large, we could perform the comparison in time linear with respect to the number of pages in the system by implementing page lists using bitmaps.

6.3 Global Synchronization

The interval comparison algorithm is currently 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 [8]. 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.

P_1	P_2	P_3
$w_1(qPtr)100$		
$w_1(qEmpty)0$		
{*missing release*}		
	{*missing acquire*}	
	$r_2(qEmpty)0$	$w_3(37)\ldots$
	$r_2(qPtr)37$	$w_3(38)\ldots$
	$w_2(37)\ldots$	$w_3(39)\ldots$
	$w_2(38)\ldots$	$w_3(40)\ldots$

Figure 5. The race $w_2(37) - w_3(37)$ would not occur in SC system

6.4 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 issue is whether to report all data races, or only those that would also occur during sequentially-consistent executions of the program. Their example (somewhat simplified) is shown in Figure 5, where the notation `op (loc) val` indicates a read or write operation performed on location `loc`, that respectively reads or writes value `val`. If the missing synchronization operations were present, there would not be any races. P_2's read of `qPtr`, $r_2(qPtr)37$, would return 100 instead of some older value (37 in this case), and P_2's subsequent writes would be to locations 100 and above. However, given that the synchronization is not present, only the `qPtr` and `qEmpty` races would have occurred on sequentially consistent hardware. If $w_1(qEmpty)0$ had propagated to P_2, any sequentially consistent system must also have sent the results of $w_1(qPtr)100$. This is not the case with weak memory systems, which can usually reorder the effects of write operations between synchronization points at will. Hence, the races between $w_2(37)$ and $w_3(37)$, etc., only occur on weak memory systems.

This is an instance of a more general problem, i.e. whether to return all data races, or only "first" data races, those that are not affected or caused by any prior race. Our system currently reports all races, but could be modified to report only first races without requiring more information to be gathered. Determining whether one race is affected by another effectively consists of deciding whether a happens-before-1 relationship exists between any of the operations in one race and any of the operations in another. Since barrier operations are semantically equivalent to releases by all arriving processors to the barrier master, followed by the barrier master releasing to all other processors, any race in a prior barrier epoch must necessary affect all races in subsequent epochs. Hence, all "first" races must occur in the same barrier epoch. Modifying our system to perform this check online is a trivial extension.

The second problem with accuracy of dynamic race-detection algorithms is reliability of ordering 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.

6.5 Further Performance Enhancements

There are several ways that overhead can be further reduced. First, the ATOM team has promised a new version that allows instrumentation code to be inlined. The Shasta project [20] has already demonstrated a version of ATOM with this feature. Figure 3 shows that an average of 6.7% of our overhead is caused by the procedure call. This overhead will be eliminated when we get the new version of ATOM.

Second, we currently instrument both load and store instructions. This is necessary because our system is currently built on top of a single-writer LRC protocol [12]. Converting our system to use the multi-writer protocol would allow us to exploit existing *diffs*, which summarize per-page modifications, to extract write accesses. We would then be able to dispense with the monitoring of store instructions. Since approximately 68% of the overhead is from instrumentation, and 25% of all data accesses are stores, we should be able to eliminate at least 17% of overall overhead.

A disadvantage of this approach is a slightly weaker correctness guarantee. Diffs only contain *modifications* to shared data. If a shared value is overwritten with the same value, the data location will not be in the diff, and any data race involving this location may not be detected.

Finally, Table 2 shows that nearly 68% of the calls to our instrumentation routines turn out to be for private data. Our current analysis tracks references only through the same basic block. If the value defined before that point is used to reference an unknown data location, we conservatively assume that the location is shared, and hence instrument the access. Inter-procedural analysis would allow us to eliminate many of these "false" instrumentations, and reduce overall overhead. This analysis can be done with the current ATOM system, but will be much easier with a version promised in the near future.

7 Related Work

There has been a great deal of published work in the area of data race detection. However, as previously mentioned, most prior work has dealt with applications and systems in more specialized domains. Bitmaps have been used to track shared accesses before [5], but we know of no other implementation of on-the-fly data-race detection for explicitly-parallel, shared-memory programs without compiler support.

Our work is closely related to work already alluded to in Section 5, 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 *online*. We are therefore able to perform all of the analysis online as well, and do away with trace logs, post-mortem analysis, and much of the overhead.

We have also just become aware of unpublished work on execution replay in TreadMarks that could be used to implement race-detection schemes. The approach of the Reconstruction of Lamport Timestamps (ROLT) [19] 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 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. The primary thrust of our work is in using the underlying consistency mechanism to prune enough information *online* 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 our perturbation of the parallel computation.

8 Conclusions

We have presented a new on-the-fly data race detection technique that allows data-race detection in explicitly-parallel, shared-memory programs. Our technique abstracts synchronization ordering from consistency information already maintained by 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 online.

The primary costs of data-race detection in our system are in tracking shared data accesses. We use ATOM to instrument load and store instructions with calls to our library. We are able to statically eliminate more than 99% of all loads and stores in binaries by identifying accesses to stack variables and statically-allocated global variables. Nonetheless, the majority of the runtime calls to our library are for non-shared accesses. Overall, our applications slow down by an average factor of approximately two.

We used our system to analyze four shared-memory programs, finding data races in two. One of the programs, TSP, allows data races in order to improve performance without violating correctness. The data race in the other program, which was from a standard benchmark suite, was a bug. We believe that the utility of our techniques, in combination with the generality of the programming model that we support, can help data-race detection to become more widely used.

Additional information on CVM is available at: `http://www.cs.umd.edu/projects/cvm.html`.

9 Acknowledgments

We would like to thank Carla Ellis and the anonymous referees for all their helpful suggestions and feedback on earlier drafts of this paper.

References

[1] S. Adve and M. Hill. Weak ordering: A new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.

[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] 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.

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

[14] F. Mattern. Virtual time and global states of distributed systems. In *Parallel & Distributed Algorithms*, pages 215–226. Elsevier Science Publishers, Amsterdam, 1989.

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

[16] 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.

[17] 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.

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

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

[20] Daniel Scales and Kourosh Gharachorloo. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[21] 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.

[22] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–37, June 1995.