

# Protocol-Based Data-Race Detection

Brad Richards

James R. Larus

Computer Science Department  
Vassar College  
124 Raymond Avenue  
Poughkeepsie, NY 12604 USA  
richards@cs.vassar.edu

Computer Sciences Department  
University of Wisconsin–Madison  
1210 West Dayton Street  
Madison, WI 53706 USA  
larus@cs.wisc.edu

## Abstract

Distributed Shared-Memory (*DSM*) computers, which partition physical memory among a collection of workstation-like computing nodes, are now a common way to implement parallel machines. Recently, there has been much interest in DSM machines that use software, instead of hardware, to implement coherence protocols to manage data replication and cache coherence. Software offers many advantages, not the least of which is the possibility of adding significant functionality — such as race detection — to a protocol. This paper describes a new, transparent, protocol-based technique for automatically detecting data races on-the-fly. An implementation of this approach in a DSM system running on a Thinking Machines CM-5 found data races in two of a set of five shared-memory benchmarks. Monitored applications had slowdowns ranging from 0–3 on 32 nodes.

## 1 Introduction

Race conditions arise in shared-memory parallel programs when accesses to shared memory are not properly synchronized. There has been much interest in efficient tools for detecting and reporting these race conditions since a lack of synchronization can lead programs to behave unpredictably. One promising approach to implementing these tools exploits recent fine-grained distributed shared-memory systems in which the coherence policies are implemented in software, instead of being rigidly encoded in hardware. Experiments have shown that the performance penalties for implementing coherence actions in software, instead of hardware, are relatively small (especially if there is hardware support for common operations [7, 17]), and that using the flexibility of software to tailor protocols to the needs of applications can result in tremendous performance increases [5]. Research platforms, such as FLASH [7] and Tempest [16], have paved the way for systems from Sequent [8] and DEC [20].

This paper describes a new use for the flexibility offered by software coherence policies: Implementing a transparent,

---

This research supported by: NSF NYI Award CCR-9357779, with support from Sun Microsystems, and NSF Grant MIP-9625558.

protocol-based technique for detecting data races on-the-fly. Fine grained DSM systems enable efficient, real-time detection of data races, since they already contain a mechanism to invoke the coherence protocol in response to shared-memory accesses. The protocol need only be extended to monitor each access to shared memory and maintain a history of the accesses. The information in the histories is sufficient to detect data races on-the-fly in programs with barrier-only synchronization. Data races can be found in programs with pairwise synchronization by using the access histories in conjunction with additional sequencing information, such as vector timestamps [24], or by using techniques like lockset refinement [19].

Most previous race-detection techniques required support from a custom parallelizing compiler or other tools. This tied race detection to a particular language, implementation, and platform. The fine-grained, protocol-based race detection scheme described in this paper has low overheads and is completely independent of program source code. Race detection can be performed on programs written in any language, and on library routines for which the source may not be available. The approach therefore makes efficient race-detection available to a wider audience than was previously the case.

## 2 Background

An *event* in a parallel shared-memory program represents the execution of a *set* of consecutive program statements that do not include any explicit synchronization. A *data conflict* exists between two events if one writes a shared memory location that the other reads or writes. Data conflicts by themselves do not cause problems. Only when a program's synchronization is insufficient to prevent conflicting events from executing concurrently does a data race arise. (We make no attempt to detect general races, which arise when programs intended to be deterministic execute nondeterministically.)

Netzer and Miller [11, 13] recognized three distinct types of data races: *actual*, *apparent*, and *feasible*. An actual data race occurs when two conflicting events execute concurrently in a given parallel execution. Apparent data races capture the notion that, although a pair of conflicting events might not have executed concurrently, they *could* have given a program's synchronization. But, since apparent races are defined only in terms of a program's explicit synchronization, the set of apparent races may contain data races that are prevented from occurring by the program's semantics. Feasible races are those permitted by both a program's syn-

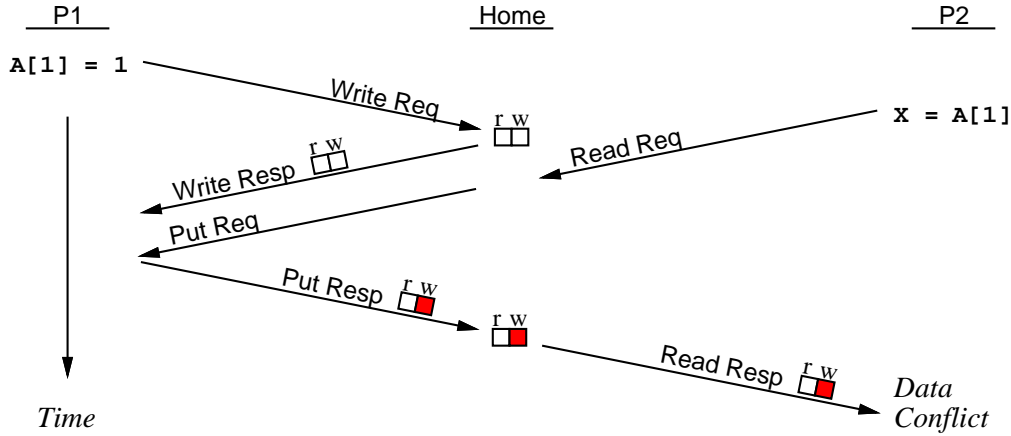


Figure 1: Protocol detecting a data conflict

chronization and its control and data dependences. Netzer [11] proved that finding either feasible or apparent races in programs containing synchronization strong enough to implement mutual exclusion is NP-hard. Thus only actual races can be found in practice, though apparent races can be found efficiently for programs with weaker synchronization, such as Post/Wait.

Data-race detection involves two components: Finding conflicting accesses — references to the same location by a pair of events — and determining whether the conflicting events did (or could have) executed concurrently. Either component can be performed statically, or with the aid of run-time information. Static methods are necessarily conservative, since information about both the control-flow and accessed memory locations is imprecise. The spurious races that result can overwhelm users [6, 9].

Post-mortem [1, 3, 12] and on-the-fly methods [4, 6, 9, 14, 15, 19, 22, 25] improve race-detection accuracy by collecting information from actual executions. This information is either analyzed off-line, in the case of post-mortem techniques, or during execution in on-the-fly methods. The monitored programs provide complete information about memory locations accessed, so data conflicts can be detected accurately. Determining whether the conflicting events executed concurrently can be accomplished either through vector timestamps [3, 12, 15], or by maintaining specialized access histories at runtime.

### 3 Protocol-Based Race Detection

Efficient detection of data races is possible in DSM systems because they already contain a mechanism for invoking coherence protocol actions in response to shared-memory accesses. This provides a means for monitoring accesses at run-time and detecting data conflicts. The protocols can be extended to maintain an access history for each shared-memory location and to pass histories along with data as memory blocks move from processor to processor. The protocol detects data conflicts at an access by searching a history for a conflicting previous reference.

Figure 1 shows a protocol detecting a data conflict. The access history, described in more detail in Section 4.2, is shown here as a pair of boxes denoting reads and writes to  $A[1]$ . A copy of the history is kept at the home node, and processors send the current history with all data-carrying messages. The protocol at processor P1 updates the history

for  $A[1]$  to include a write before allowing the application to complete the modification. The protocol at processor P2 detects the prior write when it inspects the access history that arrives with the readable copy of the block.

If program synchronization does not order the conflicting references, a race has occurred. For programs with barrier-only synchronization, all code between a pair of barriers runs concurrently, so data conflicts imply the presence of an actual data race. Programs employing critical sections must use other techniques, such as vector timestamps, to determine a partial ordering on the sections. References from unordered critical sections represent actual data races.

In our system, the access histories are cleared at each barrier. At access faults, the faulting processor inspects the access history to determine whether the current access completes a data race. (This is the case if the access is a write and there has been a prior read or write by another processor, or if the access is a read and there has been a prior write.) If not, the access history is updated appropriately and the access is allowed to proceed.

Note that all of the operations necessary to detect data races can be performed at the protocol level. Thus, there is no need to rely upon support from custom compilers or binary rewriting tools. Access to source code is also unnecessary, so the technique works for programs written in any language and for library routines for which source is not available.

### 4 Implementation

#### 4.1 Detecting All Accesses

Coherence protocols are typically not invoked on *every* access. Once a faulting access is handled, the data is cached locally, so that subsequent accesses to the same coherence block can proceed without protocol involvement. This is by design, since unnecessary invocation of the protocol decreases performance. But race-detection protocols potentially need to monitor every memory access or they can miss data races. (A race would go undetected if the first access of a conflicting pair was monitored, for example, but the second was not.)

One solution is keeping all coherence blocks in an invalid state, even after fetching data in response to a fault. Since every reference to an invalid block invokes the protocol, the protocol observes all references. Unfortunately,

fetching data on every fault would cause prohibitive slow-downs. Instead of fetching data on each fault, copies of the block data could be kept in local memory and used to satisfy faults.

A better solution is available on *data-maintaining* DSM systems, where blocks can be marked invalid without corrupting data. On these systems, one can maintain invalid block *access permissions* even when the block's data is valid. Accesses to such a block will invoke the protocol as required, but can be satisfied from the block's data without fetching new data from the home. Our protocol was implemented on such a system and, as will be shown in Section 5, the performance penalty for the increased number of faults is relatively small.

## 4.2 Access History Details

Like others [6, 9, 10], we choose to limit a memory location's access histories to a single entry for reasons of efficiency. Races can be missed as a result, if there are multiple races involving the same location, but at least one race involving a location is guaranteed to be caught and can be used to debug parallel programs.

Entries in a location's access history can be as simple as a pair of bits to denote whether a location has been read or written. While bits require less space than byte-entries, they also reduce the amount of information available for describing a race and can cause spurious races to be reported [18]. Byte-entries are an improvement over bits with respect to accuracy, since they allow the ID of the processor making the most recent read and write to be recorded. The results described in this paper are for histories with one-byte entries for each monitored memory location.

Access histories can be maintained for regions of memory of various sizes. For applications that manipulate large shared-memory objects, it might be acceptable to keep access histories for cache-block sized regions of memory or larger. However, data races can be missed if the application shares data at a finer granularity. Results for granularities varying from word-level to block-level are reported in Section 5.

The custom race-detection protocols currently only handle system-wide synchronization, as tests for concurrency in programs using only barriers can be performed efficiently. If access histories are reset at barriers, non-empty histories imply an access since the last barrier, and testing for concurrency reduces to testing for non-empty access histories.

## 4.3 Gathering Race Information

The race-detection protocol has complete information about the second of each pair of conflicting memory accesses (the *sink*), but all information about the first access (the *source*) must come from the access history. When a race is detected, the protocol records the value of the program counter for the sink access, the memory address involved, and the ID of the processor that made the source reference. The first access' program counter value could be maintained as well, but would quadruple the size of the access histories.<sup>1</sup>

The system writes information on detected races to a file after an application has run to completion. Without source code, this information can only serve to determine whether

<sup>1</sup>Since histories are transmitted across the network with the block data, their size influences the bandwidth requirements of the protocol. Bandwidth considerations were given priority here over race report detail in an effort to create a tool with overheads low enough to consider using it full time.

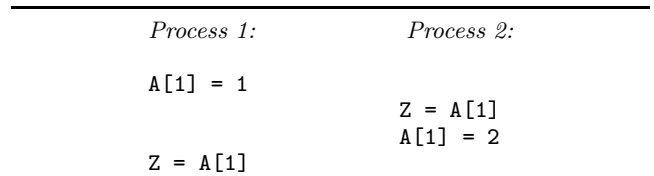


Figure 2: Monitoring the first read and write

or not races occurred. If source code is available, we have a tool that annotates the code and highlights the lines involved in races by mapping the program counter information from the race reports back to source code. Currently, the tool does not report which pairs of source lines were competing for access to a location, but that information could be presented as well. In our tests, simply highlighting lines of source code was sufficient to point out program errors.

Once the decision to keep incomplete information about the source reference is made, the protocol need not be invoked on *every* access to shared memory. Only the *first* read and write after fetching a location must be monitored. The protocol still detects all races for which these references are the sinks, and the access-history updates performed are sufficient to catch races for which these or any later references by the same processor are the source. Figure 2 shows an interleaving of accesses to A[1] that produces two races. The read of A[1] by process 2 completes a race with the write by process 1, and must therefore be monitored. The write to A[1] by process 2 must also be monitored so the access history for the region containing A[1] is properly updated. Any subsequent reads or writes by process 2 are redundant (with respect to race detection) since the access history *already* records process 2 as the most recent reader and writer. The read by process 1 at the end of the example is the first after fetching a copy of the block containing A[1], and must be monitored to detect the race with the write on process 2.

By monitoring only the first read and write after obtaining a copy of a block, we reduce the number of times the protocol must be invoked. This can result in substantial performance gains. Unfortunately, the software DSM upon which our implementation was based can only change access tags at the granularity of an entire coherence block. Thus, a block cannot be made readable, for example, until each race-detection region on the block has been read. Doing otherwise risks missing races since access histories could be incomplete.

## 4.4 Detected Races

As with all approaches that bound access histories [6, 9, 10], the protocols detect a *subset* of the actual races present in an execution. Races can be missed if there are multiple races involving the same location, but at least one race involving a location is guaranteed to be caught and can be used to debug parallel programs. Finding a subset of the actual races is not necessarily a handicap. Experience with race detection systems has shown that reducing the number of reported races makes the protocols more useful as diagnostic tools, since users need examine less information. Reporting even a single race for a location is sufficient to warn the user of a possible error.

Since the protocol only reports a race if two or more processors accessed a shared location between barriers, a race report implies that the application under test is nondeterministic unless the program used forms of synchronization of which the protocol was unaware. The custom race-detection

<i>Application</i>	<i>Description</i>	<i># Lines</i>	<i>Synchronization</i>
Appbt	3D CFD Solver	5,100	Locks, Barriers
Em3d	EM Wave Propagation	2,500	Two locks, Barriers
Gauss	Gaussian Elimination	900	One lock, Barriers
LCP	Linear Complementarity	1,550	One lock, Barriers
Water	$n^2$ Molecular Simulation	2,400	Locks, Barriers

Table 1: Benchmark applications

<i>Application</i>	<i>Race Events</i>	<i>Unique Addresses</i>	<i>Unique References</i>	<i>Unique Lines</i>	<i>Excluding Crit. Sec.</i>
Appbt	723,333	43,069	163	91	1
Em3d	149,127	43,034	13	3	0
Gauss	26,167	1	2	2	0
LCP	1,078,289	4,161	13	9	2
Water	2,466,122	4,616	36	25	1

Table 2: Races detected by Race-4. Columns show total number of races, and the number of memory addresses, PC values, and source lines involved. The last column is source lines involved outside of critical sections.

protocols could potentially be extended to handle pairwise synchronization via vector timestamps [24] or by using techniques such as lockset refinement [19]. Applications with additional forms of synchronization can be run on the protocols, but spurious races may be declared since only the ordering constraints imposed by barriers are recognized by the race-detection mechanisms. As is shown in Section 5.4, these spurious races can be effectively removed in a post-processing phase.

It is almost certainly the case that the access ordering for a given execution on the race-detection protocol is slightly different than the order that would result without the race-detection mechanisms. These differences do not lead to false race reports, however, because races are only reported when unordered accesses are performed between barriers. If such a pair of accesses were made on the race detection protocol, they were possible in some legal execution of the program on the standard protocol as well.

#### 4.5 Verification

The race-detection protocol was written in the specification language provided by the Teapot tool [2], and its verification component was used to ensure both that the race-detection protocol maintained consistent data, and that it successfully caught data races. The verification process proceeded in two steps. First, the basic protocol was designed and tested without any race-detection functionality. Even without race-detection features, the protocol is significantly more complex than the base Stache [17] protocol, since it must keep blocks in an invalid state until each location on a block has been read or written.

Once the underlying protocol was working correctly, mechanisms for recording access histories and detecting races were added, and the race-detection functionality was tested as well. Since the verification system has perfect knowledge of the accesses performed, it can be extended to recognize data races along with the protocol and ensure that the protocol also finds them.

## 5 Performance

This section describes the performance of the custom race-detection protocols on a set of five benchmarks. Slow-downs

range from zero to less than a factor of three. Actual program errors were found in two of the benchmarks, Appbt and LCP, by the custom race-detection protocols.

### 5.1 Experimental Setup

All experiments were performed on a 32-processor CM-5 using the Blizzard-S [21] DSM system. The baseline against which the race-detection protocols were compared was the Teapot-generated [2] version of the Stache protocol. All protocols maintained coherence at the 32-byte block granularity, and race-detection granularities varied from 4 to 32 bytes. The race-detection protocols are referred to by names of the form Race- $N$ , where  $N$  is the race-detection granularity in bytes.

### 5.2 Benchmarks

Each of the benchmarks used to test the race-detection protocols (Table 1) use both locks and barriers for synchronization. Since the race-detection protocols only recognize barrier synchronization, critical sections implemented by locks appear to consistently fail from a race-detection point of view. This causes detection of spurious races involving references in critical sections, but these can be automatically detected and need not be presented to the user.

Appbt is a kernel from the NAS parallel benchmarks representing the computation and communication found in 3D computational fluid dynamics problems. It makes frequent use of both locks and barriers for synchronization. Em3d models the propagation of electromagnetic waves through objects in three dimensions. The simulation is formulated as a computation on a bipartite graph with directed edges, and uses only barrier synchronization during the computation phase. The graph-building phase uses two locks. Gauss performs gaussian elimination and backsubstitution. It uses a single lock to implement a reduction across processors, and barrier synchronization elsewhere. LCP is a parallel implementation of the linear complementarity problem, written by Satish Chandra. Only barrier synchronization is used during the computation, though a single lock is used to combine normalization information at the end. Water is one of the Splash [23] benchmarks, and simulates a body of water molecules. The version used here is the original  $n^2$  application, and uses both locks and barriers during the

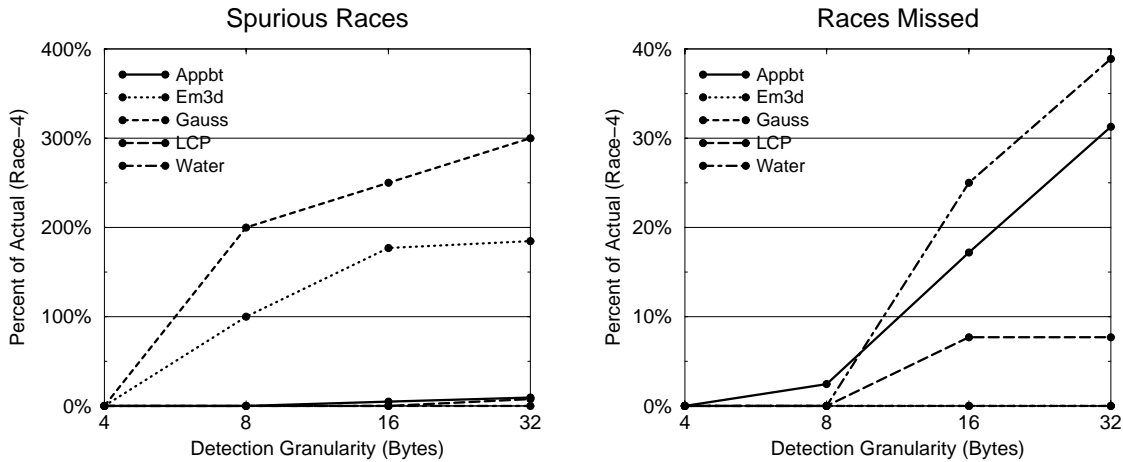


Figure 3: Spurious and missed data races

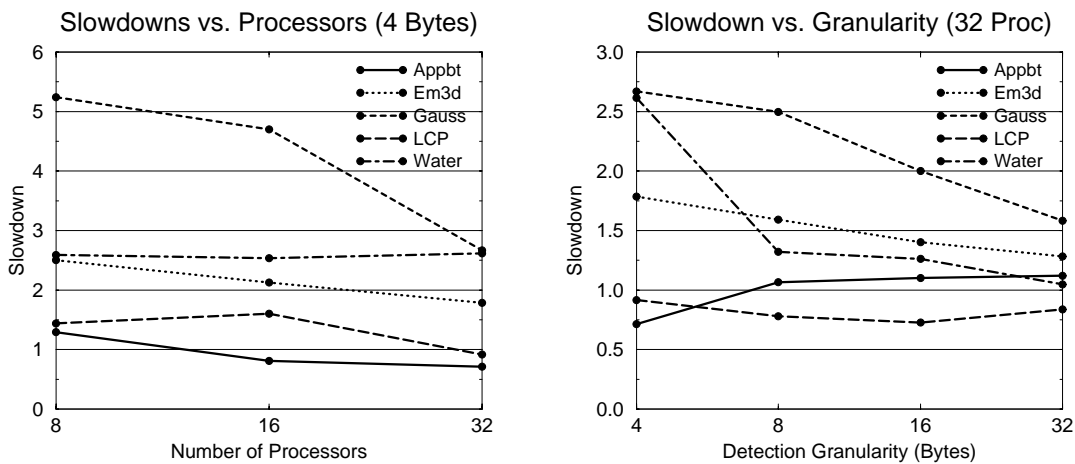


Figure 4: Race-detection slow-downs

computation.

### 5.3 Blizzard-S

Blizzard-S is an implementation of the Tempest software shared-memory interface [17] that provides block-level access control via binary rewriting of parallel applications. At each access to shared memory, an average of about 15 instructions are inserted that inspect the memory's access permissions and determine whether the access should be allowed to proceed. In the case of an access fault, the appropriate fault handler is immediately invoked.

### 5.4 Race Detection Results

Table 2 summarizes the races found by Race-4, the most precise of the race-detection protocols. The number of race events is the total number of actual data races detected. Since the same pair of source-line references can be involved repeatedly in races with each other, more useful metrics are the unique memory addresses, references, and source code lines involved in races.

The remaining line in Appbt represented an actual program error, as did the two in LCP. In neither case were the authors aware of the missing synchronization which, in each case, allowed a sequential normalization phase to overlap

with parallel computation. The line in Water was the result of a synchronization scheme (spinning on a shared memory value) that was not based on locks or barriers.

The race-detection granularity affects the accuracy of race detection. If the granularity is larger than a single data item, for example, an access to one data item is treated as an access to all items in the given memory region. This can cause reporting of false races if a pair of apparently-conflicting accesses are actually to different data items within the same memory region. It can also cause races to be missed, since a pair of races involving neighboring data items might now be collapsed into a single race involving the items' containing region. Figure 3 shows spurious races detected by the race-detection protocol as well as actual races missed as the detection granularity is increased. The data was generated by assuming that, for all benchmarks, Race-4 correctly identified the unique references involved in races. Any additional references were considered false. The graphs show spurious and missed races as a percentage of the total number of actual races. The results show an increase in missed races as the granularity increases, as expected.

### 5.5 Performance Results

Figure 4 shows the slow-downs for the five benchmarks as both the number of processors and the detection granularity

<i>App.</i>	<i>Stache</i>	<i>Byte-4</i>	<i>Byte-8</i>	<i>Byte-16</i>	<i>Byte-32</i>
Appbt	65.7	41.6	54.5	60.7	63.0
Em3d	1.3	1.3	1.1	1.1	1.1
Gauss	1.4	1.8	1.7	1.5	1.3
LCP	13.6	7.1	6.9	6.5	8.1
Water	1.3	2.0	1.2	1.2	1.3

Table 3: Network contention (tries per send)

<i>Application</i>	<i>Clearing Histories</i>	<i>Access Tests</i>	<i>Total</i>
Appbt	0.15%	0.68%	0.83%
Em3d	1.10%	0.29%	1.39%
Gauss	7.56%	0.46%	8.02%
LCP	0.38%	0.51%	0.89%
Water	0.16%	0.76%	0.92%

Table 4: Race-detection overheads

is varied. On 32 processors, slow-downs range from almost a factor of three on Gauss, to slight *improvements* in Appbt and LCP. These figures remain stable for four of the five benchmarks as the number of processors is decreased. Several factors are behind the relatively modest slow-downs. First, since Blizzard-S instruments each load and store to shared memory, the extra access faults are essentially detected for free. Second, the additional faults are handled locally – they do *not* have to transfer any data across the network – and so execute quickly. Finally, the extra faults can actually help reduce network contention on the CM-5.

A convenient way to assess contention is to measure the average number of attempts required to inject a message into the network. The data on the number of tries per send for 32-processors is shown in Table 3. Both of the applications that see speedups have extremely high levels of contention when run on the Stache protocol, but see lower levels of contention when race-detection protocols are used.

In general, as race-detection granularities decrease, protocols are less likely to access each race-detection region on a block. This implies an increase in faulting accesses, since fewer blocks can be upgraded to read-only or writable states. But, at each fault, the Tempest system pulls waiting messages out of the network and queues them for delivery. Thus, faulting more often can be beneficial as it takes pressure off of the network buffers and lowers contention. This mechanism is behind the decreased contention numbers for Appbt and LCP.

While the race-detection protocols do not send more messages than Stache, each message is potentially larger since they also include access histories.<sup>2</sup> This can exacerbate contention in some situations, as with Gauss and Water. These two benchmarks have low contention initially, so increasing the number of access faults does not help. Instead, the increase in network traffic drives up contention and gives sizable slow-downs on the race-detection protocols.

In Water, there is a large discontinuity between the slow-downs for four and eight bytes. This is because all data in Water is double-precision. When race detection is performed at the four-byte granularity, the protocol assumes that all accesses (even of eight-byte quantities) touch only four bytes. Since there are always unreferenced four-byte quantities on each block, blocks can never be upgraded to

<sup>2</sup>The difference in number of messages sent between race-detection runs and the base Stache protocol are no larger than the differences from run to run on Stache.

read-only or writable and the protocol must fault on every reference.

The per-processor overheads introduced by race detection, shown in Table 4, are all quite small. The additional cycles spent clearing access histories at barriers and performing tests for races account for less than 1% of the total execution time for three of the five benchmarks. However, the table does not show overheads for invoking the extra fault handlers.

Table 5 reports data on the fraction of each application’s accesses that fault and invoke the protocol. The total number of accesses was measured with a version of Race-4 that recorded *all* shared-memory accesses. This was used as a baseline against which to compare the number of faulting accesses for Stache and Race-4. The final columns report the relative increase in monitored accesses when using Race-4. The data indicates that the optimization of upgrading a block to read-only or writable once each region has been accessed decreases the number of faulting accesses only slightly when using Race-4.

The Blizzard-S system reports statistics after each run, including the total number of cycles spent handling faults and the number of messages sent, and a rough estimate of the cost of the extra data-race faults can be computed from this information. The Em3d benchmark on the base Stache protocol spent an average of 870 cycles handling each access fault. Assuming that the average data-moving fault on the race-detection protocol cost the same (a reasonable assumption, since network contention doesn’t change for Em3d), the average cost of locally-handled faults can be estimated by determining the extra cycles spent handling faults and the number of extra faults encountered. For Em3d, this gives about 12 cycles per locally-handled fault — roughly the same performance penalty as an access test.

## 5.6 Summary

The results presented above validate the protocol-based approach to race detection. The protocol scheme found actual program errors in two of the five benchmarks. The performance results were surprising in several respects. First, program *speedups* were not anticipated. It should be stressed, however, that they were the result of system-dependent factors (i.e. the shallow network queueing on the CM-5). Also, while the number of access faults was expected to increase when using the race-detection protocols, the magnitude of the increase in some cases was surprising. The relatively minor slow-downs are impressive given the dilutions in Table 5.

## 6 Related Work

Dinning and Schonberg [4] implemented an on-the-fly scheme for detecting apparent data races. They support both fork/join and pairwise synchronization and obtain concurrency information from a static analysis of the source code. Our protocol-based approach currently only supports barrier synchronization, and detects actual data races. Over a set of four benchmarks, Dinning and Schonberg report program slow-downs ranging from three to six on eight processors, using access histories limited to only one or two entries. The race-detection protocol has roughly the same worst-case performance on eight processors as well.

Mellor-Crummey [9] describes a method for encoding the static concurrency information, *offset-span labeling*, that has improved space and time bounds for programs that do not

App.	Total Refs (M)		Stache		Race-4		Dilation	
	Read	Write	Read	Write	Read	Write	Read	Write
Appbt	471	16	0.1%	2.7%	58.9%	100.0%	402	38
Em3d	3	9	33.7%	11.0%	100.0%	100.0%	3	9
Gauss	89	47	0.7%	0.2%	56.1%	99.7%	83	574
LCP	138	1	1.0%	16.3%	74.1%	100.0%	76	6
Water	135	16	0.7%	4.2%	98.7%	100.0%	141	24

Table 5: Statistics on read and write faults

use pairwise synchronization, but gives no performance results. Hood, Kennedy, and Mellor-Crummey [6], present a technique for detecting data races in Fortran programs that use barriers and structured synchronization based on ordered sequences. They keep only one entry in the access histories, and use static analysis to reduce the number of monitored shared variables. Their slow-downs are roughly 40%, but the technique requires compiler support.

Perković and Keleher [15] implemented race detection in CVM, a page-based release-consistent DSM. Systems that implement release consistency maintain ordering information that enables them to make a constant-time determination of whether two accesses are concurrent. Perković and Keleher extended the DSM to collect information about referenced locations and check at barriers for concurrent accesses to common locations. Pairwise synchronization is handled as well as fork/join and barrier, since release-consistent DSM systems must already be aware of all forms of program synchronization. As a page-based approach, Perković and Keleher's system works well on a smaller set of applications than the race-detection protocols described here, since page-based DSM systems do not efficiently support applications with fine-grained sharing.

Savage et. al. [19] dynamically detect data races in lock-based multithreaded programs by determining, on-the-fly, whether shared data items are consistently protected by at least one lock. They use binary rewriting to modify executables such that all shared-memory accesses are monitored at runtime. Currently, this results in program slowdowns of a factor of 10 to 30, and doubles the amount of memory required. The protocol-based approach described in this paper can increase memory consumption by as little as 6%, and produces slowdowns that are roughly an order of magnitude smaller.

Our work is most closely related to a hardware-based cache coherence protocol for CCNUMA machines that Min and Choi [10] designed but never implemented. Like us, they limit access histories to a single entry and do not support pairwise synchronization. Their scheme can miss shared-memory accesses unless the compiler organizes shared data such that at most one word per cache block is used. This restriction would cause an unacceptable increase in the memory requirements of most programs, and would waste precious bandwidth as cache blocks containing a single word of useful data are communicated between processors. The protocols in this chapter allow arbitrary data placement by keeping blocks invalid even after fetching data in response to a fault. Every reference to an invalid block invokes the protocol, so the protocol sees all references.

## 7 Conclusions

This paper describes the design and implementation of a custom cache-coherence protocol that performs on-the-fly detection of actual data races for programs with barrier syn-

chronization. Efficient detection of data races is possible on DSM systems because a mechanism already exists to invoke the coherence protocol in response to shared-memory accesses. The protocol can be extended to maintain access histories, detect concurrency, and watch for data races.

The techniques used to implement the race-detection protocols are applicable to any DSM system where coherence policies are encoded in software. Research platforms, such as FLASH [7] and Tempest [16], have paved the way for software DSM systems from Sequent [8] and DEC [20], and it seems likely that many future machines will be built along these lines.

Protocol-based race detection schemes are completely independent of program source code, and race detection can be performed on programs written in any language and on library routines for which the source may not be available. Overheads for our protocols ranged from zero to less than a factor of three on 32 processors.

## Acknowledgements

The authors would like to thank Bart Miller, Rob Netzer, and the anonymous referees for their many helpful comments and suggestions on this work.

## References

- [1] Todd R. Allen and David A. Padua. Debugging Fortran on a Shared Memory Machine. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 721–727, University Park PA, August 1987.
- [2] Satish Chandra, Brad Richards, and James R. Larus. Teapot: Language Support for Writing Memory Coherence Protocols. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI)*, May 1996.
- [3] Jong-Deok Choi, Barton P. Miller, and Robert Netzer. Techniques for Debugging Parallel Programs with Flowback Analysis. Technical Report 786, University of Wisconsin, Madison, Computer Sciences Department, August 1988.
- [4] Anne Dinning and Edith Schonberg. An Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection. In *Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 1–10, February 1990.
- [5] Babak Falsafi, Alvin Lebeck, Steven Reinhardt, Ioannis Schoinas, Mark D. Hill, James Larus, Anne Rogers, and David Wood. Application-Specific Protocols for User-Level Shared Memory. In *Proceedings of Supercomputing '94*, pages 380–389, November 1994.
- [6] Robert Hood, Ken Kennedy, and John Mellor-Crummey. Parallel Program Debugging with On-the-fly Anomaly Detection. Technical Report TR90-111, Rice University, Department of Computer Science, May 1990.
- [7] Jeffrey Kuskin et al. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, April 1994.
- [8] Tom Lovett and Russell Clapp. STING: A CC-NUMA Computer System for the Commercial Marketplace. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 308–317, May 1996.

- [9] John M. Mellor-Crummey. On-the-fly Detection of Data Races for Programs with Nested Fork-Join Parallelism. In *Proceedings of Supercomputing '91*, pages 24–33, November 1991.
- [10] Sang Lyul Min and Jong-Deok Choi. An Efficient Cache-based Access Anomaly Detection Scheme. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 235–244, Santa Clara, California, April 1994.
- [11] Robert H. B. Netzer. *Race Condition Detection for Debugging Shared-Memory Parallel Programs*. PhD thesis, Computer Sciences Department, University of Wisconsin–Madison, August 1991.
- [12] Robert H. B. Netzer and Barton P. Miller. Detecting Data Races in Parallel Program Executions. Technical Report TR90-894, University of Wisconsin, Madison, Department of Computer Science, August 1990.
- [13] Robert H. B. Netzer and Barton P. Miller. What are Race Conditions? Some Issues and Formalizations. *ACM Letters on Programming Languages and Systems*, 1:74–88, March 1992.
- [14] \*I. Nudler and L. Rudolph. Tools for the Efficient Development of Efficient Parallel Programs. In *Proceedings of the First Israeli Conference on Computer Systems Engineering*, May 1986.
- [15] Dejan Perković and Pete Keleher. Data Race Detection in Release-Consistent DSM. In *Operating System Design and Implementation*, 1996. To appear.
- [16] Steven K. Reinhardt. Tempest Interface Specification (Revision 1.2.1). Technical Report 1267, Computer Sciences Department, University of Wisconsin–Madison, February 1995.
- [17] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–337, April 1994.
- [18] Bradley E. Richards. *Memory Systems for Parallel Programming*. PhD thesis, Computer Sciences Department, University of Wisconsin–Madison, August 1996.
- [19] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. In *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.
- [20] Daniel J. Scales, Kouros Gharachorloo, and Chandramohan A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 174–185, Cambridge, Massachusetts, 1996.
- [21] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain Access Control for Distributed Shared Memory. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 297–307, October 1994.
- [22] Edith Schonberg. On-the-Fly Detection of Access Anomalies. *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, published in *ACM SIGPLAN Notices*, 24(7):285–297, July 1989.
- [23] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. *Computer Architecture News*, 20(1):5–44, March 1992.
- [24] Mukesh Singhal and Ajay Kshemkalyani. An Efficient Implementation of Vector Clocks. *Information Processing Letters*, 43:47–52, August 1992.
- [25] Guy L. Steele Jr. Making Asynchronous Parallelism Safe for the World. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 218–231, January 1990.