Optimal Tracing and Replay for Debuging Shared-Memory Parallel Programs

Robert H.B. Netzer

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-93-15
September 1993

Appears in the 3rd ACM/ONR Workshop on Parallel and Distributed Debugging, San Diego, CA, May 1993

Optimal Tracing and Replay for Debugging Shared-Memory Parallel Programs

Robert H. B. Netzer
rn@cs.brown.edu

Dept. of Computer Science
Brown University
Box 1910
Providence, RI 02912

Abstract

Execution replay is a crucial part of debugging. Because explicitly parallel shared-memory programs can be nondeterministic, a tool is required that traces executions so they can be replayed for debugging. We present an adaptive tracing strategy that is optimal and records the minimal number of shared-memory references required to exactly replay executions. Our algorithm makes run-time tracing decisions by detecting and tracing a certain type of race condition on-the-fly. Unlike past schemes, we make no assumptions about the execution's correctness (it need not be race free). Experiments show that only 0.01–2% of the shared-memory references are usually traced, a 2–4 order of magnitude reduction over past techniques which trace every access.

1. Introduction

Sophisticated debugging requires repeated program re-execution. Because locations of bugs are usually not known in advance, and because tracing every execution detail is impractical, reproducing an erroneous execution over and over is often the only way to track down bugs. However, executions of shared-memory parallel programs are difficult to reproduce. When programs are designed to be nondeterministic, or when programs intended to be deterministic contain race conditions, their executions (on the same input) can differ from run to run. Although this nonrepeatability is often intended, it is a fundamental debugging problem, because without special tools cyclic debugging is impossible. Overcoming this problem requires tracing program executions so they can be replayed as needed for the debugging cycle. This paper presents an *optimal* strategy for tracing executions for replay. Our strategy adapts to the variable access patterns of the execution and makes run-time decisions to trace the fewest shared-memory references required. We present experimental results showing that we often trace only $0.01 - 2\%$ of the shared-memory references, a $2 - 4$ order of magnitude improvement over previous work which traces every reference. For example, with past techniques generating 100 megabyte traces is not uncommon, even for executions of only a few minutes duration. With our strategy such executions can be debugged by generating traces as small as 10 kilobytes.

Our main result is an algorithm for detecting and tracing a certain type of race condition on-the-fly that captures the minimal information required for replay. From a trace of these races the execution can be replayed from its start any number of times. Instead of statically deciding to trace every shared-memory reference (as does previous work), our algorithm makes *run-time* decisions about which references to trace by dynamically adapting to the execution's variable access patterns. We trace the execution order of only those shared-memory references involved in a special type of race condition. These races correspond to edges that comprise the transitive reduction of a graph of the execution's dynamic data dependences. A trace of these races is minimal in the sense that they represent the smallest number of data dependences that must be recorded to ensure that replay exactly exhibits the original dependences. Detecting these races is equivalent to computing the transitive reduction of this graph on-the-fly. Our race detection algorithm is a variant of an existing technique for detecting data races on-the-fly[3], and has the advantage that synchronization is not specially handled (all shared-memory accesses, even those implementing synchronization, are treated uniformly).

We also show how to keep our algorithm's space usage arbitrarily low by trading off space overhead with race detection accuracy (which causes increased traces). Our algorithm shares the worst-case time and space complexity of on-the-fly data race detection, requiring $O(mp)$ space (where p is the number of processes and m is the number of shared-memory locations) and $O(p)$ time per memory reference. Reducing space overhead is important for programs with many shared variables or many processes, where large work buffers would otherwise be required. Although the trace is then not always optimal, our experiments show that trace sizes are not significantly increased.

Adaptive tracing strategies are not only becoming practical for the first time but are becoming necessary. In the past, the cost of writing to disk was insignificant compared to the cost of even a small amount of computation. Processors are now so fast that a significant amount of work can be performed in the time it takes to write only a single trace record. Since this trend is continuing, in the future it will be cheaper to use adaptive strategies that do on-line analysis (to minimize tracing) than to trace more than absolutely necessary. We expect adaptive tracing to be an effective way of coping with the bottleneck caused by fast processors and large-scale parallel machines. This work is part of our larger research effort on developing such adaptive strategies[7, 8].

2. Related Work and Motivation

Replaying an execution of a shared-memory parallel program requires information about how its processes interacted during that execution. All past approaches at tracing this information either trace something about every shared-memory access, or assume the program is race-free (a precarious assumption when the program is being debugged). Below we outline these past approaches and show that such exhaustive tracing is unnecessary.

In one approach to tracing for replay, the *order* in which processes interact is recorded[4, 1]. Each process records the order in which operations are performed on shared objects (Leblanc and Mellor-Crummey's *Instant Replay*)[4], or the order in which synchronization operations are performed (Carver and Tai's SYN-sequence trace)[10]. Another approach is to trace the *data* read from every shared-memory location[9], but too much data is traced to be practical (Pan and Linton[9] report trace generation of 1 megabyte/sec per process on a VAX 11/780). We base our work on Instant Replay, where only orderings are traced.

Instant Replay assumes the program contains a correct mechanism for performing coarse-grain operations on shared objects, and instruments this mechanism to record the relative order in which these operations are performed (but not the order of the individual memory references the coarse operations issue). If operations are of large enough granularity, Instant Replay's tracing overhead is low[4], making it a valuable replay tool. However, it assumes that the coarse-grain operations themselves contain no races (in which case the order of the operations provides information sufficient for replay). This assumption is reasonable if all interprocess communication is through structured primitives (e.g., message-passing libraries) that encapsulate all manipulation of shared data structures; however, for less structured forms of process interaction, this assumption is tenuous and in the worst case replay could fail. When the program cannot be certified race-free, Instant Replay could still be used by recording the order of the individual shared-memory references, but it was not designed for such fine-grain tracing[5] and the resulting overhead could be prohibitive. In contrast, the scheme by Carver and Tai assumes that the program is completely race free. In such a case, tracing the order of synchronization operations alone is sufficient for replay. However, their scheme fails when programs have races. As with Instant Replay, such a scheme cannot replay the execution for debugging unless it is known to be race free.

Even though no values are traced, tracing only the order in which processes interact is sufficient to replay from the execution's start. By forcing a re-execution to access shared memory in the same order, the original data will be recomputed. However, if during debugging the program is not known with certainty to be race free, this approach must trace information about *every* process interaction, and on current parallel machines could result in huge tracing requirements.

Although tracing every interaction is sufficient for replay, it is not necessary; only a subset of the accesses actually need be traced. Our goal is a technique that traces only what is necessary. Figure 1 shows an example program execution containing two processes (the variables "Queue" and "head" are shared). Process P1 inserts data into the head of the shared queue and then increments the head pointer. After P1 finishes these operations, P2 reads the head pointer and removes the data from the queue (if there were synchronization between P1 and P2's references, it would be treated as additional shared-memory references). Figure 1a shows a graph representing the dynamic shared-data dependences exhibited by this execution (a shared-data dependence exists between references

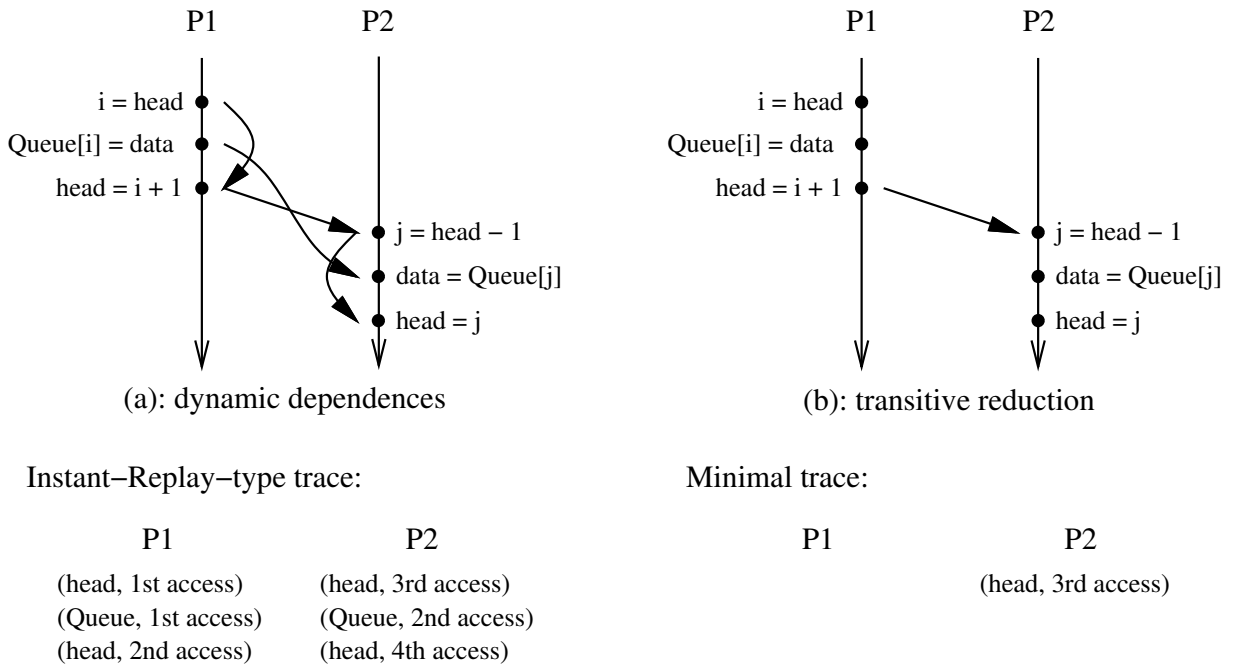


Figure 1. Example execution: (a) dynamic shared-data dependences and Instant-Replay-type trace, (b) transitive reduction of dependences and minimal trace.

a and *b* iff *a* accesses a shared location that *b* later accesses, and *a* or *b* modifies the location). An Instant-Replay-type trace would generate one trace record for each dependence, indicating the order in which the variables are accessed. For example, P1's first trace record "(head, 1st access)" indicates where the first reference to "head" was made. From this trace a re-execution can be forced to exhibit exactly the same data dependences, and indeed *reproducing* these dependences is necessary for correct replay.

However, *tracing* every dependence is not necessary to record enough information to reproduce the dependences. Figure 1b shows the transitive reduction of the dependences, which instead provides sufficient information. The transitive reduction of a directed graph is a (unique) graph with all edges otherwise connected by a path omitted. For example, tracing only that P2 makes the third reference to "head" is sufficient to ensure that P2 makes the third reference during replay; all other references will automatically occur in the correct order because of transitivity. This example illustrates that tracing only references in the transitive reduction is sufficient for replay. Since the transitive reduction is the smallest graph that represents the dependences, it also shows the *minimal* number of references that must be traced to exactly preserve the original dependences.

3. Model

We next outline our model, which is simply a notation for representing program executions. A *program execution* is a triple, $P = \langle E, \xrightarrow{T}, \xrightarrow{D} \rangle$, where E is a finite set of *events* and $\xrightarrow{T}$ and $\xrightarrow{D}$ are relations defined over E.

To reason about replay, we always view executions at a low level as a collection of individual shared-memory references†. We define an *event* to represent the execution of a read or write to a shared location, and we say that two events *conflict* if they both access the same location that at least one writes. We do not view refer-

† In past work we called such a low-level view a *single-access* view[6], denoted as $P = \langle E, \xrightarrow{Ts}, \xrightarrow{Ds} \rangle$. In this paper we omit the "S" subscripts for brevity, but always implicitly assume a single-access view.

ences that comprise synchronization operations differently, but view all references to shared memory uniformly. We also assume a fixed number of processes exist during execution, and denote the i^{th} event in process p as $e_{p,i}$.

The *temporal ordering* relation, $\xrightarrow{T}$, shows the relative order in which events execute: $a \xrightarrow{T} b$ means that a executes before b. We assume that the underlying memory system provides sequential consistency, which ensures that all shared-memory references appear as if they execute in some total order. Since we view events as the individual shared-memory references, sequential consistency means that $\xrightarrow{T}$can be viewed as a total order.

The *shared-data dependence* relation, $\xrightarrow{D}$, shows when one event can causally affect another, either because of a direct or transitive data dependence involving shared data. A direct shared-data dependence from a to b (denoted $a \xrightarrow{DD} b$) exists if a accesses a shared variable that b later accesses (where at least one access modifies the variable); we also say that a direct dependence exists if a precedes b in the same process, since data can in general flow through non-shared variables local to the process. A transitive shared-data dependence ($a \xrightarrow{D} b$) exists if there is a chain of direct dependences from a to b; $\xrightarrow{D} = (\xrightarrow{DD})^+$, the irreflexive transitive closure of the direct dependences. The dynamic dependence graph shown in Figure 1a is a graphical representation of $\xrightarrow{D}$.

4. Problem Statement

Section 2 illustrated that tracing each shared-data dependence provides sufficient information for replay but is not necessary. In this section we formally define what is necessary by characterizing *frontier races* to show the subset of dependences that must be traced. In Section 5 we present and prove correct an algorithm to detect and trace frontier races on-the-fly.

To support trace and replay, enough information must be traced during the initial execution to ensure that a replay on the same input is always identical. Our goal is to trace enough information to reproduce the execution in exact detail. As discussed in Section 2, by forcing replay to exhibit the same shared-data dependences as the initial execution, the same execution will result[†]. We will thus consider a replay to be successful iff it exactly reproduces the execution's shared-data dependences.

Definition 1

Given two program executions, $P = \langle E, \xrightarrow{T}, \xrightarrow{D} \rangle$ and $P' = \langle E', \xrightarrow{T'}, \xrightarrow{D'} \rangle$, P' is a *successful replay* of P iff

(1) $E' = E$, and
(2) $\xrightarrow{D'} = \xrightarrow{D}$.

For debugging, replays that do not exactly reproduce $\xrightarrow{D}$ but still cause the same events to execute may be useful (e.g., if we replay an execution that contains only writes to shared memory, it may not matter in what order we re-execute those writes). Investigating such replays is left to future work.

Our goal is to trace just enough information to ensure that the dependences can be reproduced. Although we could trace all dependences, we only need trace dependences that comprise *frontier races*[‡].

Definition 2

Given a program execution, $P = \langle E, \xrightarrow{T}, \xrightarrow{D} \rangle$, a *frontier race* $\langle a, b \rangle$ exists between two events a and b iff

(1) a and b belong to different processes, and
(2) $a \xrightarrow{D/} b$, where $\xrightarrow{D/}$ is the transitive reduction of $\xrightarrow{D}$.

As shown in Figure 1b, frontier races are those inter-process shared-data dependences that make up the transitive reduction of $\xrightarrow{D}$. Note that $\xrightarrow{D}$ also has intra-process dependences (indicating the order in which events in the same process execute). These dependences need not be traced since reproducing the inter-process dependences causes each process to follow its original control flow, thus reproducing the intra-process order.

One of our main results is that forcing the frontier races to be resolved in the same way during replay as during execution is both sufficient and necessary for successful replay. To resolve a frontier race, $\langle a, b \rangle$, we add syn-

† All replay schemes also require interactions with the external environment to be reproduced, and we do not discuss this issue here.

‡ We use the term *frontier race* since these races are also exactly those event pairs below which a consistent frontier (or cut) can be drawn. Choi and Min[2] define something called a *Race Frontier* (which is a consistent cut drawn above all detected data races), but for a purpose different from our frontier races.

chronization between events during replay so that a and b are replayed in their original execution order. Enforcing every frontier race in this way is sufficient to ensure that *all* dependences (even those that do not race) are corrected enforced. The following theorem also shows that if any frontier races are left unresolved, then replay will not always be successful (proofs appear in the Appendix).

Theorem 1 (Replay Theorem)

Replay will be successful iff the frontier races are resolved in the same order during replay as during the original execution.

5. Adaptive Tracing Algorithm

Our strategy is to dynamically locate and trace the shared-memory references that comprise the frontier races. Below we sketch our optimal tracing algorithm, based on Dinning and Schonberg's on-the-fly data race detection algorithm[3] suitably modified to locate frontier races. The optimal algorithm exactly detects the frontier races. We also discuss optimizations that tradeoff race detection accuracy (and thus non-minimal traces) with space overhead. Section 6 presents experimental results showing that the optimal algorithm and its variants usually trace only $0.01 - 2\%$ of the shared-memory references.

5.1. Optimal Algorithm

Figure 2 shows the optimal version of our algorithm (Figure 2a race checks writes; Figure 2b race checks reads). The algorithm works by dynamically maintaining a graph (like the one in Figure 1) that tracks the shared-data dependences which occur during execution. At each shared-memory reference, e, the algorithm first identifies the most recent event in each process on which e is data dependent (i.e., events x such that $x \xrightarrow{D} e$). The graph is then checked to determine if any of these dependences represent a transitive edge. A transitive dependence edge is not a frontier race and can be ignored; a non-transitive dependence is a frontier race and is traced (and the graph is updated to reflect the new edge).

To identify the events on which a reference is data dependent, we maintain an *access history*[3] for each shared variable S. An access history is a list of events, one per process, that most recently accessed S. Events are identified by <process #, serial #> pairs, and each process maintains a local counter with which it assigns serial numbers to its events. The access history consists of the *writer*, showing the last event that wrote S, and the *read-*

```
1a:  lookup S's readSet, readTs, writer, and writeTs;
2a:  /* Perform race check */
3a:  for (each event h in readSet)
4a:      if (h is unordered with e)
5a:          trace that ⟨h, e⟩ is a race;
6a:  if (races were detected)
7a:      ts = MAX(ts, readTs);
8a:  if (writer is unordered with e)
9a:      trace that ⟨writer, e⟩ is a race;
10a:     ts = MAX(ts, writeTs);
11a: /* Update access history */
12a: writeTs = ts;
13a: remove from readSet events ordered before e;
14a: writer = e;
```

(a): Algorithm for write events

```
1b:  lookup S's readSet, readTs, writer, and writeTs;
2b:  /* Perform race check */
3b:  if (writer is unordered with e)
4b:      trace that ⟨writer, e⟩ is a race;
5b:      ts = MAX(ts, writeTs);
6b:  /* Update access history */
7b:  readTs = MAX(ts, readTs);
8b:  remove from readSet set events ordered before e;
9b:  add e to readSet;
```

(b): Algorithm for read events

Figure 2. Adaptive tracing algorithms invoked after each event e (including synchronization) that accesses shared variable S.

Set, showing the last event in each process that read S. Because only the last read event in each process is remembered, the history requires $O(p)$ space (where p is the number of processes). In addition, readers on which the writer or other readers are data dependent also need not be remembered (since they can never form races).

To dynamically maintain the dependence graph, each process maintains a *vector timestamp*, a vector (of length p) of event serial numbers. Timestamps are a standard way of dynamically tracking causality, and they encode the current state of the dependence graph. The timestamp for a process q has an entry for every other process, and the i^{th} entry contains the serial number of the latest event in process i from which a path to process q currently exists. This timestamp can be viewed as a frontier (or cut) that divides the execution in half; at any point, all events above the frontier have paths to q, and all events below do not. Timestamps allow a constant-time determination of whether a dependence is transitive. If process q is about to perform an event e, and we find that a dependence exists from an event x in process i (i.e., $x \xrightarrow{\text{D}} e$), that dependence is transitive if x's serial number is less than or equal to the i^{th} entry of e's timestamp.

The algorithm performs these checks after every event e that accesses a shared variable S to determine if any events in S's access history race with e. An event h in the access history races with e if (1) h conflicts with e (the *readSet* is thus not checked in the algorithm for reads), and (2) h is *unordered* with e: no path from h to e in the dependence graph yet exists (determined by checking the process' timestamp as described above). A detected race means that an edge from h to e has been found (because they data conflict), and since no path previously connected them (because they are unordered), the new edge belongs to the transitive reduction and is thus traced.

Once a race is detected, the process' timestamp is updated to indicate the new edge from h to e, reflecting that all paths to h are now also paths to e. This update requires knowing h's timestamp when h executed. So this information is available when e is finally reached, in the access history we store the *writer*'s timestamp (denoted *writeTs* in Figure 2), and a single timestamp for the *readSet* (denoted *readTs*). Only one timestamp is stored for the entire *readSet* since, after a race check involving a write, *all* readers will be ordered before the write (line 7a; each reader will either race with e or will already be ordered before e). Thus, *readTs* is simply the maximum of the timestamps of each reader in *readSet* (updated in line 7b). To add the edge representing a race, we simply compute the component-wise maximum with these timestamps (lines 7a, 9a, and 5b). After the algorithm finishes, the timestamp shows a frontier above which are all events with a shared-data dependence to e.

Figure 3 shows a race check performed for the write "S=1" in P5. *readTs* shows all events ordered before the readers. After the races with the readers are detected, P5's timestamp is updated (by taking a component-wise maximum with *readTs*) to indicate that all readers are now ordered before "S=1". In general, the timestamps are updated so that they order a before b iff $a \xrightarrow{\text{D}} b$.

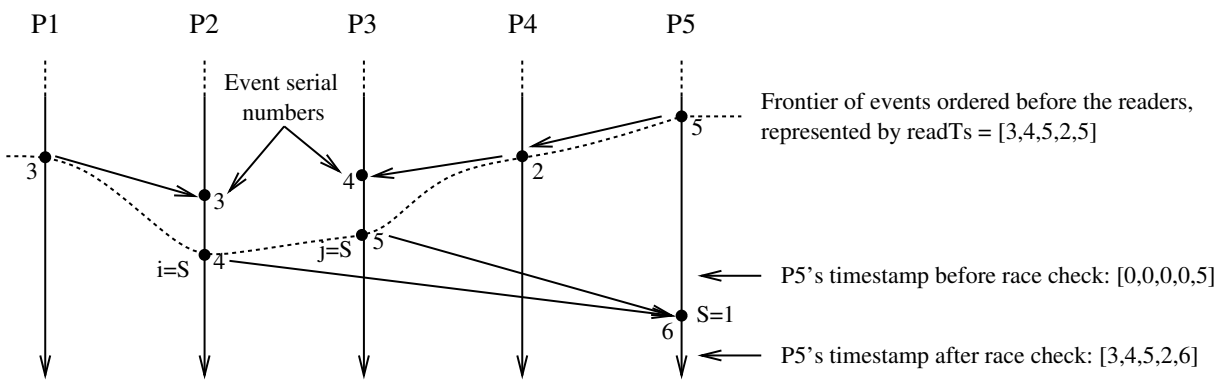


Figure 3. Race check performed at "S=1" in P5. After the race check, P5's timestamp is updated to show all events with a path to "S=1" (events on which "S=1" is shared-data dependent).

The main difference between our algorithm and Dinning and Schonberg's data race detection algorithm is that we update a process' timestamp whenever it is involved in a frontier race, as opposed to whenever it synchronizes. This update reflects the synchronization that will be added to the execution during replay to enforce the data dependence, and allows the transitive reduction of the dependences to be correctly identified. In addition, an advantage of our algorithm is that synchronization is not specially handled; by treating it as a memory reference, races between synchronization operations will be detected and traced as needed. Also, a subtly of our algorithm is that a write is checked for a race with the *readSet* before the *writer*. If races with any readers are found, the added edges (in line 7a) prevent a possible transitive dependence from the *writer* from falsely being detected as a race. Theorem 2 in the Appendix proves that the algorithm exactly identifies the dependences in the transitive reduction.

5.2. Optimizations

Our algorithm shares the drawbacks of all on-the-fly race detection algorithms: possibly large access histories, and time-consuming race checks. Several variations of our algorithm can reduce these costs.

Access histories require $O(mp)$ total storage, where m is the number of shared variables referenced during execution. This total can be large if many processes or variables are used. To reduce space overhead, we can use *approximate* access histories by maintaining them at a larger granularity, keeping one history for a large aggregate of shared data, instead of one history for each variable. Arbitrary space reductions are possible by making the aggregates as large as desired. Aggregate histories generally tradeoff space for race detection accuracy. Since aggregates introduce false dependences, the traces are still sufficient for replay, but false races can be detected which lead to non-minimal traces. Interestingly, as discussed later, false dependences can sometimes decrease the trace size. However, Section 6 shows that aggregates do not significantly increase trace size. The main drawback is that contention for the aggregates is greater than for individual histories. To be effective, updates to histories must be made as quick as possible.

Since race checks are done at each shared-memory reference, they must be kept fast. A simple way to speed our algorithm is to perform a race check *only* when the last conflicting access to S was made in a different process than the current access. By keeping track of the last access, we can determine when a race check is unnecessary. Experiments on the test programs described in Section 6 indicate that with this optimization only $2-20\%$ of the shared-memory references usually need be checked for races.

6. Performance

To test our tracing algorithm's effectiveness, we analyzed executions of shared-memory parallel programs on the Sequent Symmetry multiprocessor. To compare the effects of varying the access history granularity, we simulated our algorithm from an execution trace, instead of running it on-line (the programs' nondeterminacy would cause multiple runs to differ). We simulated both completely accurate access histories (one history per shared variable) and a granularity of 16 bytes (one access history for every 16 bytes of shared data), giving approximate race detection. Only $0.01-2\%$ of the references were usually traced. Using aggregate histories reduced space overhead but usually did not increase trace size substantially; in some cases, artificial dependences introduced by the inaccurate race detection *decreased* trace size. Below we discuss these results and argue that our algorithm's run-time overhead should usually be low. Our strategy should provide a new tool to debug long-running programs with lower overhead than previously possible.

Table 1 shows the results of using exact access histories. Three programs are from Stanford's Splash suite; others are from colleagues. Each was run on small inputs using two, four, and 16 processors. Table 1 compares the total number of shared-memory references (which is the number of trace records previous schemes emit) to the fraction of those references we trace. Usually only $0.02-2\%$ of the references were traced, a $2-4$ order of magnitude reduction. The large reductions result from the small number of shared writes in the programs, making most dependences non-races. Good program performance on shared-memory machines requires limiting the amount of data sharing (programs that frequently share data run slowly). We thus expect these results to be typical. In the extreme cases, *gauss* and *pnet* sometimes required tracing over 10% of the references; these programs were written by naive programmers and performed frequent fine-grain data sharing between processes (resulting in many frontier races). In addition, trace size increased with more processors. It is unclear whether this trend is an artifact of fixing the input size while increasing the number of processes. More experimentation is needed to investigate this issue.

Program	# of Traces / # of References (% of References Traced)					
	2 processors		4 processors		16 processors	
barnes	201 / 66887	(0.3%)	407 / 66004	(0.6%)	1475 / 79426	(1.9%)
gauss	20806 / 162023	(13%)	22362 / 162719	(14%)	33636 / 185761	(18%)
gcd	3764 / 478675	(0.8%)	732701 / 525765	(1.4%)	1779046 / 822719	(2.2%)
join	61 / 35672	(0.2%)	287 / 35831	(0.8%)	862 / 36407	(2.4%)
locus	11257 / 1932827	(0.6%)	18935 / 1996153	(0.9%)	88750 / 2530803	(3.5%)
mp3d	625 / 127041	(0.5%)	1293 / 127675	(1.0%)	3730 / 129568	(2.9%)
mtxmult	2453 / 400209	(0.6%)	7207 / 400209	(1.8%)	28894 / 400209	(7.2%)
pnet	2570 / 104298	(2.5%)	7272 / 139022	(5.2%)	26687 / 190854	(14%)
ptycho	102 / 112372	(0.09%)	762 / 336690	(0.2%)	8415 / 1188122	(0.7%)
quicksort	22 / 11040	(0.2%)	452 / 22872	(2.0%)	3377 / 86331	(3.9%)
shpath	8973 / 3760213	(0.2%)	1687484 / 4212118	(0.4%)	891089 / 61496004	(1.4%)
sort	3 / 14846	(0.02%)	17 / 29278	(0.06%)	287 / 109608	(2.6%)

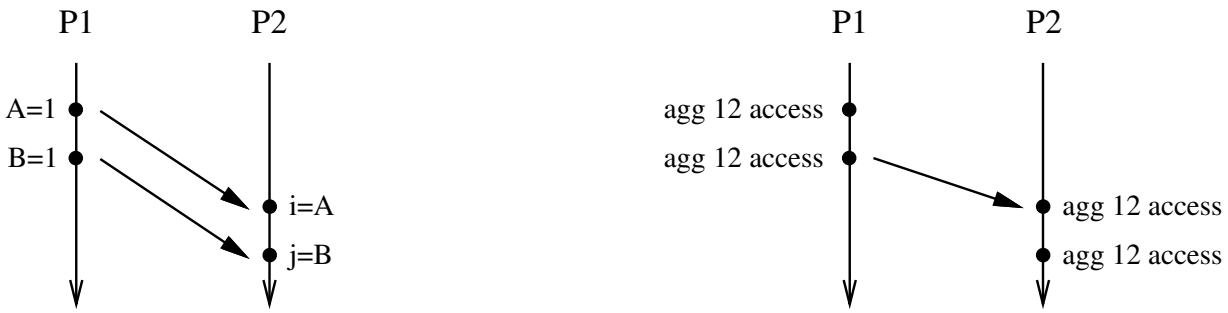
Table 1. Performance of algorithm with exact access histories

Table 2 compares exact access histories with approximate histories for two-processor executions of the programs (the four- and 16-processor runs had similar results). The ‘‘Exact’’ columns show the trace size and the algorithm's space overhead when one access history per shared variable was used. The last two columns show the effects of maintaining access histories for 16-byte aggregates of shared data (instead of one access history per shared variable). The aggregate histories were formed by bit-shifting addresses right four bits before the access history lookup (lines 1a and 1b in Figure 2). In all cases the space overhead was reduced. In five programs, the trace size increased only slightly, and in two programs no change occurred. However, in five programs the trace size *decreased*. As discussed earlier, when using exact histories the algorithm computes a minimal trace assuming

Program	# of References	Exact Access Histories		16-byte Aggregates	
		# Traces	Access Hist Size (bytes)	# Traces	Access Hist Size (bytes)
barnes	66887	201 (0.3%)	39880	347 (0.5%)	18368
gauss	162023	20806 (13%)	20896	20806 (13%)	10552
gcd	478675	3764 (0.8%)	39120	3474 (0.7%)	12992
join	35672	61 (0.2%)	39120	78 (0.2%)	27272
locus	1932827	11257 (0.6%)	622504	6897 (0.4%)	193544
mp3d	127041	625 (0.5%)	279072	452 (0.4%)	80208
mtxmult	400209	2453 (0.6%)	60032	1227 (0.3%)	15016
pnet	104298	2570 (2.5%)	17088	4854 (4.7%)	13504
ptycho	112372	102 (0.09%)	171848	109 (0.1%)	46064
quicksort	11040	22 (0.2%)	2824	22 (0.2%)	736
shpath	3760213	8973 (0.2%)	3936952	16771 (0.4%)	984264
sort	14846	3 (0.02%)	3168	2 (0.01%)	1240

Table 2. Performance of algorithm with aggregate access histories

that we wish to *exactly* reproduce the original shared-data dependences. Approximate histories introduce artificial dependences between events with no actual shared-data dependence. Although these dependences do not affect the replay's correctness (but they may slow replay compared to the original execution), they can sometimes render some of the real frontier races redundant. As shown below, a trace of two frontier races can be replaced by a trace of one artificial dependence.



For example, when aggregates are used, the references to ''A'' and ''B'' might both map to ''aggregate 12'' and then become indistinguishable, resulting in only one frontier race, not two. Investigating how artificial dependences can be used to reduce trace size is left to future work.

Although we did not measure the run-time overhead of our algorithm, we expect it to perform well. As discussed earlier, a race check is needed only when the current access is in a *different* process than the previous conflicting access. This information can be stored as one additional entry to the access history. In 10 of the programs, only $2-20\%$ of the references required race checks (and 49% of *gauss* and 60% of *shpath* required checks). In addition, one of the most time consuming parts of the check is the timestamp update after a race is detected. Since races are detected infrequently, the common cases consist of no race at all, or a simple check of the access history (which is a constant-time operation for read events, which typically occur more frequently than writes).

7. Conclusions and Future Work

Our tracing strategy fights the tracing bottleneck by making run-time decisions to adaptively decide what to trace. Fast processors are making adaptive strategies faster than writing more data than absolutely necessary to slow disks. Because our algorithm performs best when programs have few shared writes, it typically traces only $0.01-2\%$ of an execution's shared-memory references. Since in the common case a race check is not required, and since the full cost of our algorithm is incurred only when races are detected, we expect its run-time overhead to be acceptable.

Future work includes investigating how to best use aggregate access histories to keep space overhead low. By carefully aggregating histories to minimize false sharing, space overhead can be kept low with a small increase in trace size and access history contention. In addition, the artificial dependences introduced by the resulting inaccurate race detection can sometimes be profitably used. By purposely adding such dependences when they will not slow replay (i.e., will not be on the replay's critical path), trace size can be further reduced.

Acknowledgements

We thank John Mellor-Crummey (Rice Univ.) for comments on the draft, Michal Young (Purdue) and Rob Strom (IBM) for discussions, and Jim Larus (Univ. of Wisconsin–Madison) for some of the test programs.

References

[1] Richard H. Carver and Kuo-Chung Tai, ''Reproducible Testing of Concurrent Programs Based on Shared Variables,'' *6th Intl. Conf. on Distributed Computing Systems*, pp. 428-432 Boston, MA, (May 1986).

[2] Jong-Deok Choi and Sang Lyul Min, ''Race Frontier: Reproducing Data Races in Parallel Program Debugging,'' *ACM Symp. on Princ. and Practice of Parallel Prog.*, pp. 145-154 Williamsburg, VA, (Apr 1991).

[3] A. Dinning and E. Schonberg, ''An Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection,'' *ACM Symp. on Princ. and Practice of Parallel Prog.*, pp. 1-10 Seattle, WA, (Mar 1990).

[4] Thomas J. LeBlanc and John M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay," *IEEE Trans. on Computers* **C-36**(4) pp. 471-482 (April 1987).

[5] John M. Mellor-Crummey, "Debugging and Analysis of Large-Scale Parallel Programs," *Ph.D. Thesis, also available as Computer Science Dept. Tech. Rep. 312*, Univ. of Rochester, (September 1989).

[6] Robert H. B. Netzer and Barton P. Miller, "Improving the Accuracy of Data Race Detection," *3rd ACM Symp. on Princ. and Practice of Parallel Prog.*, pp. 133-144 Williamsburg, VA, (Apr 1991).

[7] Robert H.B. Netzer and Barton P. Miller, "Optimal Tracing and Replay for Debugging Message-Passing Parallel Programs," *Supercomputing '92*, pp. 502-511 Minneapolis, MN, (November 1992).

[8] Robert H.B. Netzer and Jian Xu, "Adaptive Message Logging for Incremental Replay of Message-Passing Programs," *Supercomputing '93*, Portland, OR, (November 1993).

[9] Douglas Z. Pan and Mark A. Linton, "Supporting Reverse Execution of Parallel Programs," *SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pp. 124-129 Madison, WI, (May 1988).

[10] K. C. Tai, Richard H. Carver, and Evelyn E. Obaid, "Debugging Concurrent Ada Programs by Deterministic Execution," *IEEE Trans. on Software Engineering* **17**(1) pp. 45-63 (January 1991).

Appendix. Proofs of Theorems

Theorem 1 (Replay Theorem)

Replay will be successful iff the frontier races are resolved in the same order during replay as during the original execution.

Proof.

Let $P = \langle E, \xrightarrow{T}, \xrightarrow{D} \rangle$ be the original execution and $P' = \langle E', \xrightarrow{T'}, \xrightarrow{D'} \rangle$ be the replay.

If Part. To establish a contradiction, assume that all frontier races are resolved in their original order but replay is not successful. That is, assume that P is frontier-race free but P and P' differ. Let $a \xrightarrow{D} b$ be the first dependence where they differ (if there is more than one such dependence, select any one). That is, $x \xrightarrow{D} y \Leftrightarrow x \xrightarrow{D'} y$ for all x, y such that $x \xrightarrow{D} a \vee y \xrightarrow{D} a$. The events a and b must belong to different processes, but since they do not form a frontier race (by the assumption), another event c must exist such that $a \xrightarrow{D} c \xrightarrow{D} b$ (to cause $a \xrightarrow{D} b$ to be a transitive dependence). But, the above implies that $a \xrightarrow{D'} c \xrightarrow{D'} b$, or $a \xrightarrow{D'} b$, contradicting the assumption that $b \xrightarrow{D'} a$. Thus, there cannot exist an event c that makes $a \xrightarrow{D} b$ a transitive dependence, so it must be a frontier race. P and P' cannot therefore be different, so replay is successful.

Only-If Part. Trivial. If replay is successful, then P and P' are identical, so in P' all frontier races are resolved in the same order as in P. QED

Theorem 2 (Tracing Algorithm Correctness Theorem)

For any two data-conflicting events, a and b, belonging to different processes, Algorithm 1 traces that a frontier race exists between a and b exactly when $a \xrightarrow{D/} b$.

Proof.

We first make two observations (without proof) about Algorithm 1.

Observation 1. Algorithm 1 finds a ordered before b iff $a \xrightarrow{D} b$, by the way reader and writer timestamps are updated (in lines 12a and 7b) and processes timestamps are updated (lines 7a, 10a, and 5b).

Observation 2. Algorithm 1 detects and traces a race $\langle a, b \rangle$ exactly when it finds (1) a not ordered before b and (2) a in the access history, when b is finally reached.

Given these observations, we use induction on the number of shared-memory references made between a and b (recall that in Section 3 the assumption of sequential consistency ensures that $\xrightarrow{T}$totally orders all references). Assume that we have n memory references between a and b; i.e., $a \xrightarrow{T} x_1 \xrightarrow{T} x_2 \xrightarrow{T} \cdots \xrightarrow{T} x_n \xrightarrow{T} b$ $(n \geq 0)$.

Basis (n=0,1).
$\Rightarrow$: $n = 0$: if $a \xrightarrow{D/} b$, then a and b conflict, and Algorithm 1 clearly traces the existence of $\langle a, b \rangle$ (since there is no intervening reference to order a and b or to remove a from the access history).

$n = 1$: if $a \xrightarrow{\text{D/}} b$, then x_1 cannot conflict with a or b, so this case reduces to the case of $n = 0$.

$\Leftarrow$: $n = 0$: impossible, since we cannot have $a \xrightarrow{\text{D/}\not\rightarrow} b$ when a and b conflict.

$n = 1$: if $a \xrightarrow{\text{D/}\not\rightarrow} b$, then we must have $a \xrightarrow{\text{D/}} x_1 \xrightarrow{\text{D/}} b$. We show that in this case the algorithm does not trace $\langle a, b\rangle$.

Case 1. x_1 is in the same process as a. Since a is ordered before x_1, the algorithm removes a from the reader or writer set when x_1 is processed, so when b is eventually reached, a is not in the access history.

Case 2. x_1 is in the same process as b. Since x_1 conflicts with a, by the first $n = 0$ case above, the algorithm traces $\langle a, x_1\rangle$. After this race is detected, timestamps are updated to order a before x_1. Since x_1 is ordered before b (x_1 precedes b in the same process), the algorithm finds a ordered before b.

Case 3. x_1 is in a different process than a or b (and conflicts with both). By the first $n = 0$ case above, $\langle a, x_1\rangle$ and $\langle x_1, b\rangle$ are both detected and traced. There are four sub-cases:

> *Subcase 1.* a is a read, x_1 is a write, and b is a write. After detecting $\langle a, x_1\rangle$, a is removed from the reader set, so the algorithm will not detect $\langle a, b\rangle$ as a race.
>
> *Subcase 2.* a is a write, x_1 is a write, and b is a read. After detecting $\langle a, x_1\rangle$, a is no longer the writer, so the algorithm will not detect $\langle a, b\rangle$ as a race.
>
> *Subcase 3.* a is a write, x_1 is a write, and b is a write. Same as subcase 2.
>
> *Subcase 4.* a is a write, x_1 is a read, and b is a write. After detecting $\langle a, x_1\rangle$, timestamps are updated to indicate that a is ordered before x_1. Then, because the algorithm checks for races with the reader set first, $\langle x_1, b\rangle$ is detected next, and timestamps will be updated again to indicate that x_1 is ordered before b. Thus, when a race check between a and b is performed, a is ordered before b, so $\langle a, b\rangle$ is not detected as a race.

Induction.
Assume that if we have n memory references between a and b, then $a \xrightarrow{\text{D/}} b$ exactly when the algorithm detects and traces $\langle a, b\rangle$. We will show that if we have $n + 1$ references between a and b, then $a \xrightarrow{\text{D/}} b$ exactly when the algorithm detects and traces $\langle a, b\rangle$.

$\Rightarrow$: We show that if $a \xrightarrow{\text{D/}} b$, then the algorithm traces $\langle a, b\rangle$. To establish a contradiction, assume that $\langle a, b\rangle$ is not traced. There are two cases that cause a pair of references to not be detected as a race.

Case 1: When b is reached, the algorithm finds a ordered before b. This ordering can exist only because of a sequence of races involving a, b, and perhaps some of $x_1 \cdots x_{n+1}$. That is, we must have $a \longrightarrow x_a \longrightarrow \cdots \longrightarrow x_i \longrightarrow b$ where $x_a \cdots x_i$ are some of $x_1 \cdots x_{n+1}$, and $\longrightarrow$ indicates either a race or an intra-process ordering (although at least one $\longrightarrow$ must be a race since a and b belong to different processes). By the inductive hypothesis, if a race $\langle y, z\rangle$ exists between any of these events, then $y \xrightarrow{\text{D/}} z$. Thus, there is a sequence of shared-data dependences between a and b, so $a \xrightarrow{\text{D}} b$ must be a transitive dependence, contradicting the assumption that $a \xrightarrow{\text{D/}} b$.

Case 2: When b is reached, the algorithm finds that a is unordered with b, but a is no longer in the access history (so no race is detected). By the inductive hypothesis, a is in the access history after x_n is processed, so it must be removed after x_{n+1} is processed (implying that a and x_{n+1} conflict).

> *Subcase 1.* a is a write. x_{n+1} must also be a write (or else a would not be removed from the access history), so $x_{n+1} \xrightarrow{\text{D/}} b$. But since $a \xrightarrow{\text{D}} x_{n+1}$, this contradicts the assumption that $a \xrightarrow{\text{D/}} b$.
>
> *Subcase 2.* a is a read. After the algorithm processes x_{n+1}, it must find a ordered before x_{n+1} for a to be removed from the reader set. By Observation 1, if the algorithm finds a ordered before x_{n+1}, then $a \xrightarrow{\text{D}} x_{n+1}$. In addition, b must be a write (since $a \xrightarrow{\text{D/}} b$) and $x_{n+1} \xrightarrow{\text{D/}} b$, contradicting the assumption that $a \xrightarrow{\text{D/}} b$.

$\Leftarrow$: We show that if $a \xrightarrow{\text{D/}\not\rightarrow} b$ then the algorithm does not detect and trace $\langle a, b\rangle$. By the inductive hypothesis, after the algorithm processes x_n, either a is not in the access history, or a is ordered before b. After processing x_{n+1} these conditions can not change. If a is not in the access history after x_n, it will never rejoin it again. If a is ordered before b after processing x_n, it will still be ordered before b after x_{n+1}. Thus, the algorithm will not detect $\langle a, b\rangle$. QED