BROWN UNIVERSITY
Department of Computer Science
Master's Project

CS-95-M12

"The Performance of Various Tracing Algorithms for Shared-Memory Parallel Programs"

by

Eric Stradal

The Performance of Various Tracing Algorithms for Shared-Memory Parallel Programs

Eric Stradal

Department of Computer Science
Brown University
Providence, Rhode Island 02912

October 1994

This research project by Eric S. Stradal is accepted in its present form by the
Department of Computer Science at Brown University
in partial fulfillment of the requirements for the
Degree of Master of Science.

October 1994

Date: 10/11/94

Robert H. B. Netzer

The Performance of Various Tracing Algorithms for Shared-Memory Parallel Replay

Eric Stradal
Department of Computer Science
Brown University
Providence, RI 12912-1910
ess@cs.brown.edu

October 6, 1994

Abstract

Cyclic debugging is a proven method for debugging sequential programs. In order to use the same techniques for parallel programs, special tools must be used to ensure the same data dependencies in successive runs. This is accomplished by tracing information about processor interaction during the original execution, and using that information during the replay phase. This paper investigates the performance of several adaptive tracing algorithms for parallel replay. The algorithms were measured for trace size and runtime overhead. It also attempts to make a judgment on which of the tracing schemes would be the most useful.

1 Introduction

In order to exploit the advantages of parallel computers, developers need an extensive set of debugging tools. Cyclic debugging has become an established procedure for eliminating errors in sequential programs. Similar methods for parallel programs are very desirable. The problem lies in the inherent nondeterminism of parallel programs. Special tools are required to ensure the events in a re-execution occur in the same order as they did in the original execution. All the current replay schemes trace information about program execution while it is running. This information is later used to ensure proper ordering of events

upon re-execution. Such a system is commonly referred to as a *trace and replay* system [5]. The problem with most trace and replay systems is the large amount of data they trace. The focus of this paper is comparing the results of several different schemes that reduce the amount of information traced.

The non-determinacy encountered in shared memory parallel programs can be characterized in two ways. The first is general races. The second is data races [8]. General races are what is commonly referred to as non-determinism. They are only considered to be bugs in programs which are supposed to be deterministic. General races are a global property of a program. Most parallel applications don't need to be deterministic, and contain general races. In fact, the ability to step beyond the limitation of linearity is one of the things that makes parallel computation so attractive. In a program meant to be non-deterministic, general races are not considered bugs. Data races are the result of non-atomic execution of critical sections. Data races are a local property in a program, and are usually caused by insufficient or improper synchronization. Data races have also been called access anomalies, and are always considered to be bugs, because the interleaving of the instructions in the critical sections can cause spurious data to be produced. Synchronization bugs generally manifest themselves as data races, although they can also cause general races.

In either case, these races are at the root of the problems in trace and replay systems. The schemes to reduce trace size concentrate on only tracing events that are racing with each other, thereby tracing only information which effects the order of events which are dependent on each other.

All of the algorithms implemented in this paper are for shared memory parallel programs. It should be noted that many tracing techniques can be generalized to also work for message passing programs [2]. The differences lie in how one looks at communication between processors. An access to a shared memory object is like a message from, or to, a data object in a specific processors memory in a message passing system. Although the difference looks marginal it

is theoretically significant. In particular, Netzer's optimal algorithm is optimal for shared memory programs, but is not optimal in all cases for message passing programs [7].

1.1 Previous Work

The need for a trace and replay system has been seen by other researchers. Two particularly interesting implementations are Bugnet [9] and Instant Replay [2].

Although Bugnet was implemented in a message passing environment, it is interesting because it traces the order and *content* of all inter-process communication. By recording the data and order of events, Bugnet can replay the original program, or just a single process. It can simulate the actions of the other processes, allowing a debugger to isolate processes. The obvious drawback is the extremely large traces, which would limit the effectiveness of any system which records data and ordering information.

Instant Replay is notable because it introduced the notion that a trace and replay system could be implemented tracing only the ordering between inter-process events. It was developed on a shared memory system, but the same type system could be used for message passing systems. During an execution, each access to a shared data object is noted, and its process, object and version are recorded. The replay phase forces events to happen in the same relative order as the original. Since, acting alone, each process is deterministic, forcing communications to occur in the same order will exactly reproduce the original execution. By only recording the relative order of accesses the system does not require synchronized clocks or globally consistent time.

There are a couple of drawbacks to this system. Because all memory accesses are recorded, trace sizes are large. The system does not record fine grained data access. All accesses are treated as messages to shared objects. The system institutes a CREW protocol for the shared objects, although the system can also work with a mutually exclusive protocol. This means programs have to

be written with the protocol in mind, or they must be modified to match it. One effect of this is that the overhead for tracing looks small, because some of the synchronization required for tracing is enforced in the CREW model, hiding tracing delays which are apparent in the algorithms in this paper.

2 Model

Much of the discussion of the algorithms presented here rely on several different models of time and the temporal relationship between events.

When one is talking of orderings between general events, it is most natural to think in terms of real, physical, time. It would be most advantageous if one could think along the same lines when working with parallel computers. The problem is that it is non-trivial to have a consistent global clock on a parallel processor [1]. Generally, each processor runs off its own clock. This is the root of asynchronous behavior. There may be a global clock, but there are problems with races to the clock, i.e. two process could try to get a clock value at the exact same instant, and not get consistent values. This can be caused by many things, such as differences in delay time for communications. Even in multi-processors with a single global clock there are problems, such as increased bus load from all the processors accessing resources at the same instant. Thus, a different notion of time must be used when talking of parallel programs.

> **The Temporal Relation**: For two events a, b: if $a \xrightarrow{T} b$, then a occurs before b. Where $\xrightarrow{T}$ denotes the temporal ordering relation.

Logical clocks are a way of assigning a number to an event [1]. The main condition they satisfy is the following. If event a happens before event b, then the clock value assigned to a is less than that assigned to b. The numbers don't necessarily reflect physical time, other than that lower clock values happened before larger clock values. For most purposes they can be implemented with counters. Logical clocks bypass many of the problems associated with physical

clocks, but don't reflect the ordering of events in separate processes. For that, more specialized clocks are needed.

A commonly used form of ordering parallel programs is vector clocks, or timestamps [8]. A vector clock is an abstraction which orders events in parallel programs. The vector clock reflects the inter-process communications which have taken place. Each process maintains a serial number, (logical clock) which is incremented at each event. Each process maintains its own vector clock, which has one entry per processor. Entry i in the vector clock contains the most recent serial number from process i which has communicated with the current process. Of course, the i^{th} entry for processor i simply contains the processes current serial number.

> **Vector Clock Condition**: For any events a_i, b_j:
> $VC_{i,j}(a) < VC_{i,j}(b)$ if and only if $a_i \xrightarrow{V} b_j$.
> Where i and j are the relative processes of a and b, and $a_i \xrightarrow{V} b_j$ is the vector clock temporal relation signifying that event a in process i occurred before event b in process j, and $VC_{i,j}(a)$ signifies the vector clock serial number of event a in the i^{th} index of the vector clock for process j.

Timestamps are useful because they partition the events of an execution into two sets, those that are known to have happened before the current event, i.e. its serial number is less than its processors entry in the vector clock, and those that didn't necessarily happen before the current event. Another way of viewing vector clocks is seeing that they maintain a graph of the execution, with one node per process. The i^{th} entry in the vector clock contain the serial number of the most recent event in process i that has a path to the current process. This graph is what is referred to in the rest of the paper as the *execution graph.*

Lamport clocks [1] are another abstraction that attempt to order the events in a parallel program. They were first developed to order events in systems of

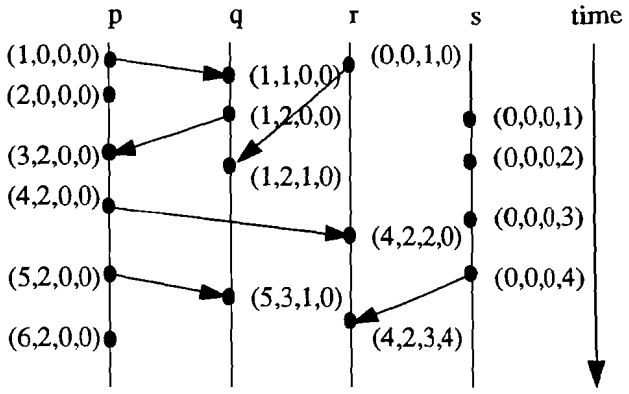

Figure 1: Vector Clock Propagation

spatially separated computers, but the notions can be generalized for parallel processing computers. Like vector clocks, Lamport clocks are based on logical clocks. Unlike vector clocks, each processor only maintains a single number to represent its Lamport clock. To order the events, Lamport clocks maintain the following condition.

> **Lamport Clock Condition**: For any events a_i, b_j: if $a_i \xrightarrow{L} b_j$ then $LC_i(a) < LCj(b)$. Where a_i signifies event a in process i, and $LC_i(a)$ signifies the Lamport Clock value for event a in processor i.

Since $\xrightarrow{L}$ and $\xrightarrow{V}$ impose a temporal ordering on events, they both can be used in place of the general temporal relation $\xrightarrow{T}$ in the following sections. In fact, this one of the differences in the algorithms presented in this paper.

Maintaining the Lamport clock condition if fairly simple. The two following conditions must be followed. If a and b are events in process i, and a occurs before b, then $LC(a) < LC(b)$. If event a in process i receives a message from event b in process j and $LC_i(a) < LC_j(b)$ then $LC_i(a) = LC_j(b) + 1$. Where $LC_i(a)$ is the Lamport clock value of event a in process i. It may be helpful to view Lamport clocks as vector clocks which contain the maximal serial number across the array. Seen in this way, it is clear that a Lamport clock partitions the

events along a line that runs flat across the execution graph of the processes.

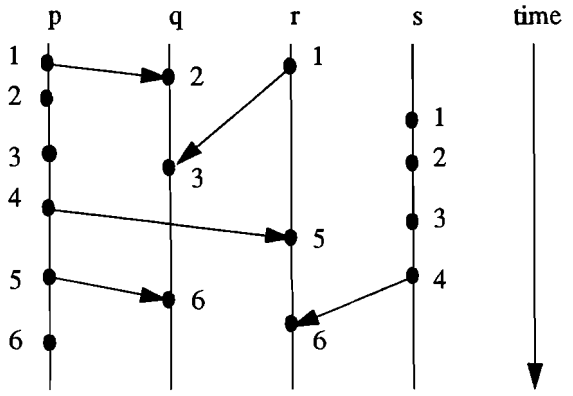

Figure 2: Lamport Clock Propagation

By examining the properties for vector clocks and Lamport clocks, it can be seen that Lamport clocks order more events than vector clocks. Stated simply, if one event is ordered before another by the vector clock condition, it will also be ordered be the Lamport clock condition. The converse is not true. There are events that are ordered by the Lamport clock condition that are not ordered by the vector clock condition.

Another key concept is data dependence [8]. Two events can only have a data dependency if they both access the same variable. Clearly, if they don't, one can not effect the other. In its weakest form, a data dependency simply states that events a and b are data dependent if a and b access the same variable. In its strong form, a data dependency means that events a and b are data dependent if a and b access the same variable, and at least one of the events writes. This condition is valid because, if both events simply look at the data, they can not effect the outcome of each other. In terms of the execution graph, data dependencies are the edges between processes.

This notion of data dependence can be combined with the temporal relations to form the data dependence relation [6]. The data dependence relation seeks to define when one event could be effected by another in the execution. The

determination of whether one event proceeds another is based on the clock condition used, and the determination of data dependence is based on the weak or strong form of data dependency. In the discussion on vector clocks and Lamport clocks, it was said that their values were updated when ever contact was made between processes. If two events are data dependent on each other, then they are considered to communicate with each other. In other words, the vector, or Lamport clock is only effected by events that are data dependent with each other.

> **The Data Dependence Relation, Strong Notion**: For any two events a, b: if $a \overset{D^S}{\rightarrow} b$, then a and b access the the same shared variable, at least one of them writes, and $a \overset{T}{\rightarrow} b$. Where $\overset{D^S}{\rightarrow}$ denotes the strong data dependence relation.

> **The Data Dependence Relation, Weak Notion**: For any two events a, b: if $a \overset{D^W}{\rightarrow} b$, then a and b access the same shared variable and $a \overset{T}{\rightarrow} b$. Where $\overset{D^W}{\rightarrow}$ denotes the weak data dependence relation.

When two processes are accessing the same variable, and their relative order can't be determined, the accesses are said to *race* to the variable [8]. Parallel programs generally contain many races. For example, a program that uses spin locks will have the processes racing to see which gets the lock and which are forced to spin. Similarly, any shared memory access, which isn't in a synchronized critical section, is the potential source for a race. Since these races are the source of non-determinacy, they are of prime importance in replay systems.

The line that partitions the execution graph, either for a vector or Lamport clock is called the *frontier* [8]. This is because, it is the frontier between events known to have happened and those which might not have happened yet. The frontier must represent a consistent state of the execution. The clock conditions ensure that they are. Vector clocks are able to represent any consistent state, but Lamport clocks can't [1]. This is because the frontier for a Lamport clock

cuts straight across the execution graph. A *frontier race* is an edge in the execution graph which crosses the frontier. In both cases, it is an event that violates the clock condition, and also contains a data dependence. The various algorithms presented in this paper all are based on the notion that in order to enforce proper ordering between processes, only the frontier races must be traced.

Critical Path: The critical path for a process is the transitive reduction of the data dependence graph for that process [6].

Artificial Dependency Edge: For the purposes of this work, an artificial dependency edge is an edge that is added to the execution graph that is not on the critical path. It represents a dependency that might not actually exist between two events [5].

3 Problem Statement

This section formally defines what is sufficient for correct replay. Enough information must be traced to insure that a replay on the same input is identical to the original execution. It has already been shown that tracing all shared-data access is sufficient [2], but not necessary for correct replay. If all the shared-data dependencies are the same in a replay as in the original, it will be correct. What is needed are schemes which trace a small subset of the data dependencies which is also sufficient for correct replay.

Sufficient Replay: Given two executions, $P = (E, \xrightarrow{T}, \xrightarrow{D})$ and $P' = (E', \xrightarrow{T'}, \xrightarrow{D'})$, P' is a sufficient replay if

1. $E' = E$, and
2. for all a, b element of E, $a \xrightarrow{D} b \Rightarrow a \xrightarrow{D'} b$.

Where E and E' are the events, $\xrightarrow{T}$ and $\xrightarrow{T'}$ are the temporal orderings between events, and $\xrightarrow{D}$ and $\xrightarrow{D'}$ are the data dependencies between

events in the executions P and P' [5].

This just states that the temporal orderings, the data dependencies, and the events are the same in the replay as in the original execution.

Four of the algorithms reproduce, exactly, the data dependencies of the original. This is because they only trace dependencies which are directly on the critical path, i.e. they only trace actual dependencies. The remaining two introduce artificial dependencies, which aren't on the critical path, to reduce trace size even further. See the explanation of the non-critical edge algorithms for more detail. Determining the minimum amount of information required to provide for a sufficient replay is an NP-hard problem [5].

4 Methods

4.1 Algorithms

All six of the algorithms explored here are based on maintaining a data dependence graph during execution. They only trace accesses whose data dependencies cause frontier races, and then update the graphs to include the new edges. The differences lie in how the dependence relation is defined, the type of clock used, and how a processes clock is updated when a race is found.

There might be some confusion as to why there are 3 versions of the 'optimal' algorithm. They are called optimal because they use the strong notion of data dependence. They never trace events which can't effect each other.

4.1.1 Optimal Algorithm Vector Clocks

This algorithm was first introduced in Optimal Tracing [6]. It dynamically locates and traces all the frontier races in an execution based on the strong data dependence relation, $a \xrightarrow{D^S} b$, and on the vector clock relation, $\xrightarrow{V}$. Each process maintains its own private counter, its *serial number*, which is incremented upon each shared memory access. Each shared variable has an access history. An

access history contains the timestamp of the most recent writer, and the timestamp of the most recent reader in each processor. Each access history requires $O(p)$ space, where p is the number of processors.

The dependence graph is maintained with vector timestamps. Upon each access to a shared variable, its access history is checked for frontier races with the current processor. If the access is a read, only the previous writers need to be examined, because two readers can not be data dependent. An event, h, in the access history is considered to be unordered with the current event e, if there is no path from h to e in the data dependence graph.

```
WRITE-RACE-CHECK-VECTOR-CLOCK(S)
1a:  increment the processes clock
2a:  get the access history for S
3a:  /* check for races */
4a:  for each event r in readset
5a:     if (r is unordered with e)
6a:        trace that < r, e > is a race
7a:  if any races were detected
8a:     timestamp = component-wise max of timestamp and readset
9a:  if (writer is unordered with e)
10a:     trace that <writer, e > is a race
11a:     timestamp = component-wise max of timestamp
                     and writer_timestamp
12a:  /* update access history */
13a:  writer_timestamp = timestamp
14a:  writer = e
15a:  remove any events in the readset that are ordered before e

READ-RACE-CHECK-VECTOR-CLOCK(S)
1b:  increment the processes clock
2b:  get the access history for S
3b:  /* check for races */
4b:  if (writer is unordered with e)
5b:     trace that <writer, e > is a race
6b:     timestamp = component-wise max of timestamp
                     and writer_timestamp
7b:  /* update access histories */
8b:  reader_timestamp = component-wise max of
                     timestamp and reader_timestamp
9b:  remove any events in the readset that are ordered before e
10b:  add e to readset
```

4.1.2 Compressed Access History Algorithm Vector Clocks

In order to reduce the memory overhead of the optimal tracing algorithm, this algorithm uses the weak data dependence relation, $a \xrightarrow{D^W} b$, based on the vector clock relation, $\xrightarrow{V}$. This has the effect of negating the difference between readers and writers. All accesses are treated the same. Therefore the algorithm no longer needs to keep track of a separate writer and readset. The access history for each shared object can be held in a single record, holding the last process and its clock value that accessed it. So, the access histories require $O(1)$ space.

```
RACE-CHECK-VECTOR-CLOCK(S)
1:  increment the processes clock
2:  get the access history for S
3:  /* check for races */
4:  if (last_access is unordered with e)
5:     trace that <last_access, e > is a race
6:     timestamp[last_proc] = last_access_clock
7:  /* update the access history */
8:  last_access = e
9:  last_access_clock = clock
10:  last_access_proc = proc
```

4.1.3 Optimal Algorithm adding Non-Critical Path Edges

This algorithm is a modification of the optimal, vector clock algorithm. When a frontier race is detected from a to b, an artificial dependence is created from b to the event that is currently in a's process. It has two main advantages over the original algorithm. First, it produces reduced trace size. This is because one artificial dependence can cut across several actual data dependencies which then are considered redundant, and don't need to be traced. The second advantage is that it doesn't create any orderings that didn't exist in the original execution. Therefore, there will be no more waiting during the replay as was encountered during the original execution.

```
WRITER-RACE-CHECK-NON-CRITICAL-EDGES(S)
1a:  increment the processes clock
2a:  get the access history for S
```

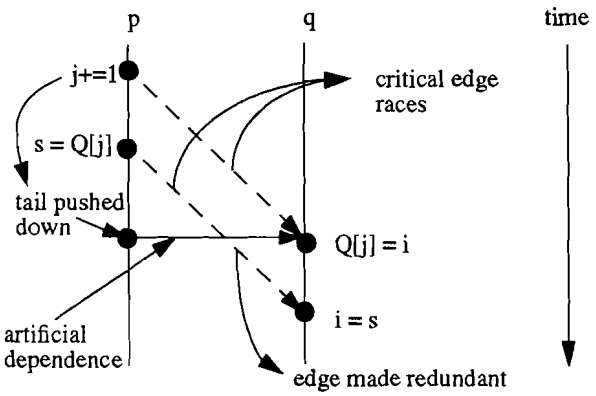

Figure 3: A non-critical path edge replaces 2 critical path edges.

```
3a:  /* check for races */
4a:  for each event r in readset
5a:    if (r is unordered with e)
6a:      trace that < r, e > is a race
7a:      timestamp = MAX(timestamp, timestamp of
                    current event in r's process)
8a:  if (writer is unordered with e)
9a:    trace that <writer, e > is a race
10a:     timestamp = MAX(timestamp, timestamp of
                    current event in writers process)
11a: /* update access history */
12a: writer_timestamp = timestamp
13a: writer = e
14a: remove any events in the readset that are ordered before e
```

READER-RACE-CHECK-NON-CRITICAL-EDGES(S)

```
1b:  increment the processes clock
2b:  get the access history for S
3b:  /* check for races */
4b:  if (writer is unordered with e)
5b:    trace that <writer, e > is a race
6b:    timestamp = MAX(timestamp, current
                    timestamp in writers process)
7b:  /* update access histories */
8b:  reader_timestamp = MAX(timestamp, reader_timestamp)
9b:  remove any events in the readset that are ordered before e
10b:  add e to readset
```

4.1.4 Compressed Access History Algorithm Adding Non-Critical Edges

This algorithm is a modification of the compressed history, vector clock algorithm. It has the same advantages that the optimal non-critical edge algorithm has.

```
RACE-CHECK-NON-CRITICAL-EDGES(S)
1:  increment the processes clock
2:  get the access history for S
3:  /* check for races */
4:  if (last_access is unordered with e)
5:     trace that <last_access, e > is a race
6:     timestamp = MAX(timestamp, timestamp of current
                       event in last_access_proc)
7:  /* update the access history */
8:  last_access = e
9:  last_access_clock = clock
10:  last_access_proc = proc
```

4.1.5 Optimal Algorithm Lamport Clocks

This algorithm is a modification of the optimal, vector clock algorithm, using the Lamport clock relation, $\xrightarrow{L}$, instead of the vector clock relation [3]. The Lamport clock relation is subtly changed so it says, event a in process i is ordered by event b in process j, if $LC_i(a) \leq LC_j(b)$. This change is made because only the races which cause the Lamport clock to be updated to a higher value need to be traced. Because Lamport clocks are a much simpler construct than vector clocks, this algorithm is simpler than the corresponding vector clock algorithm. One significant difference is that the clock isn't incremented until after the race check. This is needed to insure that the clock condition is maintained. That is, the Lamport clock value for the event must be larger than the value for any events on which it has a data dependency. When races are detected, the Lamport clock for the processor of the current event is updated to reflect the dependence on the item it races with. The access histories are also simpler. The last writer can be stored as just the processor and Lamport clock of the last

event that accessed the variable. Likewise, the read set need only contain the Lamport clock value of the last read event in each processor (that isn't ordered by the writer).

```
WRITER-RACE-CHECK-LAMPORT-CLOCKS(S)
1a:  get the access history for S
2a:  /* check for races */
3a:  for each event r in readset
4a:    if (r is unordered with e)
5a:      trace that < r, e > is a race
6a:  if races were detected
7a:    Lclock = the maximum Lclock value in the read set
8a:  if (writer is unordered with e)
9a:    trace that <writer, e > is a race
10a:     LClock = writer_clock
11a:   increment the processes LClock
12a:   // update access history
13a:   writer_clock = Lclock
14a:   writer_proc = proc
15a:   remove any events in the readset that are ordered before e

READER-RACE-CHECK-LAMPORT-CLOCKS(S)
1b:  get the access history for S
2b:  /* check for races */
3b:  if (writer is unordered with e)
4b:    trace that <writer, e > is a race
5b:    LClock = writer_clock
6b:  increment the processes Lclock
7b:  /* update access histories */
8b:  reader_timestamp(proc) = LClock
9b:  remove any events in the readset that are ordered before e
10b:  add e to readset
```

4.1.6 Compressed Access History Algorithm Lamport clocks

This algorithm is a modification of the compressed access history, vector clock algorithm. It shares the same advantages and disadvantages as the optimal, Lamport clock algorithm.

```
READER-RACE-CHECK-LAMPORT-CLOCKS(S)
1:  increment the processes clock
2:  get the access history for S
3:  /* check for races */
```

```
4:  if (last_access is unordered with e)
5:     trace that <last_access, e > is a race
6:     LClock = last_access_clock
7:  /* update the access history */
8:  last_access = e
9:  last_access_clock = LClock
10:  last_access_proc = proc
```

4.2 Implementation

The shared memory computer used in these experiments was a 16 processor Sequent Symmetry Machine. The Sequent is a shared-memory parallel processor, which uses multiple Intel 386 processors connected to a common system bus. Its operating system is a proprietary version of Unix.

In order to trace all accesses to shared variables, the compiler and parallel libraries were modified. Each instruction which could touch shared memory, and synchronization instructions, were replaced with hooks that sent control to a trace library. The trace library contains the code that identifies when a variable being accessed is in shared memory. The first scheme implemented was very simple. It simply traced all accesses which were in shared memory, and all synchronization events.

The other algorithms were implemented completely in the trace library. This had the advantage that once programs were compiled, they could be linked with different versions of the tracing library. Since this was an original attempt to implement these algorithms on a parallel computer, they were not highly optimized. For example, the access histories are held in a fairly simple hash table. Utilizing a more sophisticated hashing function would probably reduce overhead. The overheads that were generated can be looked at as a worst case.

It should be noted that since the adaptive algorithms were implemented to test their runtime overhead, the algorithms which add dependencies not on the critical path were not implemented. This is because they function similar to the original, vector clock versions of the algorithms. The estimated difference in runtime would be marginal. Thus, for purposes of testing overhead,

four algorithms were implemented, consisting of the combinations of optimal or compressed access histories, and vector or Lamport clocks.

In order to compare the relative trace sizes of the various algorithms, they were implemented as simulations that used data sets generated from tracing all shared memory accesses. This was done so each algorithm could be run on an equivalent execution. Otherwise, the non-determinacy of the parallel programs would cause any data about relative trace sizes to be suspect. The simulations were implemented on a Sun SparcStation 10.

5 Results

The tables contain the test name followed by a number. The number is the number of processors used. The test programs are:

barnes Parallel dynamics simulator

barnes2 Barnes with different (larger) input

gauss Gaussian elimination and back substitution

gcd Greatest common denominator

join Hash join algorithms

flow Multiplies, component-wise, two matrices

locus Parallel wiring router

mesh Solves a rectangular mesh problem

mp3d Wind tunnel simulator

mtxmult Parallel matrix multiply

pnet Parallel network flow simulator

ptycho Parallel cache simulation

qs Quick sort of an array of integers

shpath Dijkstra's shortest path determination

sort Merge sort

The results are represented in three tables. The first two contain the trace sizes from the simulation, and the last table contains the runtime overhead incurred on the Sequent. Since trace sizes were generated for all six algorithms, there is too much information to be put into one table. The first contains the results from the compressed access history algorithms and the second has the results from the optimal algorithms. The trace size tables show the program name, the number of processes used, and three pairs of numbers. Each pair contains the actual number of references that were 'traced', followed by its percentage of all the references in the execution.

The runtime overhead table shows the program name, the number of processors used, the runtime of the program without any instrumentation, and four numbers representing the runtime overhead for four of the algorithms. The overhead figures represent how many times slower the instrumented version was compared to the un-instrumented version. For example, if the runtime was two seconds, and the overhead given is 15.0, then the runtime of the instrumented code was 30.0 seconds. All timing figures were obtained with the **time** command and represent the average of five program executions. In all the programs, but two, the standard deviation of the runtimes was very small. This was surprising considering the non-determinacy these programs displayed. The two programs for which the instrumented versions had significant standard deviations were *flow* and *sort*. For *flow*, the standard deviation was quite high except for the optimal algorithms run on eight processors. For *sort*, the standard deviation was not as high as that for *flow*, and was small for the compressed algorithms run on four processors. It is worth noting that the two programs with the high standard deviation were also among the programs with the highest runtime

overhead. This connection was not explored.

Since the tables are quite large, here is a short list of points that were found to be interesting:

- In all cases but one (*sort* 2), the compressed access history, vector clock algorithm produced the largest traces.
- The Lamport clock algorithms always performed at least as well as the vector clock algorithms, and usually did significantly better.
- The non-critical edge algorithms always performed at least as well as the vector clock algorithms, and usually did significantly better.
- There seems to be little correspondence between the trace sizes from the non-critical edge algorithms and the Lamport clock algorithms.
- The optimal non-critical edge algorithm generated extremely small traces for *mtxmult*, whereas, the other algorithms performed about average.
- The compressed access history algorithms outperformed the optimal algorithms for *sort*. All of the other cases displayed the opposite, and expected behavior.
- The programs that generated the largest traces (percentage wise) were *gauss*, *locus*, *pnet*, and *shpath*.
- The optimal non-critical edge algorithm and the optimal Lamport clock algorithm nearly always traced less than one percent of references. And they never traced more than 5.12% (*pnet*) of references.
- The runtime overheads for *join* were unusually small.
- The overhead of the Lamport clock algorithms run a few percent faster than the corresponding vector clock algorithms (except on *flow* 8.)

- The compressed access history algorithms displayed 50-80% of the overhead of the optimal algorithms.
- The overhead figures are not linear with respect to the number of processors used.
- Adding processors nearly always resulted in an increase in the overhead (except for *join*.)
- Many programs had runtime overheads of less than 50, even when eight processors, and optimal algorithms, were used.
- The worst overhead figure was 125.57, from *sort* 8, with the optimal, vector clock algorithm.

6 Discussion

6.1 What was Learned from the Results

6.1.1 Trace Size

As was expected, the Lamport clock algorithms outperformed the vector clock algorithms. In fact, the respective clock conditions demand this. It was interesting that the Lamport clocks had nearly the same advantage whether the compressed access history, or optimal algorithms were used. The Lamport clocks order enough events so the compressed access history, Lamport clock algorithm outperforms the optimal, vector clock algorithm for some test programs. This means that the extra ordering imposed by the Lamport clocks outweighed the dependency information lost by the weak data dependence relation. The added ordering information of Lamport clocks combined with the strong notion of data dependence produced extremely small traces.

The non-critical edge algorithms also outperformed the corresponding vector clock algorithms, as expected. The interesting information was that the

optimal, non-critical edge algorithm displayed a greater advantage over the optimal, vector clock algorithm than the compressed access history, non-critical edge algorithm had over the compressed access history, vector clock algorithm. This is attributed to the fact that there are, usually, many more read-read races than either read-write or write-write races. The increased frequency of detected races, with the compressed access history algorithm, meant that the tail of the dependency edge could not be pushed down very far in the compressed access history, non-critical edge algorithm. The further the tails are pushed down, the more races can be made redundant. The compressed access history, non-critical edge algorithm displayed nearly the same performance as the compressed access history, vector clock algorithm for all the tests but one (*mtxmult*), where it had a significant advantage. This advantage was also evident, in all the test programs, with the optimal version of the non-critical edge algorithm. The strong notion of data dependence leads to fewer races being detected, which are temporally spread out. When the events, which are racing, are temporally far apart, the tail of the added artificial edge can be pushed far down the execution graph. These artificial edges can cross many critical path edges, which explains why the optimal, non-critical edge algorithm displays a greater advantage than the compressed access history algorithm.

There was an expected correspondence between the non-critical edge algorithms and the Lamport clock algorithms, which was only partially supported by the data. They were expected to have similar performance because they trace similar information. The non-critical edges are added straight across the execution graph, and the Lamport clock algorithms use frontiers that lie straight across the execution graph. The algorithms had similar performance for the optimal versions, but were very different for the compressed access history versions. In fact, the compressed access history, non-critical edge algorithm had trace sizes closer to the compressed access history, vector clock algorithm than the compressed access history, Lamport clock algorithm. The opposite was true

for the optimal algorithms, were the non-critical edge algorithm was closer to the Lamport clock algorithm.

The optimal algorithms nearly always produced many fewer traces than the compressed access history algorithms. The exceptions to this (*qs*, *sort*) had very few read-read races, which, when traced, caused other read-write, or write-write races to not be on the critical path, and not be traced. In general, there were far too many read-read races to get any advantage from this. In fact, for all the programs but *join*, *pnet*, *qs*, and *sort*, read-read races caused at least 50% of the traces, and usually were responsible for over 75% of the traces. All of the read-read traces are not needed, strictly speaking, to properly recreate the data dependencies during replay, so they can be seen as the overhead the compressed access history algorithms have over the optimal algorithms.

As was noted in the algorithms section, when a race is detected in a Lamport clock algorithm, the clock is 'bumped ahead' to be greater than that of the event it raced with. It was interesting to study the size of these clock 'skips'. The compressed access history algorithm concentrated most of the skips on very low numbers. Most of the programs had a large number of races where the Lamport clock was only bumped ahead one clock value. The optimal algorithms displayed a different behavior. The skips seemed much more evenly distributed, and there were very few instances where clocks only skipped one clock value. This indicated that the compressed access history, Lamport clock algorithm traced many read-read races, which occurred at nearly the same time. Another interesting point was that both algorithms had similar numbers of instances where the clock skipped more than 1000 clock values. These large skips show that races can occur between events that are temporally far apart in the execution. Also, those events are not read-read races, because they are captured by the optimal algorithm. This implies that there are very few read-read races from events which occur at very different times.

6.1.2 Runtime Overhead

Perhaps the most interesting data gained from these experiments was that tracing all the references was faster than using an adaptive algorithm. This can be attributed to several factors. First, the Sequent has a very efficient disk controller. It would buffer all the traces written and dump them onto the disk as each process would end. Trace file sizes were observed during several executions, and would usually sit at zero bytes, until a process would halt, at which time the trace size for the process would jump to its full, expected value. Programs which have traces larger than the buffer would probably see more of an advantage with the adaptive tracing algorithms. Second, the code for the adaptive algorithms was not optimized. The only optimization was moving the 'write to disk' out of the critical section. The critical sections were fairly large. There was a significant amount of time waiting for the access histories to be released. Third, the extra memory used by the adaptive algorithms, and the use of hash tables for the access histories, caused many page faults. Finally, it is possible that the processors were slow compared to the disk controller, so writing to the disk didn't really incur much of an overhead penalty.

It is expected that, as processors become faster, the extra work involved in adaptive tracing will be offset by the relatively slow operation of writing to disk. A highly optimized adaptive algorithm, on a computer with very fast processors would probably outperform a system that traces all references.

The compressed access history algorithms generally ran about 40% faster than the optimal algorithms. As noted above, writing to the disk didn't seem to incur much of penalty in runtime, so most of the difference can be attributed to the complexity of the algorithms. In the optimal algorithms a 'read-race-check' only checks the dependence of one event pair, but the 'write-race-check' has to determine the dependence of each event in the readset and the writer, and remove any readers ordered before the current write. This is much more complex than the compressed access history algorithms, where all events are

tested against the single entry in the access history. The additional work is the root of the extra overhead experienced with the optimal algorithms. Another factor is the extra memory required by the optimal algorithms, which results in more page swapping, and, consequently, higher overheads.

The Lamport clock algorithms run a few percent faster than the corresponding vector clock algorithms. This can be attributed to the simplicity of updating a Lamport clock, compared to updating a vector clock. Updating a Lamport clock is performed by changing a single variable, but updating a vector clock is performed by updating one variable *per processor*. Even when only two processors are used, it is twice as expensive to update a vector clock as it is to update a Lamport clock.

In addition to measuring the real time overhead for the algorithms, the processor and system time overheads were also recorded. In most cases the processor time overhead was slightly higher than the real time overhead, although adding processors generally made the processor time overhead grow quicker than the real time overhead. The system time overhead was fairly close for all the algorithms, and was only slightly lower than overhead of tracing all references. For most of the test programs, the system time encountered was between 10 and 15 times slower than the un-instrumented versions.

6.1.3 Overall

It is easiest to choose the best algorithm by first discounting those with drawbacks. The compressed access history, vector clock and compressed access history, non-critical edge algorithms trace too much information to make it worth the increased runtime overhead. The optimal, vector clock algorithm is very good, but it is out performed by the optimal, non-critical edge algorithm. Both of the Lamport clock algorithms produce small traces and have a slight runtime overhead advantage over the corresponding vector clock algorithms. Thus, there are three 'winners', the compressed access history, Lamport clock algorithm, the

optimal, non-critical edge algorithm and the optimal, Lamport clock algorithm. The Lamport clock algorithms do very well in terms of trace size and runtime overhead, but they do have drawbacks. For instance, they can't reproduce all consistent states of a program. Taking all this into consideration, the algorithm with the most 'positives' and the fewest 'negatives' is the optimal, non-critical edge algorithm.

6.2 Replay Mechanisms

Although implementing a replay mechanism was not part of this project, one can not ignore that aspect when comparing the merits of the various tracing schemes.

One method that has been proposed for replaying Lamport clock traced programs is based on *slice counters* [3]. An array of counters, one per process, is used to ensure that all events with Lamport clock values lower than the current event have occurred. The slice counter contains the Lamport clock value of the next event in each process. It is incremented by one if the event hasn't been traced, otherwise it is set to the Lamport clock value of the traced event. A process may only execute an event if the slice counters in all the other processes are greater than or equal to the Lamport clock value of the current event. This scheme has some big drawbacks. The most glaring is the runtime overhead incurred by checking the slice counters for every event in the execution. Because an array of slice counters is used, the inherent scalability of Lamport clocks is compromised.

A system that would have much less overhead would verify the Lamport clock value for an event only if the event is logged in the trace file. During the tracing phase, there are no provisions for enforcing consistent Lamport clocks at each event, therefore it seems a waste of time to enforce such conditions upon replay. The accesses that are traced capture the data dependencies. Therefore, enforcing the same data dependencies on replay will insure a proper replay.

This still has the drawback of depending on slice counters, and checking against each process at each traced event. A further refinement of this method would record which process a traced event was data dependent with. Then instead of making sure the Lamport clock of the current process was consistent with the clocks in all the other processes, one could merely enforce the condition that it is consistent with the Lamport clock value of the event in the process it is data dependent on. This restores the scalability, but the traces need to be larger to record the process numbers.

The replay mechanism for the vector clock traces is similar to the mechanism for replaying Lamport clock traces. In fact, it is simpler, because the local clock is just incremented for each event. All the dependencies that were originally traced, must be enforced during replay. At each event the player would check to see if that event is the next event in the trace file. If it isn't the player can process the event and go on to the next one. If it is in the trace file, a dependence must be resolved. The serial number of the racing event is held in the trace file. The player must wait until the serial number of the process containing the racing event is greater than or equal to the serial number in the trace file. This will satisfy the data dependence.

The same replay scheme can be used when the non-critical path edges are traced. With the benefit that there is less of a chance that processes will have to wait. By flattening the dependence edges in the execution graph, the clock values traced are closer to the clock value of the event dependent on it. So, the waits should be shorter. This means the replay will run faster than if edges, which are exactly on the critical, path are traced.

It should be noted that the same replay mechanism would work for traces made with the optimal algorithm and the compressed history algorithm. This is because the compressed history algorithm has a weaker data dependence relation. Any dependencies that are found with the optimal algorithm will also be found with the compressed history algorithm. The replay phase would

just be enforcing dependencies that aren't necessary for correct replay. When implementing a replay mechanism, it would be wise, and simple, to have it be able to replay from traces generated by either algorithm. This would allow the programmer to choose which tracing scheme to use.

6.3 The Probe Effect

No discussion on parallel debugging would be complete without a few words about the probe effect. Any instrumentation that is added to the code can potentially change the way a program interacts. In fact, this was encountered during the implementation of the algorithms on the Sequent. The library was dead locking, but when print statements were added to try to find out where, the dead lock disappeared. The bug eventually was found through the use of PDBX, the standard style debugger on the sequent, and careful scrutiny of the code. This experience further reinforced the need for a complete set of parallel debugging tools.

Although there are models that attempt to account for the perturbation instrumentation introduces [4], no attempt was made to minimize its effects in this project. Since research in perturbation analysis and trace and replay are still in the early stages, it was felt that attempting to implement a combination would not produce fruitful results in either area.

6.4 Future Work

There are many areas where this research can be continued. Most importantly, the efficiency of the implementations can be improved. As stated before, the overhead figures can be seen as worst case figures. Further reduction in overhead could be produced with a more efficient hash table, or smaller critical sections. Also, in the case of the compressed algorithms, the access history might not have to be locked during the race check. This was not explored, but would surely make the algorithms run much faster.

There are special cases where the algorithm can be simplified. Such as when an execution is running on 2 processors. The algorithms could just keep track of the most recent event on the other processor. The read set would only need to know where the most recent reader was.

The amount of memory required seems prohibitive. As the set of access histories grew to very large sizes, there were many page faults, and the throughput of the algorithms declined. There is a lot of area where the size of the access history for each variable could be reduces through encoding of information.

Another aspect that wasn't explored is the effect of grouping sets of variables into aggregates. This is another are which could reduce run time overheads. The reduction would come from the need for less memory to hold the access histories, leading to less page faults. Although there might be more problems waiting for locked access histories, when the access histories are used for a group of variables.

Of course, the area most in need of future work is the replay side of the system. This project did not implement any mechanism for replaying the executions with the trace files produced. The problem was left for further research because it was felt that the tracing scheme should be decided before the replay side was explored. In other words, you can't play something if you haven't decided exactly how its recorded.

7 Conclusion

This project explored the performance of six adaptive tracing algorithms. Each method had its merits, and they all have some drawbacks. When all things are taken into consideration, the algorithm that looks the best is the vector clock, non-critical edge tracing algorithm. The trace sizes are very small. Though not as small as those for the optimal Lamport clock algorithm. The overhead is within a few percent of the less complex Lamport clock algorithm. The largest advantage would be realized during replay. There would much less waiting for

other processes to reach events which had data dependencies. That is where the Lamport clock algorithms give up ground to the others. Because races are traced only when they are dependent on events in other processes with higher clock values, traced events would generally have to wait for the other events to occur. The compressed history algorithms can't compete as far as trace size is concerned, but they have the advantage of requiring much less memory, and only 50% of the runtime overhead. Replay would also be slower because all the dependencies between readers would have to be resolved. In general, these algorithms would work the best only when memory or overhead were of prime importance.

8 Acknowledgments

I would like to thank Robert Netzer for his knowledge and guidance, without which this project would not have been possible. I would also like to thank my parents for their endless support.

References

[1] Leslie Lamport "Time, Clocks, and the Ordering of Events in a Distributed System", *Communications of the ACM*, Vol. 21, Num. 27, (July 1978).

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

[3] Luk J. Levrouw and Koenraad M. R. Audenaert and Jan M. Van Campenhout, "A New Trace and Replay System for Shared Memory Programs Based on Lamport Clocks", ELIS, Universiteit Gent, B-9000 Gent, Belgium, (1994).

[4] Allen D. Malony and Daniel A. Reed "Models of Performance Perturbation Analysis", *ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 15-25, Santa Cruise, CA, (June 1991).

[5] Robert H. B. Netzer "Trace Size vs Parallelism in Trace-and-Replay Debugging of Shared-Memory Programs", Department of Computer Science, Brown University, Providence, RI, (September 1993).

[6] Robert H. B. Netzer "Optimal Tracing and Replay for Debugging Shared-Memory Parallel Programs", *ACM/ONR Workshop on Parallel Debugging*, pp. 1-11 San Diego. CA, (May 1993).

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

[8] Robert H. B. Netzer and Barton P. Miller "What are Race Conditions? Some Issues and Formalizations", *ACM Letters on Programming Languages and Systems* **1**, 1, (March 1992).

[9] Larry D. Wittie "Debugging Distributed C Programs by Real Time Replay", *SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pp. 57-67, Madison, WI, (May 1988).

A Appendix: Where the Code Lives

A.1 Simulated Versions of the Algorithms

The code for the simulations on the SparcStation is in the directory /u/rn/public/ess.

The algorithms are distributed in three files. race_check_eric.c contains the code for all three compressed access history algorithms. The file r_c_lamport.c contains the code for the optimal, Lamport clock algorithm. The file r_c_vector.c

contains the code for the optimal, vector clock algorithm and the optimal non-critical edge algorithm. The trace files are in the sub-directory 'traces'. There is a makefile that compiles everything.

The simulator is run from the command line and accepts the following command line arguments.

- -b'bit shift value': This is used to simulate aggregate access histories. The default is 0.
- -d'debug option': This is the level of debugging information printed. 0 (the default) turns off debugging. 1 prints out the addresses and some other information about the races traced. 2 prints out a full debugging report.
- -q: This puts the program in quite mode. (default)
- -v: This puts the program in verbose mode.
- -a: Performs race checks using all six algorithms.
- -c: Performs race checks using the collapsed access history, vector clock algorithm.
- -l: Performs race checks using the optimal, Lamport clock algorithm.
- -m: Performs race checks using the collapsed access history, Lamport clock algorithm.
- -n: Performs race checks using the optimal, non-critical edge algorithm.
- -o: Performs race checks using the optimal, vector clock algorithm.
- -r: Performs race checks using the collapsed access history, non-critical edge algorithm.
- -z: Measures some information about the types of accesses present.

A.2 Instrumented Libraries

The code for the instrumented parallel libraries is on topaz.cs.wisc.edu. It is all contained in the directory
/usr2/bart/netzer/races.replay2.
The include files are in the sub-directory 'h'. The test programs are in the sub-directory 'test'. The instrumented code for the libraries are in the sub-directory 'instr', which contains the sub-directories 'libc', 'libpps', and 'libtrace', which contain an instrumented version of the 'c' library, an instrumented version of the 'pps' library, and the trace libraries, respectively. The different versions of the source files for the trace libraries are in
/usr2/bart/netzer/races.replay2/instr/libtrace.
Any modifications to the code to enhance performance would probably take place in the 'instr/libtrace' directory. There is a makefile in 'libtrace' which compiles the trace libraries, puts them into their own libraries and puts the libraries in the ~/lib directory. The binaries for the libraries are in 'lib'. 'bin' contains a version of the compiler, *ccreplay*, which instruments the code.

To compile a program using the tracing libraries you must do the following:

- Use ccreplay instead of cc.
- Set the include path to ~/races.replay2/h.
- Include the following libraries:
  - ~/lib/libc.a
  - ~/lib/libpps.a
  - ~/lib/libtrace_all.a or
    ~/lib/libtrace_race_ov.a or
    ~/lib/libtrace_race_ol.a or
    ~/lib/libtrace_race_cv.a or
    ~/lib/libtrace_race_cl.a

There is a makefile with each of the test programs which produces an uninstrumented version, a version that traces all references, and versions for each of the four adaptive tracing libraries. Refer to these makefiles for more information about compiling the programs.

Compressed Access History Algorithms				
Program	*# References*	*Cmpr VC*	*Cmpr Non-Crit*	*Cmpr LC*
Barnes 2	66887	4004 (5.99%)	725 (1.084%)	232 (0.347%)
Barnes 4	66004	7999 (12.12%)	1982 (3.003%)	630 (0.954%)
Barnes 16	79426	21709 (27.3%)	8385 (10.6%)	3938 (4.96%)
Barnes2 2	175671	2840 (1.617%)	1195 (0.68%)	257 (0.146%)
Barnes2 4	173340	11102 (6.405%)	4393 (2.534%)	1790 (1.033%)
Barnes2 8	173466	19972 (11.513%)	7625 (4.396%)	4038 (2.328%)
gauss 2	162023	27274 (16.83%)	23661 (14.603%)	664 (0.41%)
gauss 4	162719	51574 (31.7%)	45899 (28.2%)	19291 (11.85%)
gcd 2	478675	9058 (1.89%)	8421 (1.76%)	6394 (1.34%)
join 2	35672	63 (0.177%)	37 (0.104%)	19 (0.053%)
join 4	35831	462 (1.289%)	307 (0.857%)	237 (0.661%)
locus 2	1932827	37552 (1.943%)	22769 (1.178%)	6182 (0.32%)
locus 16	2530803	614120 (24.3%)	447600 (17.7%)	341274 (13.5%)
mp3d 2	127041	1336 (1.052%)	791 (0.623%)	164 (0.129%)
mp3d 4	127675	3983 (3.12%)	2133 (1.67%)	646 (0.506%)
mtxmult 2	400209	56744 (14.2%)	107 (0.027%)	57 (0.014%)
mtxmul 4	400209	114404 (28.6%)	199 (0.05%)	155 (0.038%)
pnet 2	104298	9510 (9.12%)	5335 (5.12%)	1557 (1.5%)
pnet 4	139022	31509 (22.7%)	21502 (15.5%)	9094 (6.54%)
ptycho 2	112372	3718 (3.31%)	2202 (1.96%)	392 (0.348%)
ptycho 4	336690	3649 (1.08%)	2474 (0.735%)	841 (0.25%)
qs 2	11040	22 (0.199%)	22 (0.199%)	13 (0.118%)
qs 4	22872	478 (2.09%)	452 (1.98%)	429 (1.88%)
shpath 2	24775	7206 (29.1%)	7033 (28.4%)	987 (3.98%)
shpath 2	3760213	1227179 (32.6%)	1224404 (32.6%)	38963 (1.04%)
shpath 4	100344	33443 (33.3%)	32470 (32.4%)	21447 (21.4%)
sort 2	14846	1 (0.007%)	1 (0.007%)	1 (0.007%)
sort 4	29278	12 (0.04%)	6 (0.02%)	8 (0.027%)

Table 1: Trace sizes from the compressed access history algorithms

Optimal Algorithms				
Program	*# References*	*Optimal VC*	*Optimal Non-Crit*	*Optimal LC*
Barnes 2	66887	201 (0.3%)	170 (0.254%)	102 (0.152%)
Barnes 4	66004	407 (0.617%)	358 (0.542%)	210 (0.318%)
Barnes 16	79426	1475 (1.86%)	1402 (1.76%)	871 (1.097%)
Barnes2 2	175671	461 (0.262%)	171 (0.097%)	145 (0.083%)
Barnes2 4	173340	2260 (1.3%)	773 (0.45%)	856 (0.494%)
Barnes2 8	173466	3489 (2.01%)	1107 (0.638%)	1050 (0.6%)
gauss 2	162023	20806 (12.8%)	197 (0.122%)	78 (0.048%)
gauss 4	162719	22362 (13.7%)	548 (0.336%)	266 (0.163%)
gcd 2	478675	3764 (0.786%)	1728 (0.361%)	36 (0.007%)
join 2	35672	61 (0.171%)	36 (0.1%)	19 (0.053%)
join 4	35831	287 (0.8%)	140 (0.391%)	77 (0.215%)
locus 2	1932827	11257 (0.582%)	1282 (0.066%)	1436 (0.074%)
locus 16	2530803	88750 (3.51%)	12787 (0.505%)	11431 (0.452%)
mp3d 2	127041	625 (0.492%)	87 (0.068%)	97 (0.076%)
mp3d 4	127675	1293 (1.013%)	226 (0.177%)	208 (0.163%)
mtxmult 2	400209	2453 (0.613%)	1 (0.0002%)	2431 (0.607%)
mtxmult 4	400209	7207 (1.8%)	3 (0.0007%)	7078 (1.769%)
pnet 2	104298	2570 (2.46%)	2519 (2.42%)	1260 (1.21%)
pnet 4	139022	7272 (5.23%)	7194 (5.17%)	5944 (4.28%)
ptycho 2	112372	102 (0.091%)	5 (0.004%)	17 (0.015%)
ptycho 4	336690	762 (0.226%)	18 (0.005%)	16 (0.005%)
qs 2	11040	22 (0.199%)	22 (0.199%)	13 (0.118%)
qs 4	22872	452 (1.98%)	452 (1.98%)	429 (1.88%)
shpath 2	24775	653 (2.64%)	575 (2.32%)	312 (1.26%)
shpath 2	3760213	8973 (0.238%)	8072 (0.215%)	4243 (0.113%)
shpath 4	100344	3547 (3.53%)	3249 (3.24%)	2027 (2.02%)
sort 2	14846	3 (0.02%)	1 (0.007%)	2 (0.013%)
sort 4	29278	17 (0.058%)	5 (0.017%)	13 (0.044%)

Table 2: Trace sizes from the optimal tracing algorithms

Runtime Overhead						
Program	*Time* Sec.	*Trace All*	*Opt VC*	*Opt LC*	*Cmp VC*	*Cmp LC*
gauss 2	1.366	4.29	15.53	14.32	11.34	10.85
gauss 4	1.122	3.73	15.77	15.2	12.68	12.63
gauss 8	1.068	3.70	18.38	17.99	14.57	14.30
gcd 2	3.21	9.43	21.64	20.14	15.89	15.82
gcd 4	3.63	10.93	34.19	30.45	20.41	20.08
gcd 8	2.886	13.77	60.91	53.76	28.73	28.18
join 2	4.28	1.55	3.46	3.34	2.35	2.44
join 4	5.176	1.79	4.97	4.85	6.35	2.77
join 8	5.408	1.65	4.68	4.63	2.70	2.66
flow 2	5.664	17.3	56.22	49.78	47.85	47.82
flow 4	1.74	14.11	59.36	52.8	48.2	44.83
flow 8	2.23	19.8	118.52	120.21	91.18	82.09
mesh 2	2.65	16.15	75.37	70.12	38.72	37.94
mesh 4	1.874	15.69	93.98	87.66	46.29	45.01
mesh 8	1.552	14.76	112.69	105.78	56.87	55.23
mtxmult 2	1.644	7.70	30.98	28.24	22.17	21.44
mtxmult 4	1.054	6.88	37.66	35.95	30.32	29.74
mtxmult 8	.818	6.10	49.02	48.86	42.57	41.54
qs 2	1.436	13.27	51.81	47.15	32.48	32.34
qs 4	1.112	11.35	56.92	52.37	38.35	36.83
qs 8	1.014	10.36	63.81	59.78	44.73	42.78
shpath 2	5.656	3.85	23.43	22.42	10.17	9.82
shpath 4	4.084	3.42	33.68	32.56	13.93	13.44
shpath 8	3.618	3.17	47.81	44.23	18.48	17.98
sort 2	6.046	21.31	94.64	83.98	61.02	59.30
sort 4	4.746	20.43	112.02	96.60	76.58	75.36
sort 8	4.222	20.72	125.57	120.58	92.89	86.37

Table 3: Runtime overheads of the various algorithms