University of California
Santa Cruz

Improving Trace-Based Parallel Debugging

A thesis submitted in partial satisfaction
of the requirements for the degree of
Master of Science
in
Computer Engineering
by
Theodore R. Haining
June 1993

The thesis of Theodore R. Haining is approved:

Charles E. McDowell

David P. Helmbold

Darrell D.E. Long

Dean of Graduate Studies and Research

Contents

List of Figures

List of Tables

Improving Trace-Based Parallel Debugging

Theodore R. Haining

Abstract

Parallel programs sometimes execute unreliably because the improper use of synchronizing constructs introduces data races. These data races can cause non-deterministic effects in the internal data state of the program, resulting in deadlock or incorrect results. To overcome this difficulty, the location of these data races must be found or otherwise eliminated (e.g. new languages). A variety of techniques currently exist to detect such races. These methods generally fall into one of three major groups: those that attempt to locate races at compile time using static analysis, those that locate races at run-time using some kind of monitor, or those that locate races after execution completes by using traces.

This thesis will present and analyze certain aspects of a new algorithm designed for use in trace analysis, based on the work of Helmbold, McDowell, and Wang. Test results show that the new algorithm is always able to find more data races than the algorithm it is meant to replace. A characterization of the approach motivating this algorithm indicates where it succeeds in finding data races, and where (and why) it fails. A proof establishes that the new algorithm finds a non-empty subset of the races which exist for all executions of a parallel program on the same set of inputs. Test results indicate the effectiveness of the algorithm with random test traces of different length. Finally, an outline of the procedures used to test the new algorithm establish strength of the conclusions presented here based on the depth and breadth of the set of inputs used as test data.

Acknowledgments

I would like to thank Charlie McDowell and David Helmbold without whose constant help and encouragement this thesis would not have been possible. I would also like to thank Darrell Long for reading this thesis. My special thanks to my mother, father, and sister whose support and understanding helped to keep me going.

This research was supported by National Science Foundation contract number CCR-9102635.

Chapter 1
Introduction

Programmers have difficulty visualizing the behavior of parallel programs. As a result, concurrent tasks are often insufficiently or incorrectly synchronized, causing races. Therefore, to eliminate non-deterministic behavior, it must be possible to detect races. To find races, better software tools are required; tools which are capable of providing information about the interaction of concurrent tasks.

At this time, three major approaches have been applied to the problem of race detection: static analysis [McD89], run time (on-the-fly) analysis [MC92], and post-mortem trace analysis [EGP89, HMW93]. None of them is able to find all the races that potentially exist in arbitrary programs. Static analysis attempts to find races at compile time by computing all possible concurrency states for the program under examination. This has the advantage of examining all possible control states of concurrent tasks at every point of execution, but, examination of every concurrency state has been shown to be an NP-hard problem [Tay83]. On-the-fly analysis attempts to locate possible races by using state information taken from the program as the program runs. It has the potential benefit of using less space than trace analysis because some data is used and discarded. Trace based analysis typically uses a listing of variable accesses, event ids and other state information recorded during a program execution. This information is analyzed to determine possible data races. It uses large amounts of memory to hold trace data and has been found to be co-NP hard for arbitrary programs [NM90]. It does have the advantage that it may be used to find races in executions of the program other than the one that actually occurred.

This thesis focuses on the problem of approximating event orders in trace based analysis. An analysis tool could correctly compute the partial order of events based on a program trace, use this information to find races, and then report this result. Because this problem is not decidable in polynomial time for arbitrary programs, algorithms must be used which approximate this process. This work proposes an improved method for finding an approximation which is practical both in terms of computational cost of the analysis and amount of race information returned on completion.

Helmbold, McDowell, and Wang produced a three stage algorithm for analyzing traces of programs using either semaphore style synchronization, or traces of programs using IBM Parallel Fortran [HMW93]. The algorithm generates a partial order of events from an inferred program based on the trace under examination. (The inferred program is the same as the program generating the trace, except that all conditional branches are replaced with unconditional branches resolved in the same direction as they occurred in the execution generating the trace [HMW93].) The partial order represents the "always happens before" relationships between events in the inferred program. (If an event e_1 "always happens before" an event e_2, then e_1 must precede e_2 in all executions of the inferred program where e_2 occurs [HMW93].) Events not ordered in this partial order are used to report races.

The results of this event ordering analysis have strengths and weaknesses. Since results are based on the "always happens before" ordering, the produced partial order is said to be *safe*, meaning that the partial order can be used to find races in all executions consistent with the program inferred from the trace in numbers and types of events. The results of the three algorithms indicate a non-empty subset of the races that actually occurred in the program run from which the trace is taken. Because the ordering of events in the program is approximate, not all order relations may be reported, leading to spurious race reports.

Testing has resulted in the refinement of the third stage of the algorithm which expands the number of "always happens before" relations in a partial order of the events in the program. This thesis presents:

- the revised algorithm,

- a proof that the partial order it produces is safe,
- an implementation of the algorithm in a testing application, and
- numerical data showing the improvement detection capability of the algorithm over the one it is meant to replace.

The remainder of this chapter outlines the notation that will be used in subsequent chapters of the work and briefly presents the previous algorithm. Chapter 2 presents the new algorithm with the supporting proof of its ability to safely add event order information. Chapter 3 details the implementation of the algorithm. Chapter 4 provides numerical data showing the improvement of the new algorithm taken over long test runs. Finally, the last chapter discusses future directions that this research could take and provides a summation of the results presented in this thesis.

1.1 A Virtual Time Model

For the purpose of this work, a parallel program is a logical construct containing a finite number of tasks. Each task consists of a unique identifier and is a sequence of synchronization operations and data manipulations which execute in linear order. An event is considered to be a significant step in program execution. This thesis deals with computing partial orders of events. Therefore, the events of interest in a program trace are synchronization events. A trace of a parallel program to be used as input for analysis represents one possible total ordering of events in the program as it executes.

This work will deal with programs employing counting semaphore synchronization. When this scheme is used, two operations, Wait and Signal,[1] are defined for each semaphore. Therefore, each synchronization event will be a tuple containing: the operation completed (Wait or Signal), the semaphore which was affected, and the id of the task that performed the operation.

[1] Wait and Signal are used here in place of the **P** and **V** to better illustrate the meaning of each operation.

Events are ordered using integer-valued timestamps [Fid88]. Each task maintains its own count of the number of events it has executed, which are updated every time the task executes an event. Each task also maintains counters recording the number of events that all other tasks have executed. These counters get updated when the task synchronizes with other tasks. The current values of local and non-local counters that a task maintains form a timestamp for each event in the task. Sequences of events can then be totally or partially ordered by assignments of timestamps to events. Such event orders can be modified by changing the timestamps associated with events rather than adding, deleting, or re-ordering event lists.

An event e with timestamp τ precedes another event e' with timestamp τ' in a partial order if and only if every component of τ is less than or equal to the corresponding component of τ'. Events e and e' are unrelated in the partial order when both some component of τ is greater than the corresponding component of τ', and some (other) component of τ' is greater than the corresponding component in τ.

Definition 1.1 *For any two timestamps τ_1, τ_2 in Z^n*

1. $\tau_1 \leq \tau_2 \iff \forall i(\tau_1[i] \leq \tau_2[i])$
2. $\tau_1 < \tau_2 \iff \tau_1 \leq \tau_2 \wedge \tau_1 \neq \tau_2$
3. $\tau_1 \parallel \tau_2 \iff \neg(\tau_1 < \tau_2) \wedge \neg(\tau_2 < \tau_1)$.

where n is the number of tasks in the program.

Timestamp τ_1 is earlier than *timestamp τ_2 (or τ_2* is later than *τ_1) in the partial order when $\tau_1 < \tau 2$. Similarly, τ_1 and τ_2 are* unordered *in the partial order when $\tau_1 \parallel \tau_2$.*

It is now convenient to define a few common functions that will be used in this work:

Definition 1.2 *For any m timestamps $\tau_1, \ldots, \tau_m$ of Z^n*

- $\overline{\min}_k(\tau_1, \ldots, \tau_m), k > 0$ *is the vector of Z^n whose ith component is the k^{th} smallest element in the collection $\tau_1[i], \tau_2[i], \ldots, \tau_m[i]$,*

- $\overline{\max}(\tau_1, \ldots, \tau_m)$ *is the vector in* Z^n *whose* i*th component is* $\max(\tau_1[i], \ldots, \tau_m[i])$.

Conventionally, $\overline{\min}_0(\tau_1, \ldots, \tau_m)$ *is defined to be* $\bar{0}$*, the all-zero vector.*

As an example, $\overline{\min}_3([1,2],[1,3],[2,4],[2,5],[3,2])$ is $[2,3]$. $\overline{\min}_k(\tau_1, \ldots, \tau_m)$ the k^{th} is often called *component-wise minimum* of $\tau_1, \ldots, \tau_m$, and $\overline{\max}(\tau_1, \ldots, \tau_m)$ the *component-wise maximum* of $\tau_1, \ldots, \tau_m$.

Definition 1.3 *Given an event e performed by task* T_i *in a causal trace, let* $\tau^{\#}(e)$ *be the timestamp containing the local event count for e (one more than the number of events previously performed by* T_i *in the trace) in the ith component and zeros elsewhere.*

Definition 1.4 *Given an event e performed by task* T_i *in a causal trace, let* e^p *denote the previous event performed by* T_i *in that trace if such an event exists.*

1.2 Algorithms for Generating a Partial Order

Since the new part of the algorithm that this thesis will present improves on a section of the trace analysis of Helmbold, McDowell and Wang, it is useful to review the details of each of the three algorithms they developed as a part of their approach.

To obtain an initial partial order for an execution trace, on which further analysis can be based, they first use an algorithm based on the one provided in Fidge [Fid88] and Mattern [Mat88]. This process associates a Wait event with an unmatched Signal event on the same semaphore, creating a partial order which pairs every Wait event with a Signal event which allowed it to precede. This generates a partial order that contains orderings that are not "always happens before" orderings. Instead, this partial order represents the causal orderings that did occur in a particular execution.

To generalize this partial order information to make it valid for all possible executions containing the same events, they use a process called *rewinding* (Algorithm 1.2) to decouple Wait events from specific Signal events. After rewinding, every Wait event has a time vector

Algorithm 1.1 (Initialize)
Given a causal trace, each event e is assigned a time vector, $\tau(e)$, as follows:

for each event e in the trace
    if e is a Wait event on semaphore S,
        let e' be Signal event unblocking e;
        set $v_s = \tau(e')$;
    else
        set $v_s = \bar{0}$, the all zero vector;
    end if;
    set $\tau(e) = \overline{\max}(\tau(e^p), \tau^{\#}(e), v_s)$;
end for;

Algorithm 1.2 (Rewind)
Repeat the following procedure until no further changes are possible.

for each event e in the trace
    if e is a Wait event on semaphore S,
        let $e_1^s \ldots e_k^s$ be all the Signal events on S;
        set $v_s = \overline{\min}(\tau(e_1^s), \ldots, \tau(e_k^s))$;
    else
        set $v_s = \bar{0}$, the all zero vector;
    end if;
    set $\tau(e) = \overline{\max}(\tau(e^p), \tau^{\#}(e), v_s)$;
end for;

that reflects the assumption that any Signal (on the same semaphore) could have been the Signal that triggered the Wait.

Unfortunately, the newly established order relation is safe but too conservative because some "always happens before" ordering arcs are deleted as a part of the rewind procedure. Some of these arcs can be restored through the application of the following observations:

Observation 1.2.1 *If some wait event e_w on semaphore A is known to follow n other waits on semaphore A (given the safe partial order already computed) then e_w must follow $n+1$ signal events on semaphore A. Thus additional edges can be inserted into the partial order*

by increasing the (vector) timestamp for e_w so that each component is at least as big as the corresponding components in $n+1$ of the timestamps for the signal events on semaphore A.

Observation 1.2.2 *If one of the $n+1$ signals, call it e_s, needed in observation 1.2.1 is known to be preceded by an additional wait event on semaphore A that is not one of the n wait events known to precede event e_w, then $n+2$ signals occur before e_w whenever e_s occurs before e_w.*

This second observation implies that $n+1$ signals other than e_s occur before e_w. In the program of Figure 1.1, the S_1 event in task D corresponds to the e_s event in Observation 1.2.2. They call this phenomenon *shadowing*, as the "shadow" cast by the preceding wait prevents e_s from satisfying the signals needed by e_w.

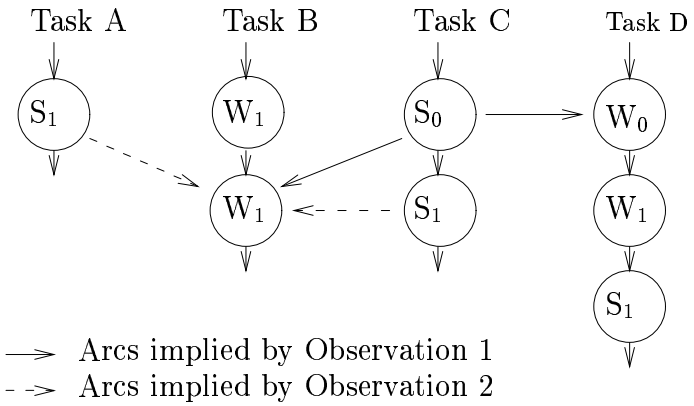

Figure 1.1: An Example of Observations 2.1 and 2.2. Note that the program can deadlock with Task D waiting on semaphore 1.

The Expand Algorithm (Algorithm 1.3) applies these two observations as it cycles through the entire timestamp representation of a trace. It does this by considering the set of Signal events, R, and the set of Wait of events, W, that exist in relation to the Wait event e under consideration and its predecessor e^p. Observation 1.2.2 is applied during the construction of $R(e)$. It then employs Observation 1.2.1 taking the k^{th} minimum of $R(e)$ to create the timestamp v_s. Finally, the $\overline{\max}$ operation is used to combine v_s, $\tau^{\#}(e)$, and $\tau(e^p)$ to create the new timestamp for e.

Algorithm 1.3 (Expand)
Repeat the following procedure until no more changes are possible.

for each event e in the trace
    if e is a Wait event on semaphore S,
        let $W(e) = \{e_w : e_w$ is a Wait event on S, $\tau(e_w) < \tau(e)\}$;
        let $R(e) = \{\hat{e} : \hat{e}$ is a Signal event on S, $\tau(e) \not\leq \tau(\hat{e})$,
                and $\hat{e}$ is not shadowed with respect to $e\}$;
        let $k =\mid W(e) \mid$;
        let $v_s = \overline{\min}_k(\tau(\hat{e}) : \hat{e} \in R(e))$;
        set $\tau(e) = \overline{\max}(\tau(e^p), \tau^{\#}(e), v_s)$;
    else
        set $\tau(e) = \overline{\max}(\tau(e^p), \tau^{\#}(e))$;
    end if;
end for;

1.3 Related Work

Much research effort has been directed toward finding partial orders of events in situations using anonymous synchronization primitives in parallel and distributed systems. This work has been motivated by the existence of several parallel programming languages which use anonymous communication methods like locks, semaphores, and monitors.

Emrath, Ghosh, and Padua [EGP89] use a combination of static and trace analysis to debug parallel programs in Cedar Fortran using fork/join synchronization with Post, Wait, and Clear events. They use static analysis to generate a DAG called a *Task Graph* containing four kinds of arcs representing types of event ordering:

- Sequential Ordering – ordering due to one event following another in the same task
- Start Ordering – ordering due to the start of a created task following a fork event
- Wait Ordering – ordering due to a join event following every event in a task
- Synchronization Ordering – ordering due to the use of post, wait, and clear events.

Since static analysis may fail to successfully give a definite dependence relation between program statements which execute several times, it may provide incomplete synchronization ordering information.

Trace analysis is used to add synchronization ordering arcs to the Task Graph. This analysis employs an algorithm called $SYNC$ which is similar to Algorithm 1.3 as presented in this thesis. For each Wait event w, it finds a set of Post events that might have triggered w. This is the set of all Posts on the same event minus all Posts that follow w or are cleared before w is executed. It computes the set of closest common ancestors of this set of Posts. Finally, an arc is added to the Task Graph between every member of this set of closest ancestors and the Wait event w.

SYNC and Algorithm 1.3 differ in two major ways. First, Algorithm 1.3 modifies event orders using timestamps, saving the computational overhead of recalculating ancestor sets for every pass of the algorithm. Secondly, Algorithm 1.3 may find more ordering relationships.

Netzer and Miller [NM91] seek to use static analysis with trace analysis to reduce the number of spurious race reports produced when analyzing traces. They do this by augmenting the graph of temporal order relations with shared-data dependencies and event-control information to determine if reported races are feasible (could actually occur) or infeasible (could not occur). Where this information is inexact, they specify a set of races which are tangled, meaning that at least one race in the set is guaranteed to be feasible, but the rest of the races in the tangle may be infeasible.

In this form of trace analysis, program execution PP is a triple consisting of:

1. a set of events E, with each event having a start time (e_s) and a finish time (e_f),
2. a set of temporal ordering relations $\xrightarrow{T}$, representing the order of events in the trace, and
3. a set of shared-data dependencies $\xrightarrow{D}$, indicating which events can affect each other because of shared variable accesses.

Because many tracing facilities only record temporal ordering information for synchronization events in trace files, $\xrightarrow{T}$ may have to be approximated with an incomplete temporal ordering $\xrightarrow{T'}$. The data dependency $\xrightarrow{D}$ is usually replaced with with an approximate set of data dependencies $\xrightarrow{D'}$ because this information is generally not recorded and must be approximated from Read and Write sets generated by static analysis. Because the methods used to find both $\xrightarrow{T'}$ and $\xrightarrow{D'}$ produce under approximations of $\xrightarrow{T}$ and $\xrightarrow{D}$, some useful information is lost.

To determine if a race on events a and b is feasible, a prefix of program execution must be found such that a is unordered with respect to b. This is done by constructing a temporal ordering graph G_D from information in PP. It is then possible to validate some races with Theorem 1.5 and Theorem 1.6 below.

Additional races may be validated by observing that not all data dependencies necessarily cause events to be ordered. By estimating a set of dependencies describing event interactions within the program called event-control dependencies $\xrightarrow{E'}$ from $\xrightarrow{T'}$ and $\xrightarrow{D'}$, it is possible to construct a graph G_E from G_D which contains only data dependencies which are not event-control dependencies. G_E can be used to validate more races with Theorem 1.7.

Finally, it is possible to indicate that some races may be caused by other races. To do this, it is necessary to define a race ordering relation $\xrightarrow{R}$, where for a race between events a and b (indicated by $< a, b >$):

$$< a, b > \xrightarrow{R} < c, d > \iff (a \xrightarrow{E'} b \wedge b \xrightarrow{E'} c) \vee (a \xrightarrow{E'} d \wedge b \xrightarrow{E'} d) \tag{1.1}$$

Races can then be identified as artifacts using Theorem 1.8.

Theorem 1.5 *An apparent data race $< a, b >$ is feasible if a and b are unordered by G_D.*

[NM91]

Theorem 1.6 *In each tangle defined by G_D (each strongly connected component of G_D), at least one of the tangled data races is feasible.*

[NM91]

Theorem 1.7 *An apparent data race* $< a, b >$ *is feasible if* a *and* b *are unordered by* G_E*, and no successor of* a *or* b *in* G_E *can event-control* a *or* b*.*

[NM91]

Theorem 1.8 *If* $< a, b > \overset{R}{\not\rightarrow} < c, d >$ *then* $< c, d >$ *could not have been an artifact of* $< a, b >$*.*

[NM91]

The work of Netzer and Miller is complementary to the work in this thesis. The purpose of the work presented here is to describe an algorithm capable of providing a more accurate temporal ordering graph for race detection. Netzer and Miller's work is directed toward reporting which subset of the races reported from such a graph could actually occur.

John Mellor-Crummey [MC92] has developed a system for combining compile time static analysis with on-the-fly techniques to reduce the amount of state information necessary at execution time to be used on Fortran programs. This is accomplished through a three stage compilation process. Local analysis calculates and stores data on the interprocedural effects for each procedure. Next, interprocedural propagation collects local summary information from each procedure and uses a call graph to perform interprocedural analysis. Finally, code for concurrency bookkeeping, access checks, and access history compilation is inserted based on computed interprocedural solutions.

Results appear promising. In test runs on scientific code, use of interprocedural and intraprocedural techniques resulted in an execution speedup of two to three times over on-the-fly analysis code employing no static analysis of this type. It should be noted, however, that the improved code still ran four to eight times slower than the code without the on-the-fly race-detection.

This research is mentioned because it presents a possible solution to the some of the problems of trace analysis. In this case, interprocedural analysis is used to improve run-time performance of code using on-the-fly analysis by reducing the amount of required state

information. Since trace-based analysis also needs state information, these methods may help to reduce the size of the trace file to be stored, and to provide automatic detection of the shared variables where races may be located.

Chapter 2
A New Algorithm For Event Ordering

The Initialize, Rewind, and Expand algorithms were proven to produce partial orders which were true for all executions of the inferred programs taken from traces. Because of a limited number of actual program traces to use for testing purposes, it was not known how frequently the Expand algorithm produced an approximate partial order that did not contain all the "always happens before" relationships that could be present for arbitrary programs.

In order to make this determination and refine the algorithm to reduce this frequency, it was necessary to determine when it failed (It did not find a program and an execution trace containing two events e_1 and e_2 such that e_1 happens before e_2 in every execution of the program in which they both occur and our algorithm indicates that e_1 and e_2 are unordered).

A random program generator was constructed and the algorithm was applied to the random programs it created (See Chapter 3 for implementation details). The algorithm's results were compared with the partial orders produced by an exponential time "brute force" algorithm which appears practical only for the relatively small programs. This process quickly identified traces where the algorithm failed to find some of the "always happens before" orderings. These experimental results have lead to considerable modifications of the original Expand part of the algorithm (the others being unchanged). This chapter will present some new observations on which the changes that were based, the new Expand part

of the algorithm, and a proof that the new Expand part safely adds to the partial order of events.

2.1 New Observations

The study of random programs has motivated changes in the algorithm so that they exploit the following additional observations:

Observation 2.1.1 *If wait event e_w on semaphore A is known to follow n waits on some* other *semaphore B (given the safe partial order already computed) then e_w must follow n signal events on semaphore B.*

This extends observation 1.2.1 to apply to semaphores other than the one used by the event e_w. The example that lead to this observation is shown in Figure 2.1.

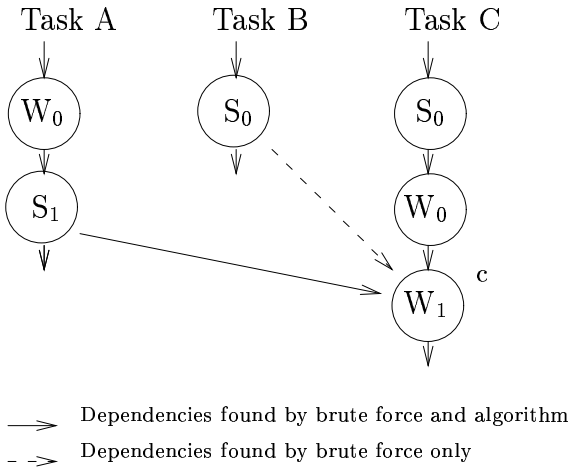

Figure 2.1: Example motivating observation 3.1. Event c is preceded by two wait on semaphore 0 events (W_0's) and therefore must be preceded by two S_0's.

Observations 1.2.1 and 2.1.1 lead to what could be called first order inferences. A particular event must be preceded by a certain number of wait events on various semaphores and therefore the event in question must be preceded by as many signals as there are waits preceding it. Shadowing (see the discussion of Observation 1.2.2) is a second order inference, i.e. if signal e_s happens before wait e_w then additional signals are also needed for the waits

that precede e_s (and have not already been accounted for because they also precede e_w). Our next observation is a second-order extension of Observation 2.1.1, somewhat like shadowing is a second-order extension of Observation 1.2.1.

Observation 2.1.2 *If some signal event e_s is going to help satisfy the signals needed for wait event e_w, then e_w will also have to follow any signals needed by wait events that precede e_s.*

This can be seen best by the example shown in Figure 2.2.

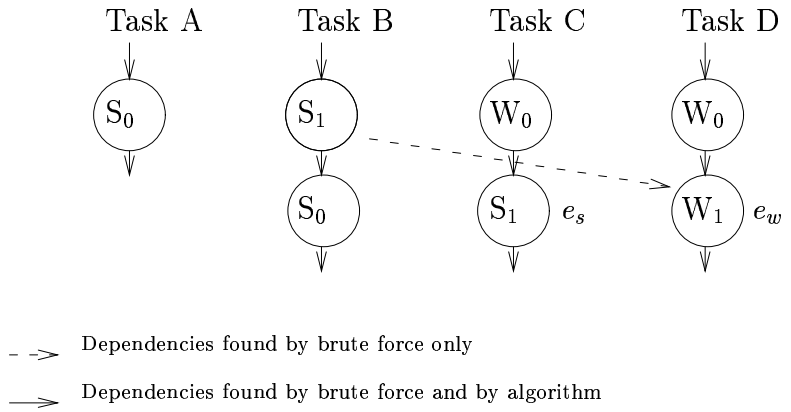

Figure 2.2: Example motivating observation 3.2. Events labeled e_s and e_w correspond to those in the observation. For e_w to proceed, a signal on Semaphore 1 is required. If the Signal to be used is the Signal in Task B, e_w follows e_s. Alternately, the Signal needed could be in Task C. If the Signal in Task C and e_w both occur, two Signal events on Semaphore 0 are required and e_w must still follow e_s.

This chaining effect (the execution needs n signals but one of those signals needs m more signals) can be repeated arbitrarily[1] and Figure 2.3 gives an example that takes this chaining effect one step further.

[1]It is believed this chaining may occur infrequently if at all in real programs.

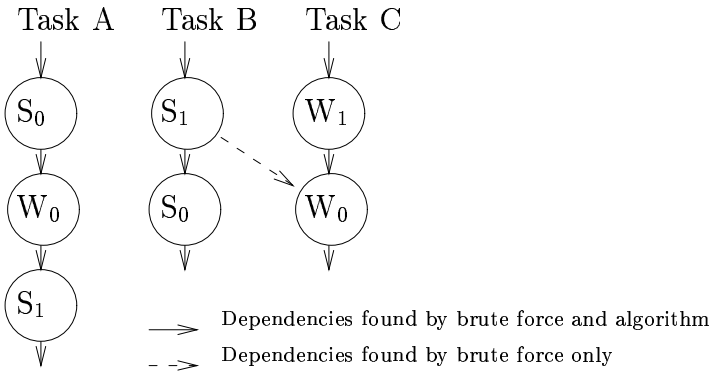

Figure 2.3: Example requiring two applications of observation 3.2. Here, one Signal event on Semaphore 1 is required if e_w in Task C is to proceed. The needed Signal is in either Task A or Task B. If e_s unblocks the Wait, then e_w follows e_s. If the Signal in Task A activates e_w, then two Signal events on Semaphore 0 are needed (one each for e_w and the Wait event in Task A). Therefore, the Signal event following e_s is required and e_w still follows e_w.

2.2 An Explanation of the Recursive Algorithm

The *Recursive Expand* algorithm (Algorithm 2.1) cycles through the entire sequence of events in the trace, using a routine called *modify* to advance the timestamps of Wait events based on Observation 2.1.2. Since an arbitrary number of applications of Observation 2.1.2 may be required to determine the orders of some events, the goal was to capture this chaining in a recursive algorithm that can be allowed to recurse as deep as time will allow. The algorithm becomes polynomial (and conservative) when the recursion is limited to a fixed bound.

The function *modify* considers the set of Signal events, R, and the set of Wait events, W, that exist in relation to two timestamps: τ_{in} and τ_s. The timestamp τ_{in} is an amalgamated timestamp representing all the events previously considered through recursive calls to *modify*. It will act as the event e_w in the current application of Observation 2.1.2. The timestamp τ_s will function as the timestamp of the Signal e_s from Observation 2.1.2. The quantity *depth* indicates the number of times that Observation 2.1.2 is to be recursively applied.

The function *modify* is used to build a set, T, of timestamps which are associated with events in R, but do not represent the timestamps that the events have in every execution of the inferred program of the trace. The *modify* function employs Observation 2.1.1, taking the k^{th} component-wise minimum of T (where k is the number of Wait events in W) to determine the earliest time that τ_s can be. It then returns the new timestamp in the quantity τ_m.

Finally, the main loop takes the timestamp returned by *modify* for a specific Wait event, and uses $\overline{\max}$ to combine it with $\tau(e^p)$ and $\tau^{\#}(e)$.

2.3 Safety of the Expanded Partial Order

In order for the Recursive Expand algorithm to replace Algorithm 1.3, it must only add ordering relationships to the partial order of events in the trace which are true for all

Algorithm 2.1 (Recursive Expand):
Repeat the following procedure until no more changes are possible.

for each event e in the trace
    if e is a Wait event on semaphore S,
        $\overline{\tau}(e) = modify(\tau(e), \tau(e), e, depth)$;
        set $\tau(e) = \overline{\max}(\tau(e), \tau(e^p), \overline{\tau}(e))$;
    else
        set $\tau(e) = \overline{\max}(\tau(e), \tau(e^p))$;
    end if;
end for;

Function $modify(\tau_{base}, \tau_{in}, \hat{e}, depth)$:
    let $\tau_m = \tau(\hat{e})$
    if $depth = 0$
        return(τ_m);
    end if;
    for each semaphore σ,
        let $T = \emptyset$;
        let $W(\sigma) = \{e_w : e_w$ is a Wait event on σ, and
                either $\tau(e_w) \leq \tau_{in}$ or $\tau(e_w) \leq \tau(\hat{e})\}$;
        let $R(\sigma) = \{e_s : e_s$ is a Signal event on σ,
                $\tau(e_s) \not\geq \tau_{in}$ and $\tau(e_s) < \tau_{base}\}$;
        for each e_s in R,
            $\tau = \overline{\max}(\tau_{in}, \tau(\hat{e}))$;
            $T = \{modify(\tau_{base}, \tau, e_s, (depth - 1))\} \bigcup T$;
        end for;
        let $k = \mid W(\sigma) \mid$;
        let $\tau_s = \overline{\min}_k(T)$;
        let $\tau_m = \overline{\max}(\tau_m, \tau_s)$;
    end for;
    return(τ_m);

executions of the inferred program for the trace. More formally, this means that all partial orders produced by Algorithm 2.1 must conform to the following definition:

Definition 2.1 *An assignment of timestamps to events, τ, represents a safe partial order if the following relation is true for all timestamps τ assigned to events in a set of events in the program, Set:*

if $\tau(e')[j] \leq \tau(e)[j]$ and e' is executed by task j then $e' \prec e$ when $e, e' \in Set$.

Since Algorithm 2.1 was developed from Algorithm 1.3 as a part of the experiments which produced Observations 2.1.1 and 2.1.2, the safety of the Recursive Expand Algorithm must be proven.

The proof will begin with notation that will be used throughout the proof, then will provide a series of Facts about the Recursive Expand Algorithm, and finally present one Theorem and a series of Lemmas that constitute the proof itself.

2.3.1 Notational conventions

The following notation will be used in statements about the Expand Algorithm.

- P is the parallel program under consideration, and uses counting semaphores as synchronizing events.
- τ is a previously assigned set of timestamps which represent a safe order of the events in P.
- e_w and e_s are a Wait event and a Signal event in P on some semaphore S (not necessarily the same one).
- Σ is the number of semaphores in P.
- σ is the number of an arbitrary semaphore, $0 \leq \sigma \leq \Sigma$.
- Set is a set of events in P which always contains e_w.
- $Sema(e)$ is the semaphore accessed by event e.
- e_{wait} and e_{sig} are an arbitrary Wait event and an arbitrary Signal event.

- e, e' are events of any type and any semaphore.
- $\tau_1 \leq \tau_2 \Longleftrightarrow \forall i(\tau_1[i] \leq \tau_2[i])$
- $\tau_1 > \tau_2 \Longleftrightarrow \forall i(\tau_1[i] \geq \tau_2[i])$ and $\tau_1 \neq \tau_2$.
- $\tau^{\#}(e)$ is the timestamp containing the local event count for the event e in its i^{th} component and zeros elsewhere where e is in task i.
- e^p represents the event which immediately precedes event e on the same task.
- $e \rightarrow e'$ indicates that e precedes e' in the total order of the events in the execution of P under discussion.
- $e \prec e'$ indicates that e precedes e' in a causal, minimal partial order of events of every execution of P where e and e' are present.
- an operation is declared to be 'safe' if it is used to update timestamps in a safe partial order and is guaranteed to produce another partial order which also is safe.

2.3.2 Useful Facts

To better understand any claims made about the Recursive Expand Algorithm, it is useful to concisely reiterate the following facts taken from the statement of Algorithm 2.1 [2]. The following hold during any call to *modify*,

$$\tau_m = modify\,(\tau(e_w), Set, \tau(e_s), depth),$$

where Set contains e_w.

Fact 2.3.1 *$W(\sigma)$ is a set of Wait events on σ generated during the call to modify.*

$$W(\sigma) \;=\; \left\{ \;\; e_{wait} |\, (\tau(e_{wait}) \leq \overline{\max}(Set) \;or\; \tau(e_{wait}) \leq \tau(e_s),\; and\; Sema(e_{wait}) = \sigma) \;\; \right\}$$

[2] While these statements are equivalent with those made in the statement of the algorithm, some notation has been changed to make the proofs that follow more readable

Fact 2.3.2 *$R(\sigma)$ is a set of Signal events on σ generated during the call to modify.*

$$R(\sigma) = \left\{ \begin{array}{c} e_{sig} | \tau(e_{sig}) \not\geq \overline{\max}(W(\sigma)) \; and \; \tau(e_{sig}) \not\geq \tau(e_w), \\ and \; Sema(e_{sig}) = \sigma \end{array} \right\}$$

Fact 2.3.3

$$k_\sigma = \mid W(\sigma) \mid$$

Fact 2.3.4 *$T(\sigma)$ is a set of virtual timestamps based on the timestamps of events in $R(\sigma)$.*

$$T(\sigma) = \left\{ \begin{array}{lcl} m(e_r) | m(e_r) & = & modify\,(\tau(e_w), Set \bigcup \{e_s\}, \tau(e_r), depth \text{ - } 1), \\ & & e_r \in R(\sigma) \end{array} \right\} \quad (2.1)$$

Fact 2.3.5 *Within a call to modify, τ_m is determined by two relationships.*

for each $1 \leq \sigma \leq \Sigma$,

$$M_\sigma = \overline{\min}_{k_\sigma}(T(\sigma)) \quad (2.2)$$

$$\tau_m = \overline{\max}(\tau(e_s), M_1, ..., M_s) \quad (2.3)$$

Fact 2.3.6 *Let i be the task executing e.*

By Equation 2.3, if $\tau^{\#}(e) \leq \tau_m = \overline{\max}(\tau(e_s), M_1, ..., M_s)$*:*

$$\tau^{\#}(e)[i] \leq \left\{ \begin{array}{ll} \tau(e_s)[i] & if \; \tau(e_s)[i] \geq M_\sigma[i], \; 1 \leq \sigma \leq s \\ M_\sigma[i] & if, \; for \; some \; \sigma, \; M_\sigma[i] > \tau(e_s)[i] \\ & and \; M_\sigma[i] = \overline{\max}(M_1[i], ..., M_s[i]) \end{array} \right. \quad (2.4)$$

The following hold whenever the timestamp $\tau(e)$ for event e is updated by the Expand algorithm.

Fact 2.3.7 *If event e is a Signal event, then:*

$$\tau(e) = \overline{\max}(\tau(e), \tau(e^p))$$

Fact 2.3.8 *If event e is a Wait event, then:*

$$\tau(e) = \overline{\max}(\tau(e), \tau(e^p), \mathit{modify}\ (\tau(e), \tau(e), \tau(e), \mathit{depth}))$$

2.3.3 A Proof of Safety

Since the fundamental difference between Algorithm 1.3 and Algorithm 2.1 is in the recursive function *modify*, this proof shows that the timestamps produced by repeated calls to *modify* combine to create a safe result.

To make claims about the results of a *modify* call, it is necessary to examine the operations within the function itself. The set W represents the set of Wait events which must precede e_w and e_s in Observations 2.1.1 and 2.1.2 (represented in the Algorithm by τ_{in} and $\tau(\hat{e})$). Set W must be of the correct size because a k^{th} minimum of the set of modified timestamps, T, could miss an ordering relation if W is too small, or could add an incorrect relation if W is too big.

Lemma 2.3 shows that W will always contain a sufficient number of Wait events to satisfy the Observations 2.1.1 and 2.1.2 . Because Algorithm 2.1 enlarges timestamps to order previously unordered events in safe partial orders, the algorithm is unsafe when $\tau(e')[j]$ is too large. Since the set W is generated by comparing event timestamps with the $\tau_{set} = \overline{\max}(\{\tau(e_{set}) : e_{set} \in Set\})$, Lemma 2.2 supports Lemma 2.3 by showing that if an event e that could be in W and $\tau(e) \leq \tau_{set}$, then e precedes an event in Set and should be included in W.

The correctness of a call to *modify* also depends on the set R, because R represents the group of Signal events k_σ of which precede e_w and e_s in Observation 2.1.2. Lemma 2.4 shows by contradiction that if a Signal event e_{sig} happens before e_w or any Wait event in W, then e_{sig} is in R, making R the correct set of Signals for e_w and e_s.

Lemma 2.2 *Let τ be a set of timestamps representing a safe partial order and let τ_{set} be the timestamp such that:*

$$\tau_{set} = \overline{\max}(\{\tau(e_{set}) : e_{set} \in Set\})$$

If $\tau(e) \leq \tau_{set}$ *then there exists at least one* $e_{set} \in Set$ *such that* $e \prec e_{set}$.

Proof: By contradiction.

Let $\tau(e) \leq \tau_{set}$.

Assume $e \not\prec e_{set}$, for all $e_{set} \in Set$.

Assume without loss of generality that e is in task j.

Because e is in task j and $e \not\prec e_{set}$, then $\tau(e)[j] > \tau(e_{set})[j]$, for all $e_{set} \in Set$.

Since $\tau(e)[j] > \tau(e_{set})[j]$ for all e_{set}, then $\tau(e)[j] > \tau_{set}[j]$.

But $\tau(e)[j] \leq \tau_{set}[j]$, for all j. Contradiction.

Therefore, there exists at least one $e_{set} \in Set$ such that $e \prec e_{set}$.

□

Lemma 2.3 *If Set is a set of events such that* $e_w \in Set$*, then for every execution of P where* $e_s \rightarrow e_w$ *and* $e_{set} \rightarrow e_w$*, for all* e_{set} *in Set, the number of Wait events on* σ *that happen before* e_w *is at least the* k_σ *as computed during the call*

$$\tau_m = modify\,(\tau(e_w), Set, \tau(e_s), depth)$$

Proof: Let $\tau_{set} = \overline{\max}(\{\tau(e_{set}) : e_{set} \in Set\})$.

First, it must be shown that $W(\sigma)$ is a set of Wait events such that $\forall e_{wait} \in W(\sigma)$, either $e_{wait} \prec e_s, e_{wait} \prec e_w$, or there exists an $e_{set} \in Set$ such that $e_{wait} \prec e_{set}$.

By Fact 2.3.1 if $e_{wait} \in W(\sigma)$ and e_{wait} is an event in task i, then either $\tau(e_{wait})[i] \leq \tau(e_s)[i]$, or $\tau(e_{wait})[i] \leq \tau_{set}[i]$.

Case 1: $\tau(e_{wait})[i] \leq \tau(e_w)[i]$

When $\tau(e_{wait})[i] \leq \tau(e_w)[i]$, $e_{wait} \prec e_w$.

Case 2: $\tau(e_{wait})[i] \leq \tau(e_s)[i]$

When $\tau(e_{wait})[i] \leq \tau(e_s)[i]$, $e_{wait} \prec e_s$.

Case 3: $\tau(e_{wait})[i] \leq \tau_{set}[i]$

When $\tau(e_{wait})[i] \leq \tau_{set}[i]$, at least one $e_{set} \in Set$ exists such that $e_{wait} \prec e_{set}$ by Lemma 2.2.

Therefore, because $e_s \rightarrow e_w$, and $e_{set} \rightarrow e_w, \forall e_{set} \in Set$, then all the Wait events in $W(\sigma)$ happen before e_w.

k_σ is the number of Wait events on σ in $W(\sigma)$ (Fact 2.3.3).

At least k_σ Wait events on σ happen before e_w.

□

Lemma 2.4 *Consider a Wait event, e_w, a Signal event, e_s, a set of events, Set, and a call to modify:*

$$\tau_m = modify\,(\tau(e_w), Set, \tau(e_s), depth)$$

Let $W(\sigma)$ be the set of Wait events for some semaphore σ generated in the call to modify and let e_{wait} be an arbitrary Wait event in $W(\sigma)$. Let e_{sig} be a Signal event in P on semaphore σ.

If there is any execution E of P where $e_{sig} \rightarrow e_{wait}$ and $e_{sig} \rightarrow e_w$, then $e_{sig} \in R(\sigma)$.

Proof: By contradiction.

Assume $e_{sig} \rightarrow e_{wait}$ and $e_{sig} \rightarrow e_w$ in execution E of P and $e_{sig} \notin R(\sigma)$.

Since $e_{sig} \notin R(\sigma)$, either $\tau(e_{sig}) \geq \tau(e_w)$ or $\tau(e_{sig}) \geq \overline{\max}(W(\sigma))$.

Case 1: $\tau(e_{sig}) \geq \tau(e_w)$

Let i be the task executing e_w.

Since $\tau(e_{sig})[i] \geq \tau(e_w)[i]$, so $e_w \prec e_{sig}$ which contradicts the assumption that $e_{sig} \rightarrow e_w$ in E.

Case 2: $\tau(e_{sig}) > \overline{\max}(W(\sigma))$

Let i be the task executing e_w.

Since $\tau(e_{sig})[i] \leq \overline{\max}(W(\sigma))[i]$ and $\tau(e_{wait}) \in \tau_\sigma$, we have $\tau(e_{sig})[i] \leq \tau(e_{wait})[i]$.

Since $\tau(e_{sig})[i] > \tau(e_{wait})[i]$, we have $\tau(e_{wait})[j] \prec \tau(e_{sig})$, which contradicts the assumption that $e_{sig} \rightarrow e_{wait}$ in E.

In both cases, we get a contradiction, so only the signals on σ in $R(\sigma)$ can happen before bothe$_w$ and an $e_{wait} \in W(\sigma)$ in executions of P.

□

The proof of safety of the Recursive Expand algorithm is a case by case examination of the recursion tree of *modify*. The first case that the proof will examine is the leaves the tree where $depth = 0$. Because *modify* returns the unchanged timestamp of a Signal event and that timestamp already belongs to a safe partial order, *modify* is safe.

Lemma 2.5 *Let* $\tau_m = modify\,(\tau(e_w), Set, \tau(e_s), 0)$ *where Set is an arbitrary set of events in P.*

If $\tau^{\#}(e) \leq \tau_m$ *then* $e \rightarrow e_w$ *for every execution of P where* $e_s \rightarrow e_w$.

Proof: When $\tau_m = modify\,(\tau(e_w), -, \tau(e_s), 0)$, $\tau_m = \tau(e_s)$.

Because $\tau^{\#}(e) \leq \tau_m = \tau(e_s)$, $e \prec e_s$.

Since $e \prec e_s$ then $e \rightarrow e_w$ in every execution of P where $e_s \rightarrow e_w$.

□

To understand how a Lemma will be proved for all interior nodes of the recursion tree, it is useful to examine the simplest meaningful case where *modify* can be called: $depth = 1$ and Set is composed of a single Wait event, e_w. Under these conditions, two possible cases occur when the event e_s is examined: the i^{th} element of the timestamp of the event preceding e_s will either be less than the i^{th} element of the timestamp of e_s, or, less than the i^{th} element

of the k^{th} minimum of the set of modified timestamps, M_σ. If the former case is true, the result is safe because $\tau(e_s)$ is safe. In the latter case, it can be shown that at least one Signal event in R whose timestamp has an i^{th} element larger than the i^{th} element of M_σ. Since this is true, Lemma 2.5 can be used to show that the modified timestamp associated with that Signal event is safe.

Lemma 2.6 *Let* $\tau_m = modify\,(\tau(e_w), \{e_w\}, \tau(e_s), 1)$.

If $\tau^{\#}(e) \leq \tau_m$ *then* $e \rightarrow e_w$ *for every execution of* P *where* $e_s \rightarrow e_w$.

Proof: Let E be an arbitrary execution of P where $e_s \rightarrow e_w$ and e be an event in Task i where $\tau^{\#}(e) \leq \tau_m$.

Since $\tau^{\#}(e)[i] \leq \tau_m[i]$, Fact 2.3.6 states:

$$\text{either } \tau^{\#}(e)[i] \leq \tau(e_s)[i] \text{ or } \tau^{\#}(e)[i] \leq M_\sigma[i] \text{ for some } 1 \leq \sigma \leq s$$

Case 1: $\tau^{\#}(e)[i] \leq \tau(e_s)[i]$.

When $\tau^{\#}(e)[i] \leq \tau(e_s)[i]$, $e \prec e_s$.

Since $e \prec e_s$ and $e_s \rightarrow e_w$, then $e \rightarrow e_w$.

Case 2: $\tau^{\#}(e)[i] \leq M_\sigma[i]$.

At most $k_\sigma - 1$ timestamps in $T(\sigma)$ have i^{th} components strictly less than $M_\sigma[i]$.

By Lemma 2.4, $R(\sigma)$ contains all Signal events $e_r \in R(\sigma)$ such that $e_r \rightarrow e_w$ in E.

Since k_σ Wait events are known to happen before e_w in E, at least k_σ Signal events from $R(\sigma)$ must also occur for e_w to happen.

Therefore, at least one $e_r \in R(\sigma)$ exists for which $e_r \rightarrow e_w$ in E and $M_\sigma[i] \leq m(e_r)[i]$.

Because $\tau(e) \leq m(e_r) = modify\,(\tau(e_w), Set \bigcup \{e_s\}, \tau(e_r), 0)$ and $e_r \rightarrow e_w$ in E, we can apply Lemma 2.5 to show that $e \rightarrow e_w$ in E.

Since E was chosen arbitrarily, $e \rightarrow e_w$ in every execution where $e_s \rightarrow e_w$.

□

A more general case of the interior node where $depth = 1$ is complicated by Set containing e_w and an arbitrary number of Signal events. The number of Wait events in W must be the number that must occur given that all the Signal events in Set precede e_s. Lemma 2.3 is used to show that W and k_σ are correct when this occurs. Otherwise, the structure of the proof of the Lemma for the general case is similar to the case where Set is limited.

Lemma 2.7 *Let Set be an arbitrary set of events containing e_w and*
$\tau_m = modify\,(\tau(e_w), Set, \tau(e_s), 1)$.

If $\tau^{\#}(e) \leq \tau_m$, then $e \rightarrow e_w$ for every execution of P where $e_s \rightarrow e_w$ and $e_{set} \rightarrow e_w, \forall e_{set} \in Set$.

Proof: Let E be an arbitrary execution of P where $e_s \rightarrow e_w$ and $e_{set} \rightarrow e_w, \forall e_{set} \in Set$. Let e be an event in Task i.

When $\tau^{\#}(e)[i] \leq \tau_m[i]$, Fact 2.3.6 shows:

$$\text{either } \tau^{\#}(e)[i] \leq \tau(e_s)[i] \text{ or } \tau^{\#}(e)[i] \leq M_\sigma[i] \text{ for some } 1 \leq \sigma \leq s$$

Case 1: $\tau^{\#}(e)[i] \leq \tau(e_s)[i]$.

When $\tau^{\#}(e)[i] \leq \tau(e_s)[i]$, $e \prec e_s$.

Since $e \prec e_s$ and $e_s \rightarrow e_w$, $e \rightarrow e_w$.

Case 2: $\tau^{\#}(e)[i] \leq M_\sigma[i]$.

By Lemma 2.3, at least k_σ Wait events on σ happen before e_w in execution E.

At most $k_\sigma - 1$ timestamps in $T(\sigma)$ have i^{th} components strictly less than $M_\sigma[i]$.

By Lemma 2.4, $R(\sigma)$ contains all Signal events e_r on semaphore σ such that $e_r \rightarrow e_w$ in E.

Since k_σ Wait events are known to happen before e_w in E, at least k_σ Signal events from $R(\sigma)$ must also occur before e_w in E.

Therefore, at least one $e_r \in R(\sigma)$ exists for which $e_r \rightarrow e_w$ in E and $M_\sigma[i] \leq m(e_r)[i]$.

Because $\tau(e)[i] \leq m(e_r)[i] = \mathit{modify}\,(\tau(e_w), Set \bigcup \{e_s\}, \tau(e_r), 0)$ and because $e_r \rightarrow e_w$ and $e_s \rightarrow e_w$ in E, we can apply Lemma 2.5 to show that $e \rightarrow e_w$ in E.

Since E was chosen arbitrarily, $e \rightarrow e_w$ in every execution where $e_s \rightarrow e_w$ and $e_{set} \rightarrow e_w$, for all $e_{set} \in Set$.

□

The proof of the case for an arbitrary recursion depth is an inductive proof on the *depth*. Lemmas 2.7 and 2.5 form the base cases for $depth = 1$ and $depth = 0$ respectively. As with the case in Lemma 2.7, the proof of the case of arbitrary depth j employs two cases. In the first case, the i^{th} component of the timestamp $\tau^\#$ for an event e in Task i is less than the predecessor of the event under examination e_w. The result is safe because the modified timestamp of e_w is the first to be incorrect. In the second case, $\tau^\#(e)[i]$ is less than the k^{th} minimum of a set T of timestamps produced by calls to *modify* based on timestamps of Signal events in R. Lemmas 2.3 and 2.4 show that k and R are both constructed to yield a safe result. T is composed of safe timestamps by the inductive hypothesis. Therefore, the resulting timestamp τ_m is safe.

Lemma 2.8 *Let* $\tau_m = \mathit{modify}\,(\tau(e_w), Set, \tau(e_s), depth)$.

If $\tau^\#(e) \leq \tau_m$, *then* $e \rightarrow e_w$ *for every execution of* P *where* $e_s \rightarrow e_w$ *and* $e_{set} \rightarrow e_w, \forall e_{set} \in Set$.

Proof:

We prove this Lemma by induction on the recursion depth, *depth*.

We use as a base case, $depth = 1$. Lemma 2.7 shows that Lemma 2.8 holds when $depth = 1$. Lemma 2.5 shows that Lemma 2.8 holds when $depth = 0$.

We now consider the case for an arbitrary $depth = j$, where $j > 1$; and assume that Lemma 2.8 holds for all values of *depth* between 0 and $j - 1$.

Let E be an arbitrary execution of P where $e_s \rightarrow e_w$ and $e_{set} \rightarrow e_w, \forall e_{set} \in Set$. Let e be an event in Task i where $\tau^{\#}(e) \leq \tau_m$. We now show that $e \rightarrow e_w$ in E.

Since $\tau^{\#}(e)[i] \leq \tau_m[i]$, Fact 2.3.6 states:

$$\text{either } \tau^{\#}(e)[i] \leq \tau(e_s)[i] \text{ or } \tau^{\#}(e)[i] \leq M_\sigma[i] \text{ for some } \sigma, 1 \leq \sigma \leq s$$

Case 1: $\tau^{\#}(e)[i] \leq \tau(e_s)[i]$.

When $\tau^{\#}(e)[i] \leq \tau(e_s)[i]$, $e \prec e_s$.

Since $e \prec e_s$ and $e_s \rightarrow e_w$, $e \rightarrow e_w$.

Case 2: $\tau^{\#}(e)[i] \leq M_\sigma[i]$.

By Lemma 2.3, at least k_σ Wait events on σ happen before e_w in execution E.

At most $k_\sigma - 1$ timestamps in $T(\sigma)$ have i^{th} components strictly less than $M_\sigma[i]$.

By Lemma 2.4, $R(\sigma)$ contains all Signal events e_r on semaphore σ such that $e_r \rightarrow e_w$ in E.

Since k_σ Wait events are known to happen before e_w in E, at least k_σ Signal events from $R(\sigma)$ must also occur before e_w happens.

Therefore, at least one $e_r \in R(\sigma)$ exists for which $e_r \rightarrow e_w$ in E and $M_\sigma[i] \leq m(e_r)[i]$.

Because $\tau(e)[i] \leq m(e_r)[i] = \mathit{modify}\ (\tau(e_w), Set \bigcup \{e_s\}, \tau(e_r), 0)$ and $e_r \rightarrow e_w$, $e_s \rightarrow e_w$, and $e_{set} \rightarrow e_w, \forall e_{set} \in Set$ in E, we can apply the inductive hypothesis and conclude that $e \rightarrow e_w$ in E.

Since E was chosen as an arbitrary execution where $e_s \rightarrow e_w$ and $e_{set} \rightarrow e_w, \forall e_{set} \in Set$, we conclude $e \rightarrow e_w$ in every execution where these conditions are met.

□

Since Lemma 2.8 proves safety for an arbitrary interior node on the recursion tree where a Signal event e_s is examined, and the first call made to *modify* is uses only the Wait event e_w, these first calls must be proved safe.

Lemma 2.9 *Let* $\tau_m = modify\,(\tau(e_w), \{e_w\}, \tau(e_w), j)$.

If $\tau^{\#}(e) \leq \tau_m$, *then* $e \prec e_w$.

Proof: Let E be an arbitrary execution of P.

Given $R(\sigma)$ from Fact 2.3.2:

$$T(\sigma) = \left\{ \begin{array}{lcl} m(e_r)|m(e_r) & = & modify\,(\tau(e_w), \{e_w\}, \tau(e_r), j-1), \\ & & e_r \in R(\sigma) \end{array} \right\} \tag{2.5}$$

$$M_\sigma = \overline{\min}_{k_\sigma}(T(\sigma)), 1 \leq \sigma \leq s \tag{2.6}$$

$$\tau_m = \overline{\max}(\tau(e_w), M_1, ..., M_s) \tag{2.7}$$

Let i be the task executing e in E. By Equation 2.7, if $\tau^{\#}(e) \leq \tau_m$:

$$\tau^{\#}(e)[i] \leq \left\{ \begin{array}{ll} \tau(e_w)[i] & \text{if } \tau(e_w)[i] \geq M_\sigma[i],\ 1 \leq \sigma \leq s \\ M_\sigma[i] & \text{if, for some } \sigma,\ M_\sigma[i] > \tau(e_w)[i] \\ & \text{and } M_\sigma[i] = \overline{\max}(M_1[i], ..., M_s[i]) \end{array} \right. \tag{2.8}$$

Case 1: Assume $\tau^{\#}(e)[i] \leq \tau(e_w)[i]$.

When $\tau^{\#}(e)[i] \leq \tau(e_w)[i]$, $e \prec e_w$.

Case 2: Assume $\tau^{\#}(e)[i] \leq M_\sigma[i]$.

At most $k_\sigma - 1$ timestamps in $T(\sigma)$ have i^{th} components strictly less than $M_\sigma[i]$.

By Fact 2.3.1, $e_{wait} \in W(\sigma)$ iff $e_{wait} \prec e_w$ because $Set = \{e_w\}$ and τ is safe.

By Fact 2.3.3, at least k_σ Wait events happen before e_w.

By Lemma 2.4, $R(\sigma)$ contains all Signal events e_r on semaphore σ such that $e_r \rightarrow e_w$ in E.

Since k_σ Wait events are known to happen before e_w in E, at least one $e_r \in R(\sigma)$ exists for which $e_r \rightarrow e_w$ in E and $M_\sigma[i] \leq m(e_r)[i]$.

Because $m(e_r) = \mathit{modify}\,(\tau(e_w), \{e_w\}, \tau(e_r), j-1)$:

$$\text{if } e_r \rightarrow e_w \text{ and } \tau^{\#}(e) \leq m(e_r), \text{ then } e \rightarrow e_w \tag{2.9}$$

by Lemma 2.8.

Since E was chosen arbitrarily, $e \rightarrow e_w$ in every execution, and $e \prec e_w$.

□

Finally, a theorem states that the timestamps produced by the Recursive Expand algorithm are safe. It shows that by Lemma 2.9 that in an analysis of a trace of an arbitrary execution of a program P there is no first timestamp which is made unsafe by application of the Algorithm. Therefore the events in the generated partial order all are safe.

Theorem 2.10 *The timestamps produced by the Expand Algorithm are safe.*

Proof: Assume that the contrary is true: that the timestamp updates produced by the Expand Algorithm are not safe.

Let the timestamps initially assigned to the events in P be safe.

If the Expand Algorithm produces unsafe timestamps, then there must be a first update that changes the safe timestamp $\tau(e)$ to an unsafe timestamp $\overline{\tau}(e)$. At the start of this update, the timestamps assigned to all events are safe.

Case 1: e is a Signal event

$$\overline{\tau}(e) = \overline{\max}(\tau(e), \tau(e^p))$$

by Fact 2.3.7.

Let e' be any event such that $\tau^{\#}(e) \leq \overline{\tau}(e)$.

Thus either $\tau^{\#}(e') \leq \tau(e)$ and $e' \prec e$ (since $\tau(e)$ is safe) or $\tau^{\#}(e') \leq \tau(e^p)$ and $e' \prec e^p \prec e$, contradicting our original assumption that $\overline{\tau}(e)$ is unsafe.

Since $\tau(e)$ is safe e, $\overline{\tau}(e)$ can only become unsafe if it inherits an unsafe timestamp from e^p. Because $\overline{\tau}(e)$ is the first unsafe timestamp, $\overline{\tau}(e)$ must be safe if e is a Signal event.

Case 2: e is a Wait event

$$\overline{\tau}(e) = \overline{\max}(\tau(e), \tau(e^p), \tau_m)$$

where

$$\tau_m = \mathit{modify}\,(\tau(e), \{e_w\}, \tau(e), \mathit{depth})$$

by Fact 2.3.8.

Let e' be any event such that $\tau^{\#}(e') \leq \overline{\tau}(e)$, and e' is in Task i.

Either $\tau^{\#}(e') \leq \tau(e)$, $\tau^{\#}(e') \leq \tau(e^p)$, or $\tau^{\#}(e') \leq \tau_m$.

If $\tau^{\#}(e') \leq \tau(e)$ or $\tau^{\#}(e') \leq \tau(e^p)$, $\overline{\tau}(e)$ inherits safe components from $\tau(e)$ or $\tau(e^p)$ by the argument in case 1.

We conclude that $\tau^{\#}(e') \leq \tau_m$ and by Lemma 2.9, $e' \prec e$.

Thus $e' \prec e$ for any e' where $\tau^{\#}(e) \leq \overline{\tau}(e)$, contradicting our assumption that $e' \not\prec e$..

Since $\overline{\tau}(e)$ is the only timestamp changed during the update, the timestamps produced by the update are safe, contradicting the assumption.

Therefore, the timestamps produced by the Expand Algorithm are safe.

□

Chapter 3
An Implementation

A test of the changes made to Algorithm 1.3 to create Algorithm 2.1 is to use both algorithms on many real programs and compare the results. Unfortunately, too few programs are currently available with recorded trace information to prove that the Recursive Expand Algorithm orders more events in partial orders used in trace-based analysis with certainty.

To overcome this difficulty, a testing application called tracerC has been written using the C++ programming language. This program does not use actual program traces, but, it generates random sequences of events using a counting semaphore model and then analyzes these event sequences as if they are real traces. The results of algorithmic analysis are compared against timestamps assigned to events by a brute force trace analyzer that exactly determines the "always happens before" relationship.

3.1 tracerC - A C++ Test Implementation

The tracerC program uses a simplified representation of program traces consisting of an ordered list of Wait and Signal events. Each event is composed of a randomly assigned semaphore, a Task Id (to indicate which task performed each event) and a type (Signal or Wait). As events are entered in the trace only when the associated operation completes, every prefix of the trace contains at least as many signal events on each semaphore as wait events.

The "program" corresponding to this trace format can be viewed as in Figures 1.1, 2.1, 2.2, and 2.3. The events are grouped in vertical lists by task. Within each vertical list, the events appear in the same order as they occurred in the trace. A "state" of the program corresponds to selecting a (possibly empty) prefix of each vertical list. Usually some program states are unreachable by any execution. For example, in Figure 2.3 the state where Tasks A and B have executed no events but Task C has executed its first event (W_1) is unreachable as the wait cannot complete until after semaphore 1 has been signaled.

Once an event sequence has been generated, a brute force analyzer performs a depth-first search on the program's (exponentially large) state space to discover all of the reachable states. If event e_1 has been executed in every reachable state where event e_2 has been executed then the brute force analyzer indicates that e_1 "must happen before" e_2. Performing this test for each pair of events allows the brute force analyzer to compute the "always happens before" relationship for the "program".

After brute force analysis is complete, the Rewind Algorithm is run on a copy of the unordered test sequence. With the events in the sequence in a safe partial order, Algorithm 1.3 is run on the trace representation of the "program" and the results compared to the partial order generated by the brute force analyzer. If the two orders are not identical, Algorithm 2.1 is run on the sequence beginning with $depth = 1$. If a comparison between the partial order produced by the Recursive Expand algorithm and the order found by brute force shows the two still not to be identical, the value of *depth* is increased and the trace is analyzed again. This process is repeated until the two partial orders match, or a specified maximum depth is reached. If the Recursive Expand algorithm is unable to find the partial order at maximum depth that the brute force analyzer finds, the trace and the partial order is written to a file for later analysis by hand.

The tracerC application keeps track of the CPU time consumed during each call of the brute force analyzer, the Expand algorithm, and the Recursive Expand algorithm and records this information in a file.

If a real program makes conditional branches that could be affected by races, then the the brute force analyzer may overestimate the "must happen before" relationship. However, even the overestimated "must happen before" relationship allows the race affecting the conditional branch to be flagged. The experiments presented here concentrate on how often the "always happens before" relationship computed by the other algorithms matches that returned by the brute force analyzer and how often the Recursive Expand algorithm locates event orders that the Original Expand algorithm misses.

3.2 Event Generating Schemes

At this time, tracerC generates random events by a single method which can be used to automatically generate and analyze traces for a specified amount of time. The generation scheme uses the parameters MaxTasks, MaxSemaphores, and NumEvents. To generate a trace, tracerC randomly selects NumTasks (between 2 and MaxTasks) and NumSemaphores (between 1 and MaxSemaphores) using a uniform random distribution. Once these values have been set, tracerC executes a loop to create the trace of a non-blocking execution.

The generator loop is presented in Figure 3.1. This method uses random numbers to generate the Task id and semaphore, and a 50/50 chance of an event being a Wait or a Signal, when the partially constructed trace contains more Signals than Waits. Otherwise a Signal event is generated.

3.3 Frequency Data

To increase the value of the results of tracerC, it is useful to know how often tracerC is actually examining different traces. The tracerC program attempts to determine the quantify the number of traces that are generated randomly by maintaining a database using *Postgres*. The following information about a random trace is used as a key:

```
while less than NumEvents generated do
      select T randomly between 1 and NumTasks
      select sem randomly between 1 and NumSemaphores
      if more signals than waits have been generated for sem
            then
                  flip a coin
                  if heads then output: W_sem by task T
                  if tails then output: S_sem by task T
            else
                  output: S_sem by task T
end while
```

Figure 3.1: Coin Toss program generator.

- The number of Tasks.
- The number of Semaphores.
- An array of the number of Signal events per Task.
- An array of the number of Wait events per Task.
- An array of the number of Signal events per Semaphore.
- An array of the number of Wait events per Semaphore.

To exactly determine the number of traces that the generator produces would require the development of a canonical form for describing all traces. The determination of whether a given trace matches a canonical form is complex because of Task Id and Semaphore number re-labeling. For the purposes of this work, it was decided that an approximate approach be used because of programming simplicity.

Use of the Postgres trace archive has shown that out of approximately 1500 traces in the database, only two were found to be similar under the system used to describe traces in the previous chapter. Since the attributes used to describe these traces are necessary but

not sufficient to show the different between traces, these traces may not be structurally identical.

Chapter 4
Results

A series of tests have been performed to determine the effectiveness of Algorithm 2.1. In all cases, reported times are CPU time inside and outside the Operating System kernel. Since such times include time for memory page faults, the time data tracerC produces is affected by the load of the machine on which the application is being run. It is assumed that this interference is significant but small and that figures are within five percent of what they would be on a machine with little user load.

The most significant result of the experiments is that the Recursive Expand Algorithm is a significant improvement over the original Expand Algorithm. Figure 4.1 shows that the percentage of traces where algorithmic and brute force analysis produce identical traces is significantly higher than the percentage produced by the original Expand algorithm when Algorithm 2.1 is used with the recursion depth set to its lowest level. Figure 4.2 shows that the Recursive Expand algorithm shows a similar improvement in the number of timestamps that are found incorrect when the produced partial orders are compared. Since this percentage is strongly correlated with the number of correct traces, it is difficult to determine if the partial orders that are incorrect contain more ordered pairs of events as recursion depth increases.

It should also be noted that it is assumed that real programs will exhibit more regular ordering properties than the inferred programs of the randomly generated traces used here.

It is hoped that the percentages of correct traces of real programs will be higher than those for random programs.

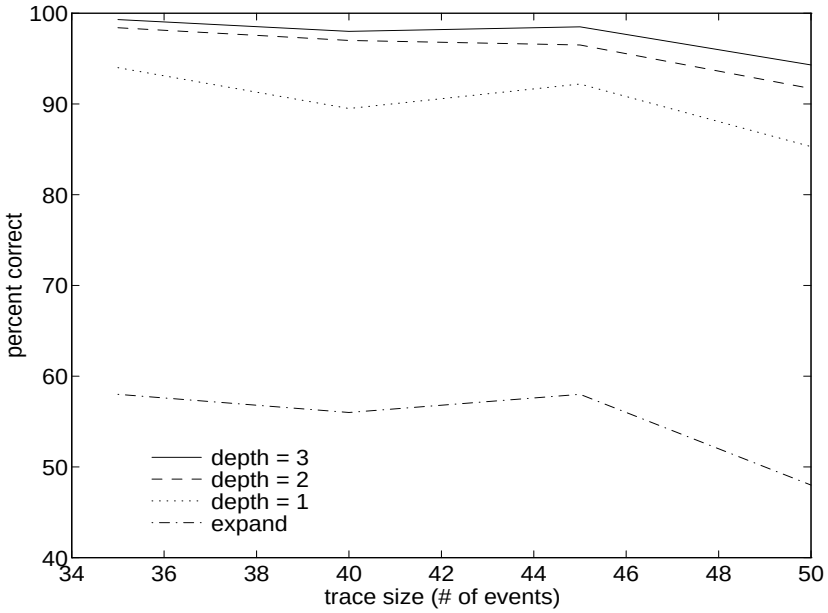

Figure 4.1: Percentage of traces the are correctly analyzed by Expand and Recursive Expand at $depth = 1$, $depth = 2$, and $depth = 3$.

Unfortunately, the more complex analysis that the improved algorithm uses takes more time to execute. Figures 4.3 and 4.4 show the mean execution times of the Recursive Algorithm compared to brute force analysis and the original Expand algorithm. The results show that the Recursive Algorithm provides analysis which is less time intensive than brute force analysis when recursion depth is low but that inefficiency in the current implementation makes the Recursive analysis unattractive at higher depth because of execution time. The large confidence intervals around the mean times seem to be the result of a significant number of executions with long running times in the tail of the frequency distributions of execution times with all analysis tools. (See Figures 4.5 to 4.8.)

The range of execution times seems to be related to the number of different possible executions that the inferred program can exhibit. For brute force analysis this means an increased number of concurrency states that the program must examine. For Recursive Expand this

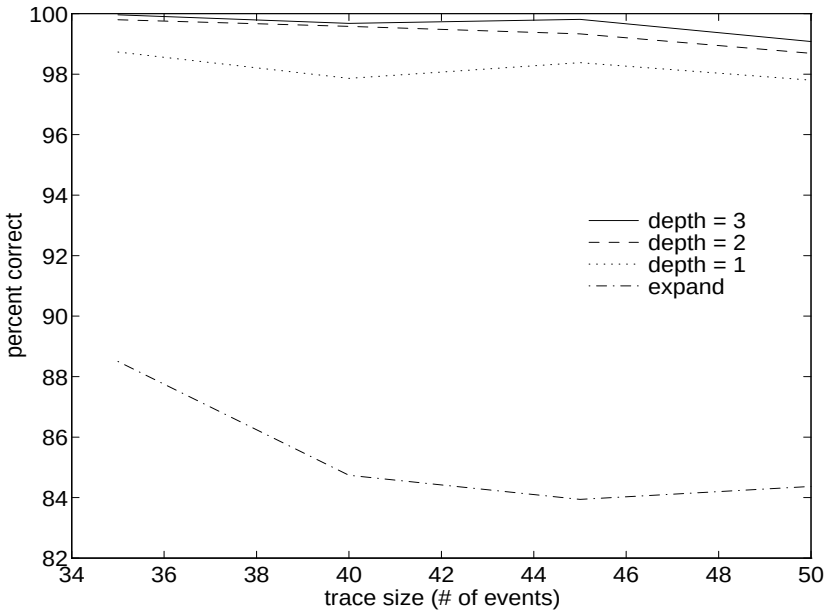

Figure 4.2: Percentage of timestamps that are correctly determined in partial orders produced by Expand and Recursive Expand at $depth = 1$, $depth = 2$, and $depth = 3$.

is related to the number of Signal events compared to any Wait event which increases the size of the recursion tree in Recursive Expand. Examination of the raw data reveals that these two factors do not necessarily correlate and a significant number of cases exist where algorithmic methods at high depth ran faster than brute force as well as where brute ran faster than Recursive Expand.

The use of small traces to collect this execution time data may lead to an incorrect inference concerning relative execution times. It is believed that for real traces of thousands of events, brute will be impractical whereas Recursive Expand at depth = 1 or depth = 2 will be practical.

Testing on traces of large size was not attempted because the brute force analyzer takes up large amounts of memory when it hashes every concurrency state it finds. This number of states is so significant that the number of states hashed by the brute force analyzer for traces of 90 events filled up the 128 megabyte swap space of a Sparcstation 2 before the

execution was killed. This attempted analysis was ended before any meaningful data could be saved in an output file.

The total number of traces of various sizes produced across several runs of tracerC are shown in Table 4.1.

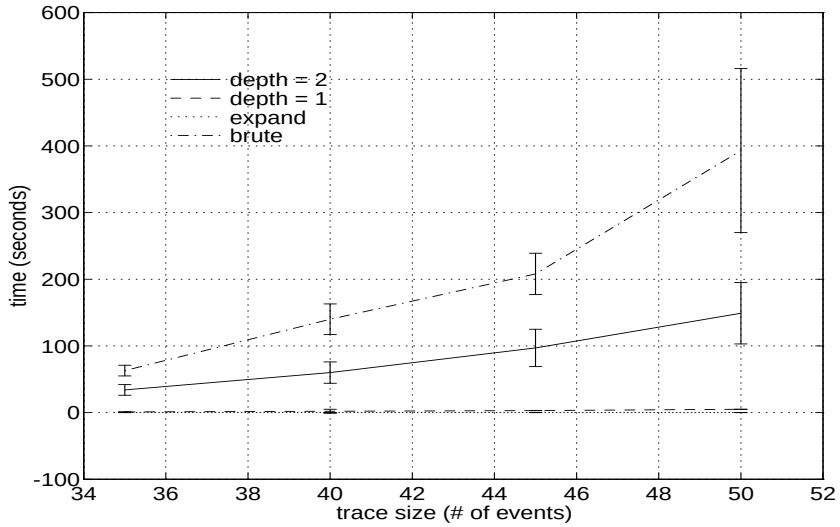

Figure 4.3: Mean running times for brute, Expand, and Recursive Expand at $depth = 1$ and $depth = 2$ with confidence intervals at 95%

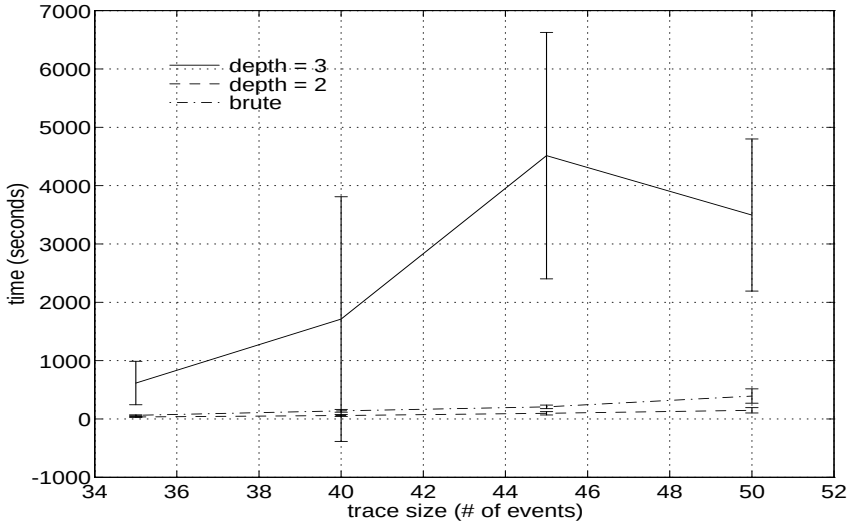

Figure 4.4: Mean running times for brute and Recursive Expand at depth = 2 and depth = 3 with confidence intervals at 95%

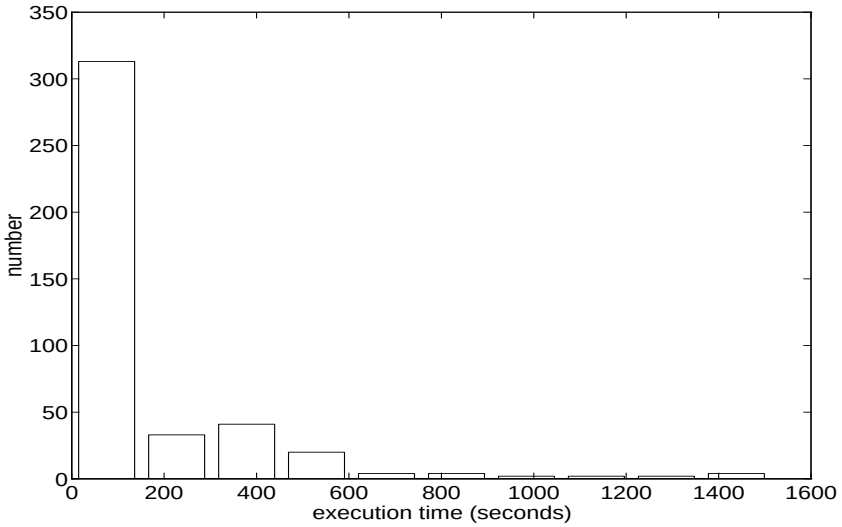

Figure 4.5: Frequency histogram of execution times for brute on traces of 40 events

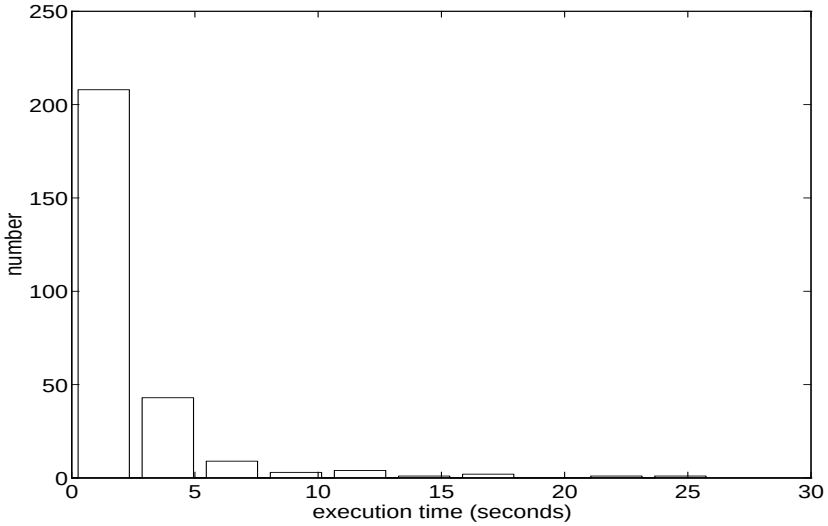

Figure 4.6: Frequency histogram of execution times for Recursive Expand at depth = 1 on traces of 40 events

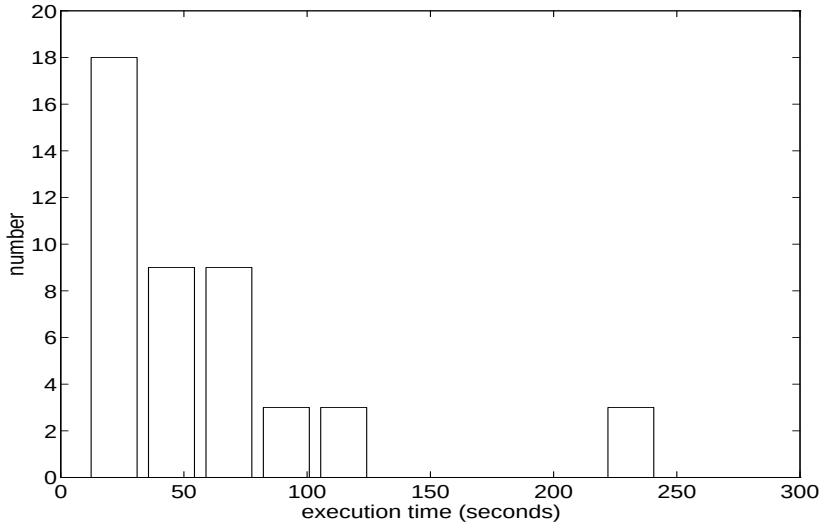

Figure 4.7: Frequency histogram of execution times for Recursive Expand at depth = 2 on traces of 40 events

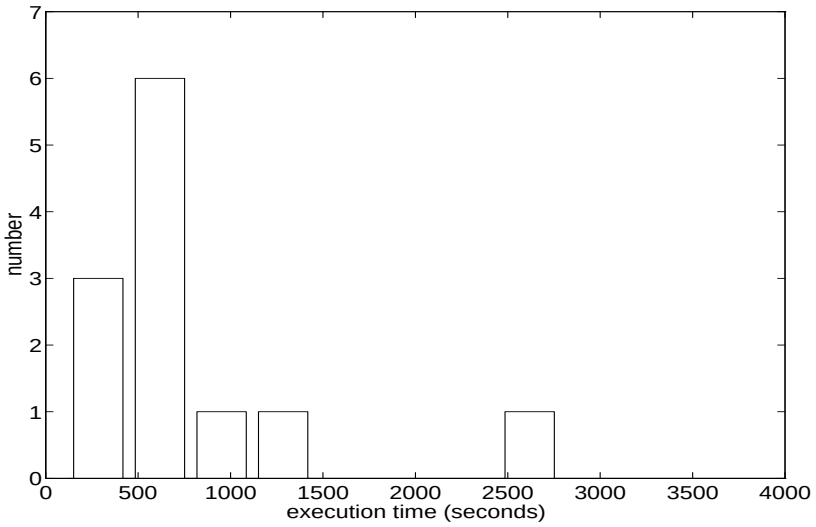

Figure 4.8: Frequency histogram of execution times for Recursive Expand at depth = 3 on traces of 40 events

# of events	# of traces
35	545
40	426
45	397
50	157

Table 4.1: Total numbers of traces for each number of events.

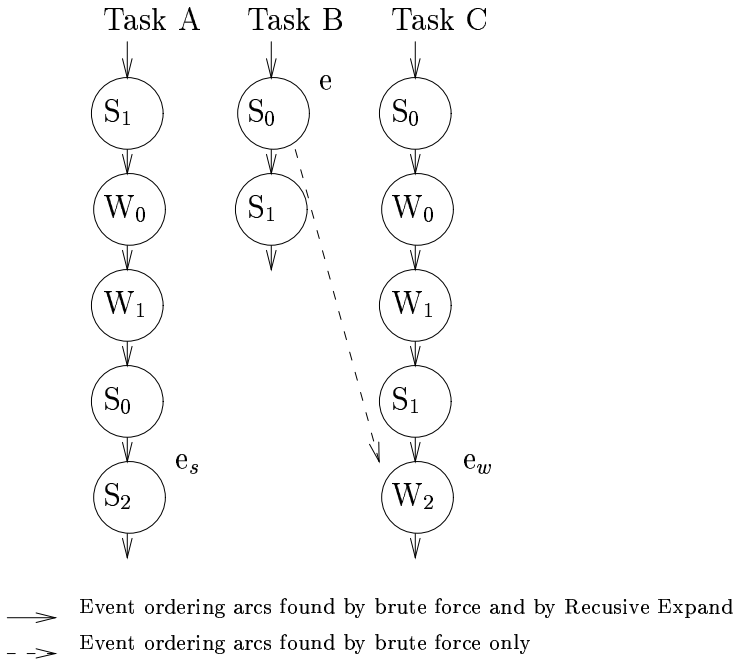

Figure 4.9: A trace where Recursive Expand fails. At least one of the Signal events in Task B is required: the Signal event e is needed if e_w is executed and Task A blocks and the Signal event following e is needed if Task A completes and Task C blocks

An interesting test trace was discovered that breaks the Recursive Expand Algorithm. The first trace (See Figure 4.9) displays a complex locking behavior that Recursive Expand is unable to detect. In order for the events e_s and e_w to both occur, two Signals on semaphore zero and two Signals on semaphore one are required. Two of the four Signal events in Tasks A and C follow a symmetric pattern of Wait events, execution can reach either e_s or e_w and cause the other of the two Tasks to block. Therefore at least one of the Signal events in Task B is required: the Signal event e is needed if e_w is executed and Task A blocks and the Signal event following e is needed if Task A completes and Task C blocks.

A bug in the implementation of the algorithm was discovered after most of these test runs were made. It was found that when k is larger than the size of R, the $\overline{\min}_k$ function returned the $\overline{\max}$ of the R, rather than a timestamp set to infinity to indicate that event could not occur.

After this bug was corrected, the nine traces of forty events that produced errors at $depth = 3$, were rerun using corrected code. None of the results differed with partial orders produced by the code with the bug.

Chapter 5
Summary

5.1 Conclusion

Parallel programs with explicit synchronization can be notoriously difficult to debug. One important kind of correctness and performance debugging tool determines and presents the temporal ordering relationships between various synchronization operations in the program. Writing such a tool which is accurate, practical, and efficient is difficult because event ordering analysis frequently addresses questions which are at least NP-hard for arbitrary programs. These problems can be avoided by basing analysis on program traces rather than the program itself, and by further confining analysis to the search for a conservative approximation of the ordering relationships that may exist between events which can be found in polynomial time.

Helmbold, McDowell and Wang presented a series of three algorithms for extracting useful ordering information from sequential traces. The type of output that these algorithms sought to produce is a partial ordering of events reflecting the "always happens before" relationship. An event "always happens before" another event if the event precedes the second event in all executions of the program that are consistent with the trace in numbers and types of events. These algorithms used semaphore style synchronizing constructs with vector timestamps very similar to those found in [Fid88]. The first algorithm was for initialization, and was similar to the one presented by Fidge [Fid88] and Mattern [Mat88].

It read a trace and provided each event with a timestamp based on the order of events in the trace. This order of events is not general enough because it describes the single execution found in the trace. The Rewind Algorithm took the partial order produced by the initialization algorithm and decoupled Wait events from specific Signal events. The partial order generated by the Rewind Algorithm is too general because the Rewinding process deletes some ordering relationships between events that may actually occur in all executions of the program consistent with the trace. The last algorithm, called Expand, attempted to recover some of these relationships.

This work describes a new algorithm called Recursive Expand which was developed while implementing and testing the Expand algorithm. By analyzing the situations where the Expand algorithm failed, it was possible to state several new observations about how synchronization events are ordered and these observations were used to create the new algorithm. The Recursive Expand algorithm was presented with the observations that motivated its creation and a proof that the new algorithm produces only partial orders which are safe (can be used to debug all program executions consistent with the trace) from partial orders which are already safe. Lastly, this work provides a synopsis of experimental data that showed the Recursive Expand algorithm capable of finding many more event orders than the old Expand algorithm.

If parallel programming is to become much simpler and less prone to errors due to poor synchronization, either a more comprehensible programming paradigm must be invented or advanced software tools must be written. Since efforts over the last decade to find an efficient approach to programming in multithreaded environments have not fully escaped synchronization problems, better debugging software would seem to be a worthwhile pursuit. The material presented here is a step in that direction.

5.2 Future Work

With an implementation of the Recursive Expand Algorithm completed and tested, there are several open issues that can be explored and some problems that must be addressed. These include:

1. Efficiency. At the present time, the Recursive Expand is very time expensive. To make a more practical tool, it would be worth the effort to attempt to reduce this execution time. Since it recurses on the sets of events that it finds in any particular call without regard for calls either before or after, it may actually duplicate effort. This problem could be solved by introducing some sort of state id mechanism for a particular call to modify and use this to guarantee that only unique states are actually executed.

2. An Analysis Tool. Time must be spent incorporating the Recursive Expand algorithm into a tool capable of reading and analyzing real traces. Such a tool called browser currently uses the Initialize, Rewind, and (original) Expand algorithms to locate races in programs in IBM Parallel Fortran. The tool could be modified both to accept a broader range of languages and incorporate the new Recursive Expand algorithm.

3. New Experiments. Since only one type of randomly generated trace was used as input for the data presented here, other generation schemes could be used and their effects on the success rate and execution time of the Recursive Expand algorithm determined. Such a new generation scheme might be similar to the first, except that the probability of generating a Wait event increases as the ratio of Signals to Waits increases. Other schemes might model more complex synchronization structures using Signal and Wait events, like a simple, properly nested message passing system where random numbers indicate the Tasks sending and receiving the message.

4. Depth Heuristics. It may be possible to find a few heuristics capable of determining the maximum necessary recursion depth for a specific program, or a good minimum

recursion depth to be used for a first pass with a tool using Recursive Expand. Since this could cut down on time spent running the tool, and possibly reduce the number of spurious races reported to the programmer, this would be very useful.

Bibliography

[EGP89] P. A. Emrath, S. Ghosh, and D. A. Padua. Event synchronization analysis for debugging parallel programs. In *Supercomputing '89*, November 1989. Reno, NV.

[Fid88] C. J. Fidge. Partial orders for parallel debugging. In *Proc. Workshop on Parallel and Distributed Debugging*, pages 183–194, May 1988.

[HMW93] D. P. Helmbold, C. E. McDowell, and J. Z. Wang. Determining possible event orders by analyzing sequential traces. *IEEE Transactions on Parallel and Distributed Systems*, 1993. Also UCSC Tech. Rep. UCSC-CRL-91-36.

[Mat88] F. Mattern. Virtual time and global states of distributed systems. In M. Cosnard, editor, *Proceedings of Parallel and Distributed Algorithms*, 1988.

[MC92] J. Mellor-Crummey. Compile-time support for efficient data race detection in shared-memory parallel programs. In *Supercomputing '92*, pages 169–177, October 1992. Dallas, TX.

[McD89] C. E. McDowell. A practical algorithm for static analysis of parallel programs. *Journal of Parallel and Distributed Computing*, June 1989.

[NM90] R. H. B. Netzer and B. P. Miller. On the complexity of event ordering for shared-memory parallel program executions. In *Proc. International Conf. on Parallel Processing*, volume II, pages 93–97, 1990.

[NM91] R. H. B. Netzer and B. P. Miller. Improving the accuracy of data race detection. *SIGPLAN Notices (Proc. PPOPP)*, 26(7):133–144, 1991.

[Tay83] R. N. Taylor. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica*, 19:57–84, 1983.