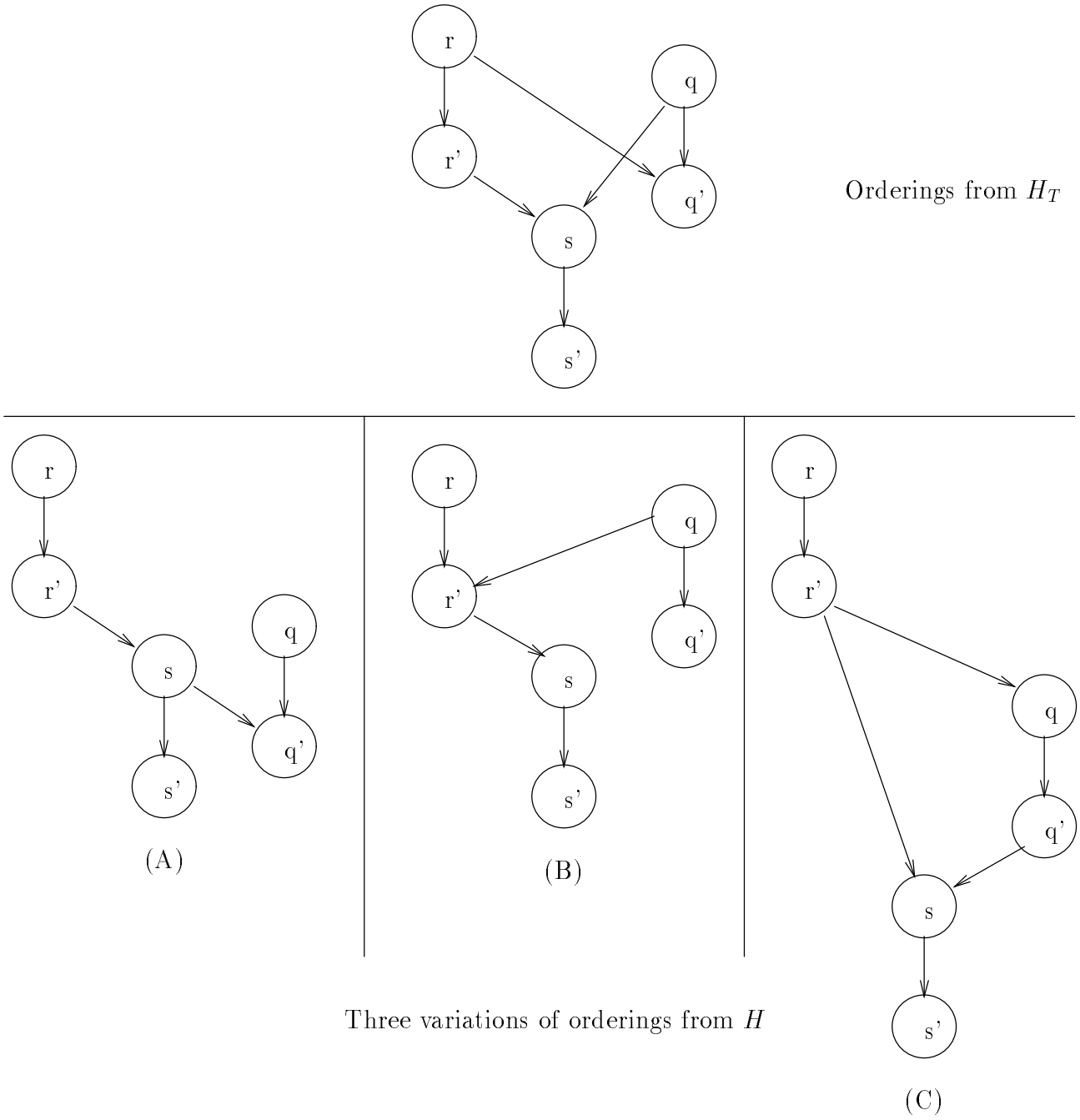

Figure A.1: Possible orderings for Lemma 2. Variable v is read in block B(s,s') and written in blocks B(r,r') and B(q,q'). Variable v is assumed to have different values at s in H and H_T. The orderings shown are only a subset - both H and H_T are totally ordered.

[IBM88] *Parallel FORTRAN language and library reference.* IBM, 1988.

[LMC87] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. on Computers*, C-36(4):471–482, April 1987.

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

[MC88] B. P. Miller and J-D. Choi. Breakpoints and halting in distributed systems. In *Proc. Int. Conf. on Distributed Computing Systems*, June 1988.

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

[NM89] R. Netzer and B. P. Miller. Detecting data races in parallel program executions. Technical Report 894, University of Wisconsin-Madison, November 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.

[Tay84] R. N. Taylor. Debugging real-time software in a host-target environment. Technical report, U.C. Irvine Tech. Rep. 212, 1984.

[Wan90] J-Z. Wang. Debugging parallel programs by trace analysis. Technical report, Masters Thesis UCSC-CRL-90-11, 1990.

(at this point in $\hat{H}$). However, if r is a blocked wait event in $\hat{H}$, then it would also be unable to appear in H. Therefore q must represent a branch where the executions producing H_T and H go different directions, and the branch condition in q is evaluated to different values by the executions corresponding to H_T and H. Since H and $\hat{H}$ are identical up to and including q, they will evaluate the branch condition to the same value. Now we have an expression (the branch condition represented by q) which is evaluated to different values in H_T and $\hat{H}$. Lemmas 2 and 3 show that there is an appropriate race in P_T, completing the proof of the theorem. □

References

[AP87] T. R. Allen and D. A. Padua. Debugging fortran on a shared memory machine. In *Proc. International Conf. on Parallel Processing*, pages 721–727, 1987.

[CKS90] D. Callahan, K. Kennedy, and J. Subhlok. Analysis of event synchronization in a parallel programming tool. In *Proceedings of Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP), SIGPLAN Notices*, pages 21–30, March 1990.

[DS90] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Proceedings of Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 1990.

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

[EP88] P. A. Emrath and D. A. Padua. Automatic detection of nondeterminacy in parallel programs. In *Proc. Workshop on Parallel and Distributed Debugging*, pages 89–99, May 1988.

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

[HMW90a] D. P. Helmbold, C. E. McDowell, and J. Z. Wang. Analyzing traces with anonymous synchronization. In *Proc. International Conference on Parallel Processing*, August 1990.

[HMW90b] D. P. Helmbold, C. E. McDowell, and J. Z. Wang. Detecting data races by analyzing sequential traces. Technical report, U. of Calif. Santa Cruz, UCSC-CRL-90-57, 1990.

[HMW90c] D. P. Helmbold, C. E. McDowell, and J-Z. Wang. Traceviewer: A graphical browser for trace analysis. Technical report, U. of Calif. Santa Cruz, UCSC-CRL-90-59, 1990.

[HMW91] D. P. Helmbold, C. E. McDowell, and J. Z. Wang. Detecting data races from sequential traces. In *Proc. of Hawaii International Conference on System Sciences*, pages 408–417, 1991.

Proof: By induction on the number of synchronization events preceding s in H.

For the base case, s is the first synchronization event (a "task start" event) in H so each variable has its initial value at s in H. If some variable v does not have its initial value at s in H_T then some assignment to v was made in H_T prior to s. Let $\mathrm{B}(r, r')$ be the block in H_T containing this write to v. We now see that $r \rightarrow s \rightarrow s'$ in H_T and $s \rightarrow r'$ in H (whether or not r' appears in H), so since v is read in $\mathrm{B}(s, s')$ there is a race in P_T (Definition 6).

For the inductive step we assume that, for all blocks $\mathrm{B}(\hat{s}, \hat{s}')$ where $\hat{s} \rightarrow s$ in H, if the value of any variable read in $\mathrm{B}(\hat{s}, \hat{s}')$ at $\hat{s}$ in H is different from its value at $\hat{s}$ in H_T, then there is a data race in P_T. We now show that the same holds for the block $\mathrm{B}(s, s')$.

Let v be any variable read in $\mathrm{B}(s, s')$ whose value at s in H differs from its value at s in H_T. Consider the latest write to v occurring before s in H, and let $\mathrm{B}(r, r')$ be the block containing this latest write. Clearly, $r \rightarrow s'$ in H. We can assume that $r \rightarrow r' \rightarrow s \rightarrow s'$ in H since if $s \rightarrow r'$ in H we immediately get that $\mathrm{B}(s, s')$ and $\mathrm{B}(r, r')$ comprise a race. For the same reason, we can assume that $r \rightarrow r' \rightarrow s \rightarrow s'$ in H_T.

In order for v to have a different value at s in H_T, either the expression assigned to v in the block $\mathrm{B}(r, r')$ is evaluated to a different value in H_T, or there is some other write to v (not in $\mathrm{B}(r, r')$) occurring between r and s in H_T. In the first case we can apply Lemma 2 and the inductive hypothesis to show that there is a relevant race in M. The second case is slightly more complicated.

Let $\mathrm{B}(q, q')$ be the block containing this other write to v. Because of the location of the accesses to v, we have that $r \rightarrow q'$ and $q \rightarrow s \rightarrow s'$ in H_T (see the top of Figure A.1). Now consider where $\mathrm{B}(q, q')$ appears in H. If $s \rightarrow q'$ in H[10] then blocks $\mathrm{B}(q, q')$ and $\mathrm{B}(s, s')$ form a race in P_T (Figure A.1 variation A). If $q \rightarrow r'$ in H then, since $r \rightarrow q'$ in H_T, blocks $\mathrm{B}(q, q')$ and $\mathrm{B}(r, r')$ are a race (Figure A.1 variation B).

This leaves us with the possibility that $r' \rightarrow q \rightarrow q' \rightarrow s$ in H (Figure A.1 variation C). However, block $\mathrm{B}(q, q')$ can not contain a write to v since block $\mathrm{B}(r, r')$ contained the last write to v in H. The resulting contradiction shows that this final case is impossible. □

We are now ready to prove Theorem 1. Let $e \prec f$ with respect to P_T and H be a global trace of an execution of P where $f \rightarrow e$. Because $e \prec f$ with respect to P_T and $f \rightarrow e$ in H, every global history of P_T differs from H at some point. Given a global history H' of an execution of P_T we can count the number of events before the first difference between H' and H. Let $\hat{H}$ be any global history for P_T with a maximal number of events before its first difference with H. Let event r performed by task t in H be the first event where H differs which $\hat{H}$, and q be the previous event in H performed by task t.

By the maximality of $\hat{H}$, r is never performed at this point in an execution of P_T. Either some other event $\hat{r} \neq r$ follows q in task t of P_T or r is a blocked Wait event

[10] If H represents a deadlocked execution, then q' may not be present in H. However, we can still say $s \rightarrow q'$ and deduce the presence of the race.

Another assumption made in the appendix is that a "task start" and a "task termination" synchronization event appears for every task (even those that are "deadlocked") in the program. This ensures that every sequential block is bracketed by two synchronization events.

Given a program P and a trace T of an execution of P, we define the inferred program P_T as in the introduction and H_T to be a global history corresponding to T (Note that H_T to both a trace of P and a trace of P_T). We wish to show that if there are no races in P_T, then $e \prec f$ with respect to P_T implies $e \prec f$ with respect to P. Recall that a race in P (or P_T) is a pair of conflicting blocks, $\mathrm{B}(e, e')$ and $\mathrm{B}(f, f')$, such that there exists two global traces of P (resp. P_T) where $f \rightarrow e'$ in the history corresponding to one trace and $e \rightarrow f'$ in the history corresponding to the other.

We now restate the theorem to be proven.

Theorem 4: *If event $e \prec f$ in P_T, then for each global history of P where $f \rightarrow e$ there is a data race, $B(g, g')$ and $B(h, h')$, in P_T such that either $g \rightarrow f$ or $h \rightarrow f$ in that global history of P.*

The proof is rather complex, and has been broken down into several steps. Recall that the program's input (including the results of clock calls and random number generators) has been fixed.

Lemma 2: *Let H be the global history corresponding to a trace of an execution of program P_T and s be a synchronization event in H. If an expression in block $B(s, s')$ has a different value in H and H_T, then there is a variable v read in $B(s, s')$ such that:*

1. *the value of v at s in H is different from the value of v at s in H_T, or*
2. *in H_T there is a write to v by some other task between s and the expression's evaluation, or*
3. *in H there is a write to v by some other task between s and the expression's evaluation.*

Proof: If the task starts executing the statements in $\mathrm{B}(s, s')$ in the same state (with respect to those variables read in the block), and no outside task changes the relevant shared memory, then all expressions in the block will be evaluated to the same result. □

The latter two cases of Lemma 2 immediately imply that there is a race in P_T between block $\mathrm{B}(s, s')$ and the block containing the write to v by the other task. Showing that the first case also implies that P_T has a race requires the following lemma.

Lemma 3: *Let H be a global history corresponding to an execution of P_T and s a synchronization event in H. If some variable v read in $B(s, s')$ has a value at s in H that is different from its value at s in H_T, then there is a race in P_T. In addition, this race includes either $B(s, s')$ or a block $B(r, r')$ where $r \rightarrow s$ in H.*

This paper contains a series of polynomial time algorithms for extracting useful information from sequential traces with anonymous synchronization. The first algorithm, used to compute the initial vector timestamps, is due to Fidge and Mattern [Fid88,Mat88]. The other algorithms systematically manipulate these vectors of timestamps in order to discover pairs of events that must be ordered in every execution which is consistent with the trace.

Some parallel programming environments view a parallel execution as a linear sequence of events. We feel that this is misleading – an execution is more properly viewed as a partial ordering on the events. Fidge and Mattern have pioneered the use of time vectors to represent these partial orders. We have extended this approach by using time vectors to analyze sets of executions rather than just capturing a single execution.

A working trace analyzer has been implemented, and some experiments have been performed. The current implementation analyzes traces generated by IBM Parallel Fortran and includes a graphical trace browser. The trace analyzer reports various data race conditions in parallel programs by finding unordered/concurrent events and variable access conflicts.

Acknowledgements

This work was supported by IBM under agreement SL 88096.

A Appendix

In this section we will need to examine the executions of programs in great detail. Therefore, all global traces and histories will include not only the synchronization events performed, but also events representing global memory accesses and evaluation of conditionals. These detailed global traces are not intended to be recorded, but rather are used conceptually to help prove Theorem 1. At this level of detail, the synchronization operations bounding sequential blocks will generally be separated by (global) memory access and expression evaluation events.

Throughout the appendix we assume that global histories correctly reflect the data dependences of the execution[9] Except in extreme cases (such as the replication of shared variables), executions will have corresponding global histories which reflect their data dependences. These global histories may be difficult to obtain from a trace, and thus we use their existence for proof, rather than algorithmic, purposes.

[9]For example, if the value of variable x read in $B(r, r')$ is the value that was written to x in block $B(w, w')$ then $r \rightarrow w'$ in the global history. Notice that the resolution of the history need not be enough to precisely reflect (or determine) the order of the variable acceses. We require only that the history be consistent with the actual data dependence. In this example the history would be inconsistent if it indicated that $w' \rightarrow r$.

been on identifying the feasible data races in a set of apparent data races. We, on the other hand, are trying to increase the number of feasible data races detected by our algorithms. The approaches appear to be complimentary.

Dinning and Schonberg [DS90] present a method of detecting access anomalies in parallel programs "on-the-fly". They use a mechanism, similar to time vectors, to identify concurrent operations in a program execution. Some compaction methods are used to reduce the storage needed for reader and writer sets. If a variable is involved in multiple data races, then some of those races may not be reported. However, at least one of the data races involving the variable will be reported by their algorithm. They need explicit coordination between tasks in order to construct the partial order execution graph (POEG). The POEG represents the order relations between operations for just one of many possible executions given the same input.

We believe that it is more helpful to analyze sets of executions rather than just one specific execution based on some trace information. We feel that, in terms of detecting data races by trace analysis, it is critical to distinguish the *ordered* events from the *unordered*, potentially *concurrent*, events. In this paper we presented a collection of algorithms that extend previous work in computing partial orders. The algorithms presented compute a partial order containing only *must occur* type orderings from a linearly ordered trace containing anonymous synchronization. The algorithms presented in this paper make few assumptions about specific trace features and can be adjusted to work with traces generated by many parallel systems.

7 Summary

Debugging parallel programs is more difficult than debugging sequential programs. One of the fundamental problems encountered when debugging parallel programs is detecting unintended non-determinacy in parallel programs. Tools which automatically detect non-determinacy can be used to debug timing and synchronization errors when the program is expected to be determinate. The tools we are developing help one find the data races which can lead to non-determinacy. This paper presents a method for detecting data races, which is based on analyzing a program trace from an execution of a parallel program.

When debugging parallel programs, it is critical to find the order and concurrency relationships among operations in the program. One of the most difficult tasks in trace analysis is determining the timing relationships between the events performed by the parallel program. Although several parallel systems include facilities for creating a trace of the significant events, the sequential nature of the trace makes it difficult to determine which events could have happened in either order or in parallel. The problem is made even more difficult in the anonymous synchronization model, where there is no clear correspondence between the blocking and enabling events in the trace. The problem of calculating all safe order relations has been shown to be co-NP-hard by Netzer and Miller [NM90].

- display all information known about the event from the trace,
- highlight all events that must happen before the selected event,
- highlight all events that must happen after the selected event,
- highlight all events that may happen concurrently the selected event, and
- display the program source with the line generating the event highlighted.

6 Related Work

Recently, much research has been directed towards determining the partial ordering of events in parallel and distributed systems. Previous models have assumed point-to-point communication which makes it very easy to determine which events were caused by which other events (e.g. "message received by B from A" is clearly caused by "message sent by A to B"). Unfortunately the synchronization models supported by several parallel programming languages allow for anonymous communication, where the partner is unknown. Examples of anonymous communication include locks, semaphores, and monitors.

Emrath, Ghosh, and Padua [EGP89] present a method for detecting non-determinacy in parallel programs that utilize fork/join and event style synchronization instructions with the `Post, Wait`, and `Clear` primitives. They construct a *Task Graph* from the given synchronization instructions and the sequential components of the program that is intended to show the guaranteed orderings between events. For each `Wait` event node, all `Post` nodes that might have triggered that `Wait` are identified. An arc is then added from the closest common ancestor of these `Post` events to the `Wait` event node. An early version of our Algorithm 3 (without the shadowed events) was motivated by their algorithm. Although their algorithm is simply stated, it may be computationally complex. Rather than repeatedly computing the common ancestor information, we use time vectors to calculate the guaranteed execution order.

Netzer and Miller [NM89,NM91] present a formal model of a parallel program execution. Their model includes fork/join parallelism and synchronization using semaphores. They distinguish between an *actual data race*, which is a data race exhibited by the particular program execution generating the trace, a *feasible data race*, which is a data race that could have been exhibited due to timing variations and an *apparent data race*, one that appeared to have occurred or be possible. Their approach and ours differ in the amount of trust placed in the trace. They rely on the trace for their ordering information. For example, when two tasks try to enter critical regions protected by a binary semaphore, their algorithm will say that the critical regions are ordered. Under their definitions there is neither an actual nor feasible data race even if two tasks write to a shared variable in the critical regions. We view the ordering relationships in the trace with suspicion, and wish to generate race reports in this situation[8]. In their more recent work [NM91] their emphasis has

[8]If the critical regions contain non-commutative operations, then the race to enter the regions can affect the remainder of the execution [Wan90].

manipulation of critical data areas. A *parallel event* is a programming construct permitting explicitly created tasks to synchronize their execution through intertask signaling (as outlined in the previous section).

The IBM Parallel FORTRAN Trace Facility can automatically record important events during the execution of a parallel program, providing useful information about the execution. The Trace Facility produces a series of time-stamped trace records during execution of a parallel program. At least one trace record is generated for each of the following operations:

- Start and end of program execution,
- Origination and termination of tasks,
- Assignment and completion of task work,
- Waiting for tasks to complete work,
- Start and end of parallel loop and parallel case execution, and
- Use of parallel locks and parallel events.

In addition to the time stamp, each trace record identifies the kind of action, the program unit and task performing the action, the virtual FORTRAN processor used, and the actual CPU on which the program unit was executing. Additional information specific to the kind of action may also be recorded.

To build an event history from an IBM trace we need to determine the enabling-blocking pairings for all synchronization events.

1. For the dispatching and scheduling (enabling) events, the corresponding task begin is the blocking event.
2. For the IBM-event construct, Posts enable the Waits in the same cycle, and Waits enable the Posts in the next cycle. Thus Posts and Waits are both blocking and enabling.
3. For task completion (enabling) events, the corresponding event waiting for the task completion is the blocking event.

Modified versions of the algorithms described in the previous section have been implemented in a trace analyzer, and several traces have been analyzed. The trace analyzer can construct the event history and calculate the time vector values for all events in the trace. The final time vectors represent a safe partial order over the events. By comparing the time vectors, the tool distinguishes between the ordered and unordered event pairs. Combining this with variable reference information from START [McD89], the trace analyzer reports those data races which can happen in any execution of the inferred program. The trace analyzer is implemented mainly in C++, and part of the code is implemented in C.

A graphical tool [HMW90c], built on top of the X Window System, has also been implemented to assist programmers in comprehending the trace information recorded during program execution and generated using the above algorithms. The tool allows the user to browse the safe partial order computed and display the detected races. Using a pointing device the user may request that various information related to a selected node be displayed. The following information requests are included:

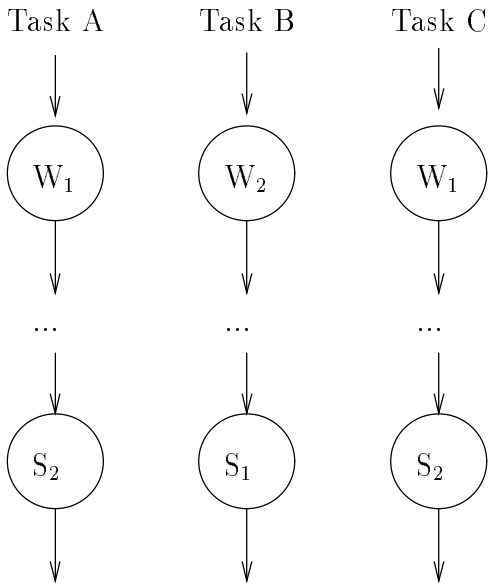

Figure 3.6: Undetected Critical Region

short summary of the IBM-event mechanism and the intuition for the new algorithms follows.

IBM-event based synchronization operates in a cycle with two phases[6]. During the first phase (Post Request Processing), Post operations are enabled and Wait operations are blocked. During the second phase (Wait Request Processing), the opposite holds, Wait operations are enabled and Post operations are blocked. This cycle can repeat indefinitely. The number of Posts or Waits necessary to switch from one phase to the next is determined when the IBM-event is initialized.

In the semaphore model we used the fact that if k Waits were known to always precede a particular Wait operation, then that Wait must be preceded by $k+1$ Posts. This led to Algorithm 3 which takes the kth min of the relevant unblocking events instead of just the min. A similar idea has been applied to the IBM-events. In this case we compute the (lower) *cycle bound*[7] which is a lower bound on the earliest cycle in which the event could occur. As new safe arcs are added to an event history, the cycle bounds may increase allowing still more safe arcs to be added.

5 A Prototype for IBM Parallel Fortran

In IBM Parallel FORTRAN, a task can be explicitly created, assigned work, and waited for until the work assigned to it has been completed. A task can also be implicitly created by parallel loops and parallel cases. Tasks can be executed concurrently. A *parallel lock* can be used to prevent interference between tasks during

[6]This is a simplified description, a complete description can be found in [IBM88] or [HMW90b].

[7]A cycle includes the Post/Wait operations from the first Post in a Post Request Processing phase to the last Wait operation in the following Wat Request Processing phase.

As an example, consider the trace shown in Figure 3.4 (the same events are shown Figure 3.5). Let A_i, B_j, C_k be the ith, jth and kth events in tasks A, B and C respectively. The two unordered Wait events B_1 and C_1 cannot happen concurrently because there is only one Signal (A_1) available for one of them to proceed in every execution of the inferred program. They form two critical regions. In the executions where B_1 occurred before C_1, C_1 becomes the second Wait on the semaphore. Using Algorithm 3, we get time vectors as shown in Figure 3.5(a) where the event pairs {(B_1,C_1), (B_1,C_2), (B_1,C_3), (B_2,C_1), (B_2,C_2), (B_2,C_3)} appear ordered. Similarly, in the executions where C_1 occurred before B_1, event pairs {(B_1,C_1), (B_2,C_1), (B_3,C_1), (B_1,C_2), (B_2,C_2), (B_3,C_2)} are ordered as shown in Figure 3.5(b). At this point, we can conclude that the intersection of these two sets contains event pairs that are not concurrent in any executions, whenever B_1 happened before C_1 or C_1 before B_1. Therefore, {(B_1,C_1), (B_1,C_2), (B_2,C_2), (B_2,C_1)} are unordered sequential event pairs in the critical region, and can be moved from Conc to Seq.

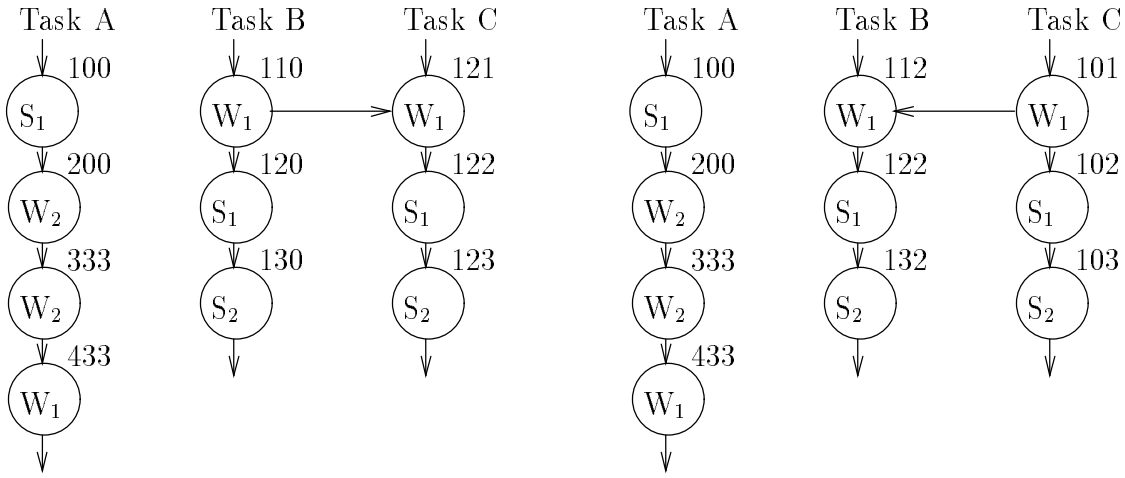

Figure 3.5: Detect Critical Regions

Note that Algorithm 4 does not always succeed as three tasks can conspire to create a critical region that is not detected. For example, consider the inferred program fragment in Figure 3.6. If exactly one signal on semaphore 1 is available (and no signal on semaphore 2) then the three elided regions are mutually exclusive. However, Algorithm 4 only detects the critical regions starting with a Wait on semaphore 1.

4 Generalizing the Semaphore Model

The previous section described algorithms for systematically determining order relationships between events in a counting semaphore model. We have also applied a similar approach to the event-based synchronization mechanism provided by IBM Parallel FORTRAN [IBM88]. The more general nature of the IBM-event mechanism required modifications to the algorithms, especially the Algorithms 3 and 4. The details of the algorithms needed to handle the IBM events can be found in [HMW90b]. A

Algorithm 4: Initially let Conc $= \{\{e, e'\} : \hat{\tau}(e) \parallel \hat{\tau}(e')\}$ and Seq $= \emptyset$.
Repeat the following procedure until no more changes are possible.

Pick any two unordered Wait events e and e' for semaphore S where $(e, e') \in$ Conc.
Let $Get(e, e')$ be the set of Wait events[a] for semaphore S which precede either event e or e' (based on current time vectors $\hat{\tau}$).
Let $Release(e, e') = \{e'' : e''$ is a Signal event using S and e'' precedes e or $e'\} \cup \{e'' : e''$ is a Signal event using S and does not follow either e or e' and e'' is not shadowed with respect to either e or $e'\}$.
Let $s = \mid Release(e, e') \mid$ and $w = \mid Get(e, e') \mid$.

- If $s - w \geq 2 \Longrightarrow e \parallel e'$, i.e., if there are enough Signals for both Waits to precede, then the two Waits can happen concurrently.
- If $s - w = 1 \Longrightarrow \neg(e \parallel e')$, i.e., there is only one Signal for a Wait to precede, then we can conclude that they cannot happen concurrently. The starting points of critical regions have been found. The following procedure is used to determine the unordered sequential event pairs in the critical region.

  1. First, assume that event e happened before e'. Thus $w+1$ Wait events on S happened before e'. Use Algorithm 3 with $k = w + 1$ to calculate a new time vector for event e'. Continue with Algorithm 3 (with the modification that whenever the time vector for e' is calculated, event e is counted when determining k) to obtain a set of temporary time vectors.

     Let Seq_1 be the set of event pairs which are in Conc but are ordered by the temporary time vectors. After obtaining Seq_1, the original time vectors are restored. We can not yet move these events from Conc to Seq since they may be concurrent in executions where e' happens before e.

  2. Now assume that event e' happened before e. Thus e is the $w + 2$nd Wait for S. As before, starting from the original time vectors, we run a modified Algorithm 3 (with the adjustment when the time vector for e is calculated). Let Seq_2 be the set of event pairs which are in Conc and are ordered by the resulting time vectors. Again, the original time vectors are restored after determining Seq_2.

  3. The intersection of Seq_1 and Seq_2 gives the unordered event pairs in the critical regions. We therefore set Seq $=$ Seq $\cup$ $(\text{Seq}_1 \cap \text{Seq}_2)$ and Conc $=$ Conc $-$ $(\text{Seq}_1 \cap \text{Seq}_2)$.

- $s - w \leq 0$ means neither Wait event can precede. In this case, the inferred program has a potential deadlock.

End Algorithm 4.

[a] A Wait on a semaphore can be thought of as "Getting" a shared resource. A Signal can likewise be thought of as a "Release" of a shared resource.

so, the algorithm then finds those unordered sequential event pairs within the critical regions by considering the effect of different execution orders of the two Wait events.

The algorithm calculates two sets. The set Conc contains the potentially concurrent event pairs, while the set Seq contains known unordered sequential event pairs. The event pairs in neither Conc nor Seq are ordered. Initially, every unrelated pair of events are considered potential concurrent. As critical regions are detected, the algorithm moves the appropriate unordered sequential event pairs from Conc to Seq.

3.4 Adjusting the Time Vectors to Determine Concurrency

The previous algorithms compute time vectors representing a safe history. Given any two events e and e', if $\hat{\tau}(e) < \hat{\tau}(e')$ or $\hat{\tau}(e') < \hat{\tau}(e)$ then the two events are ordered. Otherwise, we say that e and e' are unordered in the history. The unordered events need not necessarily be concurrent events. They may be free to occur in either order, but constrained to occur sequentially. In this case, we call them *unordered sequential* events. For example, if the program has a properly implemented lock around a critical region, then different executions may have tasks entering the critical region in different orders. In no execution, however, do two tasks concurrently enter the critical region.

In some cases it is reasonable, and even desirable, to have unordered sequential races. Consider a program consisting of n tasks each taking various amounts of time to compute a subresult. The program's output is the sum of the n subresults. It is reasonable to include a block of code like:

```
Wait(mutex);
total := total + mysubresult;
Signal(mutex);
```

at the end of each task. There are races between each of these blocks since they all write to the shared variable `total`. However, all of the operations on `total` are commutative and associative, so `total` receives the same final value regardless of the order in which these blocks are executed.

Although the general detection of commutative and associative operations on shared data is beyond the scope of this paper, we do provide a way to help distinguish the blocks that are concurrent in some execution of the inferred program from those that are sequentially unordered. Information on whether or not unordered blocks can be concurrent may also be useful to a programmer trying to understand the program's behavior.

Unfortunately, the concurrent relation cannot be determined immediately from the time vectors. It is neccessary that $\hat{\tau}(e) \parallel \hat{\tau}(e')$ for e and e' to be concurrent, but this is not sufficient. As an example, in Figure 3.4, if e = the first Wait in B and e' = the first Wait in C, then even though $\hat{\tau}(e) \parallel \hat{\tau}(e')$, the two Wait events cannot occur at the same time in any execution. Determining whether or not two unordered events can happen concurrently in some execution of the inferred program is an NP-complete problem. Therefore we must settle for an approximate solution.

We now present an algorithm which detects many critical regions and determines the associated pairs of unordered sequential events[5]. The algorithm first determines if a pair of Wait events on the same semaphore starts a pair of critical regions. If

[5]The algorithm may not, however, detect all of the unordered sequential event pairs. This is partly due to the difficulty in detecting all of the safe orderings, and partly due to the many unusual ways that locks can be implemented.

[NM90]. We have presented a series of polynomial time algorithms that find many of the orderings that occur in all executions of an inferred program. Here we bound the execution times of these algorithms. Throughout this section we use m for the total number of events, and n for the number of tasks, in the inferred program. Note that the sum of the components of any time vector is bounded by m and the total of the components of all time vectors is bounded by m^2.

Assertion 1: *The time required by the initialization algorithm (Algorithm 1) is* $O(nm)$.

First we topologically sort the events, so that each unblocking Signal has been assigned a time vector before we consider the corresponding Wait. This takes time $O(m)$ since each event has outdegree at most two (the next event in the task and the corresponding Wait). By keeping an array containing the last time vector assigned for each task and a back pointer to the unblocking Signal events, the relevant previous time vectors can be located in constant time. Order n steps suffice to compute the component-wise maximum. Therefore, the running time of Algorithm 1 is $O(nm)$.

Assertion 2: *The time required by the rewinding algorithm (Algorithm 2) is* $O(nm^3)$.

Since each iteration of Algorithm 2 decreases at least one component of a time vector, there can be at most m^2 iterations through the m events in the trace. By keeping the component-wise min of the time vectors of the Signal events on each semaphore, each event can be processed in $O(n)$ time (including the possible update to the component-wise min after processing Signal events). Therefore, the running time of Algorithm 2 is $O(nm^3)$.

Assertion 3: *The time required by the expanding algorithm (Algorithm 3) is* $O(nm^4)$.

As above, there are at most m^2 iterations. The time for processing Signal events is dominated by the cost of processing Wait events. Finding the set $R(e)$ for Wait events is made easier by storing a pointer to the shadowing Wait event (if any) with each Signal event. Now the set $R(e)$, as well as the value k, can be found with a single pass through the trace, taking $O(n)$ time per event for time vector comparisons. The $k+1$st component-wise minimum can be calculated in $O(nm)$ time using a bucket sort on each component. Therefore, the overall time required by Algorithm 3 is in $O(nm^4)$.

In the above analysis of Algorithms 2 and 3, we used a very pessimistic m^2 bound on the number of iterations required. This is based on the assumption that, for each iteration: only one timestamp is changed, only one component of the timestamp is modified, and the value of the modified component only changes by one. We expect most iterations will modify many of the timestamps by various amounts, particularly when the events are examined in topological order. Thus these algorithms will usually require only $O(m)$ iterations, saving a factor of m over the pessimistic bounds stated above.

Algorithm 3: *Expand:*
Initially $\hat{\tau}(e) = \tau'(e)$ for all events e in the inferred program.
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 $W(S)$ be the set of Wait events on semaphore S;
        let k be the number of Wait events $e_w \in W(S)$ where
            $\hat{\tau}(e_w) < \hat{\tau}(e)$;
        let $R(e) = \{\hat{e} : \hat{e}$ is a Signal event on S, $\hat{\tau}(e) \not\leq \hat{\tau}(\hat{e})$,
            and $\hat{e}$ is not shadowed with respect to $e\}$;
        let $v_s = \overline{\min}_k(\hat{\tau}(\hat{e}) : \hat{e} \in R(e))$;
        set $\hat{\tau}(e) = \overline{\max}(\hat{\tau}(e^p), \tau^{\#}(e), v_s)$;
    else
        set $\hat{\tau}(e) = \overline{\max}(\hat{\tau}(e^p), \tau^{\#}(e))$;
    end if;
end for;

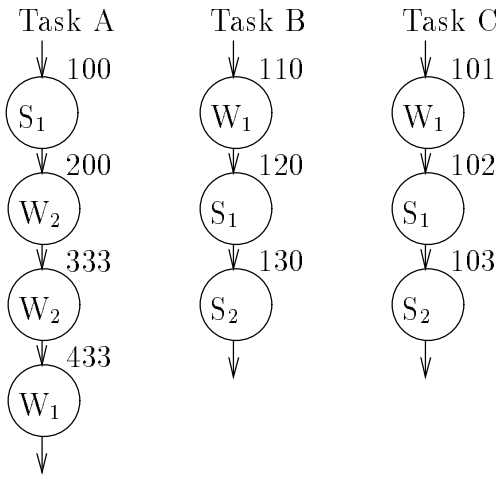

Figure 3.4: Expanding the Safe Order Relation

Figure 3.4 shows the new $\hat{\tau}$ time vectors generated when Algorithm 3 is executed starting with Figure 3.1.

Theorem 3: *Algorithm 3 generates only safe order relations with respect to the inferred program P_T, i.e., for any two distinct events e and $e' \in H$:*

$$\hat{\tau}(e) < \hat{\tau}(e') \Rightarrow e \prec_{P_T} e'$$

3.3 Running Time Analysis

Given an inferred program, P_T, containing two distinguished events, e and e', the problem of determining whether or not $e \prec_{P_T} e'$ has been shown to be intractable

the same semaphore S performed by task T_j, with $\hat{\tau}(e) \parallel \hat{\tau}(e_s)$. Let $H(e, e_s)$ be the subsequence of events performed by T_j containing those events e_j where both $\hat{\tau}(e_j) < \hat{\tau}(e_s)$ and $\hat{\tau}(e_j) \parallel \hat{\tau}(e)$. If any suffix of $H(e, e_s)$ contains more Wait events on S than Signal events on S, then the Signal event e_s is shadowed *with respect to e.*

Definition 12: *Let $H'(e, e_s)$ be the shortest suffix of $H(e, e_s)$ which contains more Wait events than Signal events on S, and let e_w be the first event of $H'(e, e_s)$. We say e_s is* shadowed by *event e_w with respect to e.*

Lemma 1: *Given a Wait event e_w and a Signal event e_s on the same semaphore S, if e_s is shadowed by some event e with respect to e_w then:*

- *event e is a Wait event on semaphore S,*
- *the subsequence of events performed between e and e_s (by the same task) contains as many Signal events as Wait events on semaphore S,*
- *the event e, which shadows e_s with respect to e_w, is unique – whenever e_s is shadowed with respect to some event e', e_s is shadowed by e. We say e is* the shadowing Wait event *corresponding to e_s, and*
- *the correspondence between shadowed Signal and shadowing Wait is one to one, so no event other than e_s is shadowed by e.*

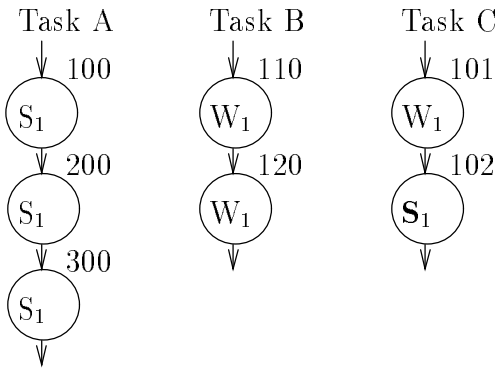

Figure 3.3: Shadowed Signal Event

In the example shown in Figure 3.3, the Signal event in C is shadowed by the Wait in C with respect to the two Wait events performed by task B.

Algorithm 3 is based on the following observation. If e is a Wait event on semaphore S and k other Wait events on S must happen before e, then at least $k + 1$ non-shadowed Signal events happen before e in every execution of the inferred program.

Suppose e is a Wait event for some semaphore S, and there are k other Wait events for S which precede e in every execution of the inferred program. In this case, at least $k+1$ Signal events on S are needed in order to unblock e and its predecessors. We will exploit this fact to enrich the safe event history. As an extreme example, consider the causal trace where task A executes three Signals and task B executes three Waits on the same semaphore, with the obvious Signal-Wait pairings. Figure 3.2(a) shows the result of rewinding. It appears that the only inter-task arcs in the safe history are from the first Signal in task A to the Waits in task B. However, the second Signal in A must happen before the second Wait in B, and the third Signal must happen before the third Wait, as shown in Figure 3.2(b).

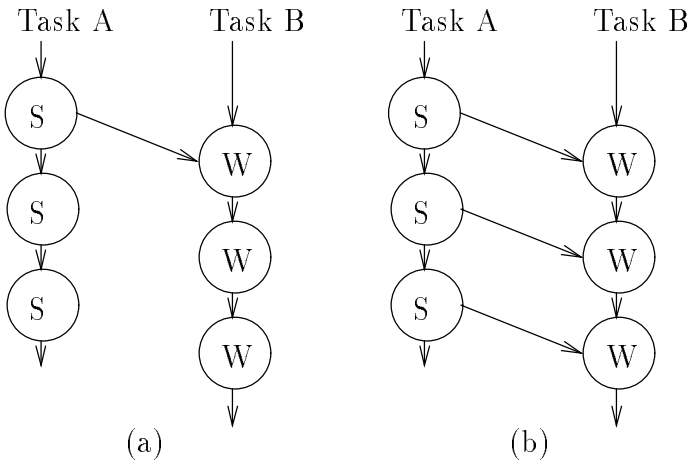

Figure 3.2: Safe Ordering (all Signals and Waits are on the same semaphore)

Additional safe ordering arcs can be found based on another observation. In the example shown in Figure 3.3, the Signal event in C is preceded by a Wait event in the same task. From the indicated time vectors (computed using Algorithm 2), we know that there is a Wait event (the first Wait in B) preceding the second Wait in B. Therefore, at least two Signal events must precede the second Wait in B. The Signal in C could be one of them, but if this is the case, then at least three Signal events are needed to allow the second Wait in B to precede. In any execution of the inferred program, the second Wait in B must happen after at least two Signal events other than the Signal in C. Similarly, the first Wait in B must be preceded by at least one Signal event, not counting the signal in C.

In general, if some Wait event e is known to follow a set of k other Wait events on the same semaphore, then $k+1$ Signals on that semaphore are needed to unblock e and its predecessors. If some Signal used to meet this demand is itself preceded by another Wait on the semaphore which is not in the set, then this additional Wait increases the number of Signals needed to unblock e and its predecessors to $k+2$. Therefore the unblocking done and additional blocking created by including the Signal cancel. When this happens we say that the Signal is shadowed.

Definition 11: *Let $\hat{\tau}$ be an assignment of time vectors to events representing a safe event history. Let e be a Wait event on semaphore S and e_s be a signal event on*

Observe that the only difference between Algorithm 2 and Algorithm 1 occurs when e is a Wait event. Here v_s is the minimum of a set of time vectors which includes the time vector used for v_s in Algorithm 1. Therefore the components of time vectors can only decrease as Algorithm 2 executes.

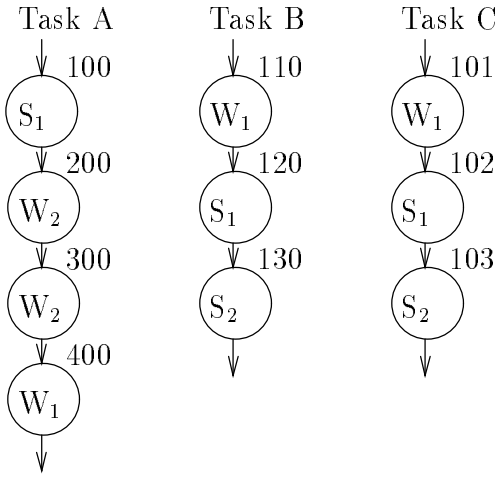

Figure 3.1: Rewinding the Time Vectors

After rewinding, we have a partial order that represents a safe history (with respect to the inferred program). If event e_i has an earlier time vector than e, then e_i happens before e in every execution of the inferred program.

Theorem 2: *Algorithm 2 generates a safe event history with respect to the inferred program P_T, i.e., for any two events e_1, e_2 in the trace:*

$$\tau'(e_1) < \tau'(e_2) \Rightarrow e_1 \prec_{P_T} e_2.$$

Although the τ' timestamps represent a safe event history, they may be too conservative. As an example, the τ' time vectors in Figure 3.1 represent a history where the three Wait events in task A are unrelated to all of the events in tasks B and C. However, it is obvious that in any execution of the corresponding inferred program the W_1 in task A must follow the two S_1 events in tasks B and C, and the second W_2 in task A has to wait until all of the events in B and C have occurred. Our next algorithm increases the time vectors while maintaining a safe event history.

3.2 Expanding the Safe Order Relation

The result of the rewind step is a set of time vectors representing a safe event history. As noted above, it may be an overly conservative safe order relation where some of the safe ordering arcs have been lost during the rewinding procedure. We now describe an algorithm which enriches the partial order while maintaining its safety. The partial order resulting from this process is represented by the time vectors $\hat{\tau}(e)$. Initially, $\hat{\tau}(e) = \tau'(e)$.

3.1 Rewinding the Time Vectors

The τ time vectors resulting from the initialization process represent an unsafe history. It is unsafe because the causal trace associates a particular unblocking Signal event with each Wait event. The algorithm in this section rewinds the time vectors to account for the fact that some executions of the inferred program might use different signal events on the appropriate semaphore to unblock a given Wait. The result of rewinding is a new event history which (usually) does not correspond to a causal trace of any execution of the inferred program, but rather represents orderings which occur in all causal traces of the inferred program, regardless of what execution generated them.

This new history is represented by a new set of time vectors assigned to the events, which we denote by τ'. At the start of Algorithm 2, the $\tau'(e)$ time vectors are initialized to the $\tau(e)$ time vectors computed by Algorithm 1. We now give the intuition behind the rewinding algorithm.

Suppose e is a Wait event, and e_1 and e_2 are the only two Signal events which could have unblocked e. In this case, we only know that either e_1 or e_2 must have happened before e. A global trace for executions of the inferred program might list the three events (with the other events elided) in any of the orders:

$$\begin{array}{ll} \ldots, e_1, \ldots, e, \ldots, e_2, \ldots; & \\ \ldots, e_2, \ldots, e, \ldots, e_1, \ldots; & \\ \ldots, e_1, \ldots, e_2, \ldots, e, \ldots; & \text{or} \\ \ldots, e_2, \ldots, e_1, \ldots, e, \ldots. & \end{array}$$

However, we can conclude that those events that precede both e_1 and e_2 in every execution of M must also occur before e. Formally if $e_a \prec_M e_1$ and $e_a \prec_M e_2$ then $e_a \prec_M e$. The rewind algorithm repeatedly uses this fact to obtain a safe event history.

Algorithm 2: (Rewind)
Initially, for all events e in the trace, $\tau'(e) = \tau(e)$.

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 $e^s_1 \ldots e^s_k$ be all the Signal events on S;
    set $v_s = \overline{\min}(\tau'(e^s_1), \ldots, \tau'(e^s_k))$;
  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;

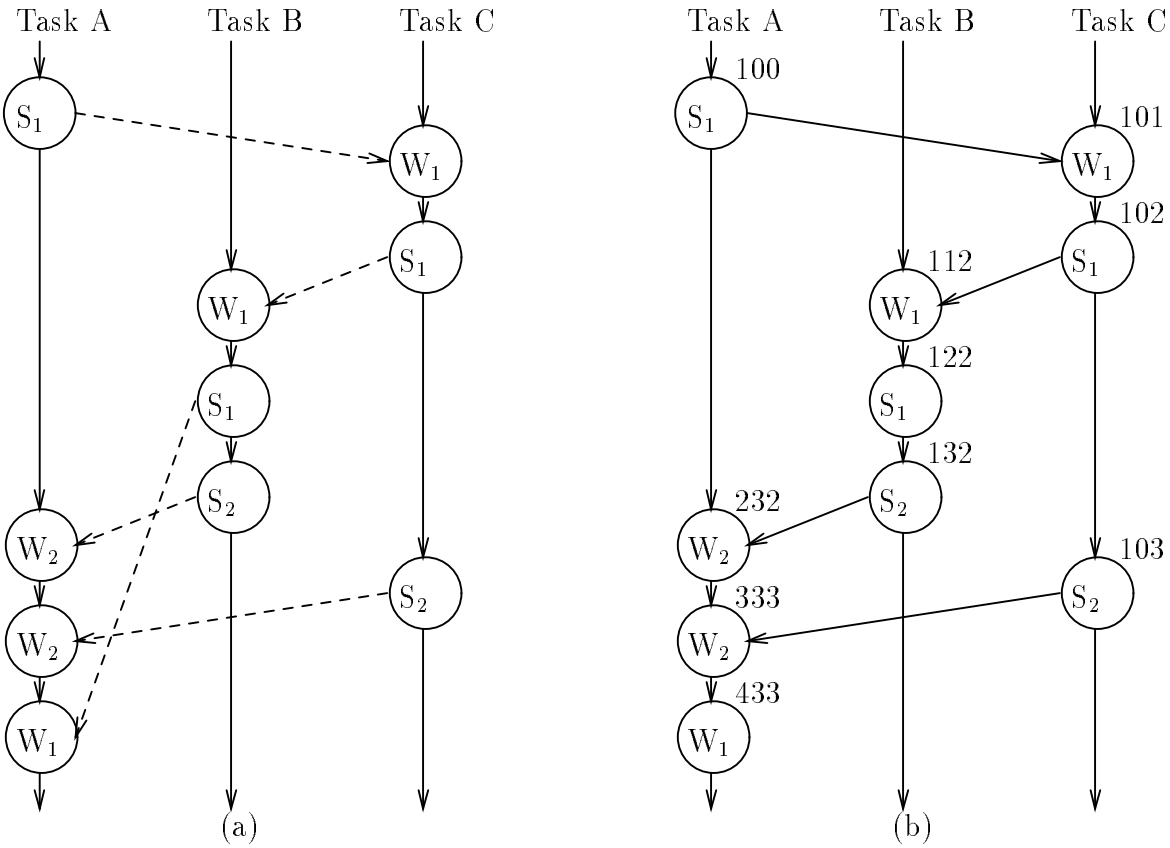

Figure 2.1: Initializing the Time Vectors: (a) depicts a causal trace. The dotted edges show the Signal-Wait pairings. (b) shows the partial order and time stamps resulting from Algorithm 1.

3 Analyzing Traces with Anonymous Synchronization

Our analysis method consists of three algorithms. Algorithm 1 initializes a time vector $\tau(e)$ for each event e. The rewinding algorithm (Algorithm 2) reduces the $\tau(e)$ time vectors, creating another time vector, $\tau'(e)$, for each event e. The effect of this rewinding algorithm is to remove edges from the represented partial order so that it is safe with respect to the inferred program. Unfortunately, the rewinding produces an overly conservative safe order relation. The third algorithm (Algorithm 3) increases the time vectors to restore some of the removed edges while maintaining a safe partial ordering.

The initialization process creates the history corresponding to the given causal trace. Unfortunately, this partial order is (in general) an unsafe order relation. The correspondence between Signals and Waits in the causal trace need not hold for other executions of the inferred program. The following is an obvious property of the time vectors generated by Algorithm 1.

Property 1: *The maximum value of any time vector component is the number of events performed by the task associated with that component.*

The following functions on sets of time vectors will be very useful when describing our algorithms.

Definition 8: *For any m time vectors $\tau_1, \ldots, \tau_m$ 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 ith component is* $\max(\tau_1[i], \ldots, \tau_m[i])$.

Conventionally, we define $\overline{\min}_0(\tau_1, \ldots, \tau_m)$ 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 $[2,3]$. We often call $\overline{\min}_k(\tau_1, \ldots, \tau_m)$ the k^{th} *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 9: *Given an event e performed by task T_i in a causal trace, let $\tau^{\#}(e)$ be the time vector 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 10: *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.*

The following algorithm (derived from [Mat88,Fid88]) computes time vectors for the history corresponding to a causal trace. This algorithm is also comprises the first phase of our analysis, converting the causal trace into a time vector representation of the corresponding event history.

Algorithm 1: Given a causal trace, each event e is assigned a time vector, $\tau(e)$, as follows[a]:

```
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 = τ(e');
    else
        set v_s = 0̄, the all zero vector;
    end if;
    set τ(e) = max(τ(e^p), τ#(e), v_s);
end for;
```

[a]Note that e^p must have already received its time vector, and if e is a Wait event then the unblocking signal event needs to have been assigned a time vector. Because any history corresponding to a trace is acyclic, there is always at least one event which can be assigned a time vector.

After completing Algorithm 1, $\tau(e) < \tau(e')$ iff $e \rightarrow e'$ in the history corresponding to the trace. Figure 2.1b shows the result of applying Algorithm 1 to the causal trace in Figure 2.1a.

Theorem 1: *If event $g \prec h$ in P_T, then for each execution of P where $h \rightarrow g$ there is a data race, $B(e, e')$ and $B(f, f')$, in P_T such that either $e \rightarrow h$ or $f \rightarrow h$ in the execution of P.*

An implication of the theorem is that if our algorithms report that there are no races in P_T then there are no races in P. The Appendix contains a proof of Theorem 1.

2.1 Virtual Time

Since a linearly ordered representation of time is not always adequate for reasoning about parallel programs, we use time vectors to represent a partial order on the events. In our trace analysis, each event is assigned a vector of timestamps. The ordered event pairs and unordered event pairs can be easily distinguished by comparing these time vectors.

The time vectors we compute in this paper are an extension of the time vectors of Fidge [Fid88] and Mattern [Mat88]. The easiest way to describe their system is in the context of message passing. There, each task T_i keeps its own count, $C_i[i]$, of the number of events it has performed. In addition, T_i keeps a count $C_i[j]$ on the number of events known to have been performed by each other task T_j. These other counts are generally underestimates, and are updated only when T_i synchronizes in some way with another task. Each time a message send event is performed by T_i, $C_i[i]$ is incremented and the event is timestamped with the vector C_i. This vector value C_i is also piggybacked onto the message. When T_i performs a receive event, it will obtaining a message with some timestamp C'. Task T_i sets, for each j, $C_i[j]$ to the maximum of $C_i[j]$ and $C'[j]$. It then increments $C_i[i]$ and timestamps the receive event with the new value of C_i.

When time vectors are assigned to events in this way, they represent a partial ordering of the events. An event e with timestamp τ precedes another event e' with timestamp τ' in the 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.

Our algorithms use these vector valued timestamps to represent event histories. The event histories are modified by updating the vector timestamps associated with events rather than explicitly adding (or deleting) arcs in the partial order.

Definition 7: *For any two time vectors τ_1, τ_2 in Z^n*

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

We say time vector τ_1 <u>is earlier than</u> *time vector τ_2 (or τ_2* is later than *τ_1) when $\tau_1 < \tau 2$. We say τ_1 and τ_2 are* <u>unordered</u> *when $\tau_1 \parallel \tau_2$.*

Definition 3: *Given a program P and two events, e and e', occurring in some execution of P, if e happens before e' in every execution of P in which both events occur then we write $e \prec_P e'$.*

When the program P is obvious from the context we will use $\prec$ instead of $\prec_P$.

Definition 4: *Given program P and event history H, if $e \rightarrow_H e'$ implies $e \prec e'$ then H is a safe partial order.*

The concept of safe histories is important, as these are the only orderings between events that a programmer can rely on (i.e. hold for every execution of the inferred program).

Definition 5: *If e and e' are two consecutive synchronization events performed by the same task in an event history, then $B(e, e')$ is the sequence of statements executed by the task between e and e'. We call $B(e, e')$ a (sequential) block.*

For data race detection, the only important characteristics of a block are the shared variables read and written by statements in the block. The list of variables read and written in a block can be recorded in the trace. Alternatively, the program itself can be analyzed to determine which variables can be accessed between pairs of synchronization statements. For the purposes of this paper, we assume that the read/write lists for blocks are available and concentrate on the ordering relationships between the blocks.

We say two blocks *conflict* if some shared variable is written in one of the blocks and either read or written in the other block.

Definition 6: *A race in P is a pair of conflicting blocks, $B(e, e')$ and $B(f, f')$, such that there exists a global trace of P where $f \rightarrow e'$ in the corresponding history and a (not necessarily different) global trace of P where $e \rightarrow f'$ in the corresponding history.*

This definition may be too general when shared variables are updated in certain ways while protected by lock-like structures. We deal with this issue in Section 3.4.

The algorithms in the remainder of this paper compute a safe partial order for a program inferred[4] by a local trace T. We use this safe partial order to report potential race conditions in the original program P.

In the inferred program P_T, each task executes sequentially the corresponding synchronization events from the local trace. Between synchronization events each task in P_T reads and writes the same variables that were read and written by P during the execution that created the local trace.

An important result is given by the following theorem. It says that if event g always happens before event h in P_T, but g does not always happen before h in P, then there is a race in P_T that happens before h.

[4]The inferred program is never actually computed. It is used to describe the limitations of the algorithm.

When H is clearly indicated by the surrounding context we will often drop the subscript and write $e \rightarrow e'$ instead of $e \rightarrow_H e'$.

For any trace there is always a corresponding event history that is the transitive closure of the order in which events appear in the trace. If this history is a local history then the trace is a *local trace*, with a separate log for each task. Likewise if this history is a global history then the trace is a *global trace*, and tracing may be a significant bottleneck. Conceivably, the trace could represent each event in its own file without timestamps or any other ordering information. In this case the corresponding history would contain no arcs. It is important to note that traces are an abstraction of the execution which produced them so that all event orderings need not be explicitly represented. Thus different executions can generate the same trace and, since the tracing mechanism could be nondeterministic, identical executions of the program might generate different traces.

We do, however, make two assumptions on traces and their corresponding histories. We assume that histories corresponding to traces are acyclic. Secondly, if there is a causal relationship between e and e' in the execution (for example, e represents the sending of some message and e' represents that message's reception) then no possible trace of that execution has a corresponding history where e' precedes e, contrary to the causal relationship.

Any trace analysis methodology is useful only if their is some minimum amount of information in the trace. In particular, our algorithms perform best when given a *causal trace*.

Definition 2: *A* causal trace *is a trace that indicates, for each blocking event (e.g. semaphore Wait) the particular enabling event (e.g. semaphore Signal) that caused the event to become unblocked.*

Every global trace can be interpreted as a causal trace but a causal trace need not be global. In particular a causal trace can be produced from strictly local information. In our semaphore model, this requires only that a log, indicating the order in which tasks successfully execute the Signals and Waits, be kept for each semaphore. This kind of information is commonly required in trace-and-replay systems such as Instant Replay [LMC87].

The history corresponding to a causal trace contains arcs from each enabling event to the blocking event it unblocks in addition to the arcs from each event to the next event performed by the same task.

There is a similarity between a causal trace and the notion of explicit synchronization introduced in the previous section. However, these two concepts have a subtle, but very important, difference. When the explicit synchronization is part of the events, the executions of the inferred program are constrained. The causal trace is used only for initialization, and does not restrict the set of executions that are considered. Thus our algorithms will report races that can occur with different Signal-Wait pairings. Although our algorithms can be run on local traces, the additional information provided by a causal trace allows a more accurate initialization, and hence more accurate results.

events contain anonymous synchronization (e.g. semaphores, locks, and signals). In this case, determining if two events occur in the same order in every execution is much more difficult[2].

The next section contains definitions and descriptions of our basic model involving counting semaphores. Our algorithms for this basic model are described and analyzed in Section 3. We have implemented a version of our algorithms for the Post/Wait style synchronization used in IBM's Parallel Fortran. Our implementation and the necessary modifications to our algorithms are described in Sections 4 and 5. In Section 6 we survey some related work. Finally, Section 7 contains conclusions and a brief summary of our results.

2 Description of the Model

In our basic model, programs synchronize using counting semaphores (initialized to zero). Two operations, **P** and **V**, are defined for each semaphore. In this paper, we use the more mnemonic Wait and Signal to represent the **P** and **V** operations respectively. Therefore, each synchronization event is a tuple containing: the operation completed (Wait or Signal), the affected semaphore, and the id of the task that performed the operation.

Many other kinds of synchronization operations can be simulated with counting semaphores. Consider, for example, the event "*init task t*" which creates a new task t and the event "*await task t*" which blocks the running task until task t has terminated. Given a trace containing these events, we can easily create a trace with equivalent synchronization properties that contains only semaphore events.

We distinguish between event traces, which are the information recorded during an execution of a program, and event histories, which are partial orders as defined below. An event history might correspond directly to a trace, or it could contain either more or less information than is directly accessible from a trace.

Definition 1: *An* event history *is an irreflexive partial order representing (some of) the order relations between events that occurred during some execution of a program. We use "$e \rightarrow_H e'$" to indicate that e precedes e' in history H.*
If a partial order contains only arcs from each event to the next event performed by the same task then we call it a local history.
If a partial order is actually a total order then we call it a global history[3]

[2]The difference between explicit and anonymous synchronization is somewhat in the eye of the beholder. Ada-like rendezvous becomes anonymous when the events "task t calls t'" and "task t' accepts some task" are used. Furthermore, semaphores can be made explicit by using the event "task t obtains the semaphore released by t'."

[3]We assume that the granularity of events recorded in traces of a program P is such that each execution of P can be represented by one or more total orderings of the events performed during the execution. This representation is often misleading in that events that were executed concurrently are portrayed as occurring in a particular order.

as many potential races as possible from a single trace. In particular the algorithms described in this paper will generate an ordering relation among program events that can be used to identify all races occurring in a potentially large set of related executions. Furthermore, if our algorithms report that there are no data races, then there will be no data races in any execution given the same input as that used to generate the trace.

We view the execution of a program as a collection of abstract *events* with various ordering properties. Although what comprises an event depends on the programming language, application, and desired level of detail, each event is a piece of program activity executed by a single task. Furthermore, each task executes events one at a time, with no overlap. We make the assumption that each synchronization statement executed by a task is represented by one (or more) events.

Many parallel systems (e.g. [IBM88]) provide facilities for recording important events during the execution of parallel programs. By limiting the debugger's activity, the *probe effect* should be reduced. The recorded information can be analyzed following the program's execution.

The following kind of query is helpful in the debugging process and critical for trace-based race detection.

> Given a program P running on a particular input and two program events, e_1 and e_2, is it true that in every execution which includes both e_1 and e_2, e_1 always occurs before e_2?

Unfortunately this question involves termination issues and thus is too hard to solve in polynomial time. We therefore compute a conservative approximate response. Our algorithms may respond No to the above query when in fact the correct response is Yes (resulting in possibly spurious race reports). However, if our algorithms respond Yes then either Yes is the correct response or else a race reporting tool based upon our algorithms will report a data race that occurs before either e_1 or e_2. This last condition can result in unreported data races but only when other races are reported and the reported races hide the unreported races ([AP87]). Theorem 1 assures us that if our algorithm reports that there are no races then there will be no races when the program is executed with the same input.

To accurately describe what it means when our algorithms respond Yes to the above query, it is helpful to consider the *inferred program* program P_T derived from a program P and a trace T of P. P_T is the same as P except that all conditional branches are replaced with unconditional branches resolved in the same direction as they were in the execution of P that generated T. Our algorithms compute a strictly conservative response to the query for this derived program. That is, for P_T, our algorithm may respond No incorrectly but will never falsely respond Yes. To always answer correctly, even for programs such as P_T is NP-Hard.

If the events explicitly identify the participating tasks (for example task t rendezvous with t', or task t forks into tasks t_1, t_2, and t_3) then there are relatively few ways of executing programs without conditional branches (such as P_T) and it is easy to determine those pairs of events that must occur in a particular order. The difficulties in answering the above query for programs such as P_T arise only when the

1 Introduction

Writing and debugging a parallel program is, in general, more difficult than writing and debugging a sequential program. A major reason for this difficulty is the need for explicit synchronization between the tasks in a parallel program. A program with errors in synchronization will often be *non-determinate*, i.e., generate different results even when started with exactly the same inputs. In a parallel program, nondeterminism often introduces unexpected program behavior, making the debugging process extremely difficult.

Unwanted non-determinate behavior of parallel programs often starts with a *data race*. One of the fundamental problems encountered when debugging a parallel program is locating the data races in the program. A data race exists between two statements[1], $S1$ and $S2$, if

1. the statements access the same memory location,
2. at least one of the accesses is a write, and
3. there exists an execution of the program where $S1$ happens before $S2$ and another execution, with the same input, where $S2$ happens before $S1$.

Notice that this definition includes both accesses that may occur "at the same time" and accesses that must occur serially but can occur in either order (such as accesses protected by a lock).

The current methods for determining potential races in parallel programs can be roughly divided into three groups: compile time analysis [Tay84,McD89,CKS90], run time (on-the-fly) analysis [DS90] and post-mortem trace based analysis [EP88, MC88,EGP89,NM89,HMW90a,HMW91]. No single approach has yet proven to be unquestionably superior. A brief sample of some advantages and disadvantes include the following. Compile time analysis has the advantage of being independent of the input data. Run time analysis has the advantage of not requiring the storage of massive amounts of trace data, because the information about the execution is processed and then discarded. Trace based analysis may be potentially less intrusive than run time analysis and can be used to detect races in alternative executions. Both compile time analysis and trace based analysis generally try to answer questions that are NP-complete. Therefore approximation methods are necessary for practical tools. The on-the-fly techniques are generally limited to reporting races that actually occurred in the execution being analyzed. They can never guarantee the absence of races in the program.

We have chosen to work on a trace based approach. For the type of programs we are interested in analyzing (programs containing semaphore style synchronization), the known compile time techniques can fail due to their time and space complexity. Because run time analysis only reports races that occurred in a single execution, alternative approaches must be investigated. We are primarily concerned with detecting

[1]For the purposes of this definition we assume that loops are unrolled and subroutines copied so that each statement is executed at most once during an execution of the program.

Determining Possible Event Orders by Analyzing Sequential Traces

D. P. Helmbold, C. E. McDowell, J-Z. Wang

91-36
September 25, 1991

Board of Studies in Computer and Information Sciences
University of California at Santa Cruz
Santa Cruz, CA 95064

ABSTRACT

One of the fundamental problems encountered when debugging a parallel program is determining the possible orders in which events could have occurred. Various problems, such as data races and intermittent deadlock, arise when there is insufficient synchronization between the tasks in a parallel program. A sequential trace of an execution can be misleading, as it implies additional event orderings, distorting the concurrent nature of the computation. This paper describes algorithms which generate those event orderings which can be relied on by the programmer from the trace of an execution.

By its very nature, the information in an execution trace pertains only to that execution of the program, and may not generalize to other executions. We tackle this difficulty in a systematic way: defining an "inferred program" based on the trace and original program, analyze this inferred program, and prove a relationship between the inferred program and the original.

The results of our algorithms can be used by other automated tools such as a data race detector or constraint checker. The basic algorithms described here have been implemented in a working trace analyzer for IBM Parallel Fortran. The trace analyzer graphically presents the discovered event orderings and reports various potential data races in the subject program.

Keywords: data race, time vector, program trace, parallel programming, debugging, distributed systems