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.

[Dij65] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9), September 1965.

[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.

[GPH*88] M. D. Guzzi, D. A. Padua, J. P. Hoeflinger, , and D. H. Lawrie. Cedar fortran and other vector and parallel fortran dialects. In *Proceedings Supercomputing '88*, pages 114–121, 1988.

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

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 21(7):558–565, July 1978.

[Lam86] Leslie Lamport. The mutual exclusion problem: part i–a theory of interprocess communication. *JACM*, 33(2):290–312, April 1986.

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

[Tay84] R. N. Taylor. *Debugging Real-Time Software in a Host-Target Environment.* Technical Report, U.C. Irvine Tech. Rep. 212, 1984.

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.

After adding the virtual edge from BW1 to CW1, CW1 becomes the second wait on S1. Using Algorithm 5: $\mathcal{S}$(BW1,CW1) is {(BW1,CW1), (BW1,CS1), (BW1,CS2), (BS1,CW1), (BS1,CS1), (BS1,CS2)}.

After adding the virtual edge from CS1 to BW1, BW1 becomes the second wait on S1. Again using Algorithm 5: $\mathcal{S}$(CW1,BW1) is {(BW1,CW1), (BS1,CW1), (BS2,CW1), (BW1,CS1), (BS1,CS1), (BS2,CS1)}.

$$\mathcal{S}(\text{BW1,CW1}) \cap \mathcal{S}(\text{CW1,BW1}) = \{(\text{BW1,CW1}),(\text{BW1,CS1}),(\text{BS1,CS1}),(\text{BS1,CW1})\}$$

Figure 5.1: Detect Critical Regions

The problem is made even more difficult when there is no clear correspondence between the blocking and enabling events in the trace.

This paper contains a series of algorithms for extracting useful information from sequential traces with anonymous synchronization. The first algorithm is very similar to the vector timestamp methods of 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. In addition to presenting our algorithms, we have also proved their correctness.

Although our algorithms find many of these "must-be-ordered" relationships, we have been unable to prove that they find all of them. We are investigating additional procedures which can increase the number of "must-be-ordered" relationships found. We would also like to distinguish all pairs of events that are concurrent in some consistent execution from pairs of events which can happen in either order, but not concurrently.

Some parallel programming environments view a parallel execution as a linear sequence

- 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 unordered sequential event pairs in critical region.

  1. First, assume that event e happened before e'. Thus e' is the $w+2$nd wait for S. Using Algorithm 4 with $k = w + 2$ to calculate time vectors for event e' and other events.

     Let $S(e, e') = \{(e_i, e_j) : (e_i, e_j) \in \text{Conc}, e_i \in E_i, e_j \in E_j$, and $\hat{\tau}(e_i)[i] \leq \hat{\tau}(e_j)[i]$ or $\hat{\tau}(e_j)[j] \leq \hat{\tau}(e_i)[j]\}$.

     Undo the timestamp updating.
  2. Similarly, assume that event e' happened before e. Thus e is the $w+2$nd wait for S. Using Algorithm 4 with $k = w + 2$ to calculate time vectors for event e and other events.

     Let $S(e', e) = \{(e_i, e_j) : (e_i, e_j) \in \text{Conc}, e_i \in E_i, e_j \in E_j$, and $\hat{\tau}(e_i)[i] \leq \hat{\tau}(e_j)[i]$ or $\hat{\tau}(e_j)[j] \leq \hat{\tau}(e_i)[j]\}$.

     Undo the timestamp updating.

     Let $\text{Seq}_t = S(e, e') \cap S(e', e)$. Notice that Seq_t maintains the set of unordered event pairs in the critical region. They are not concurrent in any executions, whenever e happened before e' or e' occurred before e.
  3. Let $\text{Seq} = \text{Seq} \cup \text{Seq}_t$.

     Let $\text{Conc} = \text{Conc} - \text{Seq}_t$.
- $s - w \leq 0$ means neither of them can precede. In this case, there is a deadlock.

End Algorithm 5.

Algorithm 5 generates two sets of event pairs. Conc contains those concurrent event pairs. Seq contains those unordered sequential event pairs. The remaining event pairs are ordered. Figure 5.1 shows the application of this algorithm to the trace from Figure 1.1.

6 Conclusion

One of the most difficult tasks in debugging parallel programs 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 parallel.

From equation 4.6 we know that at most k non-shadowed signals (excluding e_i) in $R(e)$ do not follow e_i (i.e. the $k+1$st smallest and later always follow e_i).
Therefore, in every execution, at least one of the $k+1$ non-shadowed signals preceding e follows (or is equal to) e_i.
By transitivity e_i happens before e in every execution consistent with, so, $e_i \prec e$.
$\square$

5 Adjusting the Timestamps to Determine Concurrency

Up to now, we have computed a partial order that reflects a safe order relation between events from the trace E. Given any two events $e_i \in E_i$ and $e_j \in E_j$, if $\hat{\tau}(e_i)[i] \leq \hat{\tau}(e_j)[i]$ or $\hat{\tau}(e_j)[j] \leq \hat{\tau}(e_i)[j]$ then the two events are ordered. Otherwise, e_i and e_j are two unordered events. The unordered events are not necessarily concurrent events. They may have 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.

When debugging parallel programs, we would like to distinguish those pairs of events that are concurrent in some consistent execution from pairs of events which can happen in either order, but not concurrently. Unfortunately, the concurrent relation cannot be determined immediately from the timestamps. We cannot necessarily say e_i can happen concurrently with event e_j even if we know $\hat{\tau}(e_i) \parallel \hat{\tau}(e_j)$. As an example, in Figure 4.2, even though $\hat{\tau}(BW1) \parallel \hat{\tau}(CW1)$, the two W1 events cannot occur at the same time. It is, in general, a hard problem to determine whether two unordered events can really happen concurrently.

Let $e, e' \in E$ be a pair of events. Event e may happen concurrently with e' only if $\hat{\tau}(e) \parallel \hat{\tau}(e')$. The following procedure can be used to detect critical regions, and to determine unordered sequential event pairs in critical regions. The algorithm will calculate two sets. The set Conc contains concurrent event pairs, while the set Seq contains unordered sequential event pairs. Initially, we assume that all unordered events are potential concurrent events. Once some critical regions have been detected, the algorithm will move those unordered sequential event pairs from Conc to Seq.

Algorithm 5: Initially let Conc $= \{\{e, e'\} : e, e' \in E$, and $\hat{\tau}(e) \parallel \hat{\tau}(e')\ \}$. Let Seq $= \phi$. 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 $G(e, e')$ be the set of wait events for semaphore S which precede either event e or e' (based on current timestamps $\hat{\tau}$).
Let $R(e, e') = \{e'' : e''$ is a signal event using S and e'' precedes e or $e'\} \cup \{e'' : e''$ is not shadowed with respect to either e or e', and e'' does not follow either e or $e'\ \}$.
Let $s = \mid R(e, e') \mid$ and $w = \mid G(e, e') \mid$.

Figure 4.2: Expanding the Safe Order Relation

Case1: Assume $\hat{\tau}(e)[i] = \hat{\tau}(e^p)[i]$ then

$$\begin{aligned} \hat{\tau}(e_i)[i] \leq \hat{\tau}(e^p)[i] \quad &\Rightarrow \quad e_i \prec e^p \qquad \text{by the induction hypothesis} \qquad (4.4) \\ &\Rightarrow \quad e_i \prec e \qquad \text{by transitivity} \qquad (4.5) \end{aligned}$$

Case2: $\hat{\tau}(e)[i] \neq \hat{\tau}(e^p)[i]$.
Event e is a wait on semaphore S. Let k be computed as specified in the algorithm, then

$$\hat{\tau}(e_i)[i] \leq \hat{\tau}(e)[i] \quad = \quad \min_{k+1}\{\hat{\tau}(e_s)[i] : e_s \in R(e)\} \qquad (4.6)$$

where non-shadowed signal set $R(e)$ is computed according to the algorithm, and $\min_{k+1}$ selects the $k+1$st smallest value from the set.
In every execution, at least $k+1$ signal events precede e since there are at least k waits on the same semaphore must happen before e.
In any arbitrary execution P, let k_s be the number of shadowed signals (with respect to e) that precede e.
By transitivity the corresponding k_s shadowing waits precede e, and at least $k + k_s$ waits on S precede wait event e.
Therefore, at least $k + k_s + 1$ signal events precede e in the execution and $k+1$ of them are non-shadowed signals.

Therefore, the signal event e'_s is shadowed by some wait event e'_w where $e_s \prec e'_w \prec e'_s$ with respect to e.
This forms a contradiction with the assumption that e'_s is shadowed by e_w.
$\square$

The Algorithm 4 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 consistent with the trace.

Algorithm 4: Initially $\hat{\tau}(e) = \tau'(e)$ for all events $e \in E$.

Repeat the following procedure until no more changes are possible.

Pick an event e. If e is a wait event using semaphore S, let

- $W(S)$ be the set of wait events on semaphore S,
- k be the number of wait events $e_w \in W(S)$ such that $e_w \neq e$ and if $e_w \in E_i$ then $\hat{\tau}(e_w)[i] \leq \hat{\tau}(e)[i]$, and
- $R(e) = \{\hat{e} : \hat{e}$ is a signal event on S, $e \not\prec \hat{e}$ as indicated by the $\hat{\tau}$ timestamps, and $\hat{e}$ is not shadowed with respect to $e\}$

and $v_s =$ the $k+1$st component-wise minimum of $\hat{\tau}(\hat{e})$ for $\hat{e} \in R(e)$.

If e is not a wait event, let v_s be the 0 vector.

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

End Algorithm 4.

Figure 4.2 shows the new $\hat{\tau}$ timestamps generated when Algorithm 4 is executed starting with Figure 3.1.

Theorem 5: *Algorithm 4 generates only safe order relations, i.e., for any two events $e_i \in E_i$ and $e \in E$:*

$$\hat{\tau}(e_i)[i] \leq \hat{\tau}(e)[i] \Rightarrow e_i \prec e$$

Proof: The proof is by induction on the number of updates. As a base case the theorem holds for the initial values of $\hat{\tau}$ from Theorem 4.

Assume the theorem holds before some update. Consider two events $e_i \in E_i$ and $e \in E$ where

$$\begin{array}{lcll} \hat{\tau}(e_i)[i] & > & \hat{\tau}(e)[i] & \text{before the update, and} \\ \hat{\tau}(e_i)[i] & \leq & \hat{\tau}(e)[i] & \text{after the update.} \end{array}$$

Because $\hat{\tau}(e_i)[i]$ never changes, $\hat{\tau}(e)[i]$ was updated.
We consider two cases.

Figure 4.1: Shadowed Signal Event

Since for each shadowed signal there is only one corresponding shadowing wait (by Definition 15), we have $\mid R_s^i(e) \mid \geq \mid R_w^i(e) \mid$.
We only need to show that $\mid R_s^i(e) \mid = \mid R_w^i(e) \mid$.
Assume to the contrary that $\mid R_s^i(e) \mid > \mid R_w^i(e) \mid$, which means that there are at least two signals e_s and e_s' in $R_s^i(e)$ shadowed by some $e_w \in R_w^i(e)$.
Assume $e_w \prec e_s \prec e_s'$. Let w_1 and s_1 be the number of waits and signals on S performed by T_i between e_w and e_s, w_2 and s_2 be the number of waits and signals on S performed by T_i between e_s and e_s'. This is shown in the following which represents the local subsequence of events performed by some task, where time moves from left to right.

$$\begin{array}{lccccc}
 & \mid\longleftarrow & s_1 \longrightarrow\mid & \mid\longleftarrow & s_2 \longrightarrow\mid & \\
\ldots e_w & & \ldots & e_s & \ldots & e_s' \ldots \\
 & \mid\longleftarrow & w_1 \longrightarrow\mid & \mid\longleftarrow & w_2 \longrightarrow\mid &
\end{array}$$

Therefore,

$$w_1 = s_1 \qquad \text{since } e_s \text{ is shadowed by } e_w \tag{4.1}$$

$$w_1 + w_2 = s_1 + s_2 + 1 \qquad \text{since } e_s' \text{ is shadowed by } e_w \tag{4.2}$$

Combining equations 4.1 and 4.2 gives us

$$w_2 = s_2 + 1 \tag{4.3}$$

However, equation 4.3 means that the subsequence between e_s and e_s' contains more waits on S than signals.

add additional safe orderings into the partial order using the fact that only some wait events in the trace can actually proceed immediately after each signal event. The partial order resulting from this final step will be represented by the time vectors $\hat{\tau}(e)$. Initially, $\hat{\tau}(e) = \tau'(e)$.

Definition 14: *Let $e \in E_i$ be a wait event and $e_s \in E_j$ be a signal event on the same semaphore S where $\hat{\tau}(e) \parallel \hat{\tau}(e_s)$. Let $E(e, e_s)$ be the subsequence of E_j containing every event e_j where $e_j \prec e_s$ and $\hat{\tau}(e_j) \parallel \hat{\tau}(e)$. If any suffix of $E(e, e_s)$ contains more wait events on S than signal events on S, then the signal event e_s is* shadowed *with respect to e.*

Definition 15: *Let $E'(e, e_s)$ be the shortest suffix of $E(e, e_s)$ which contains more wait events than signal events on S, and let e_w be the first event of $E'(e, e_s)$. We say e_s is shadowed by event e_w with respect to e.*

Lemma 2: *Given a wait event e and a signal event e_s on the same semaphore S, if e_s is shadowed by some event e_w with respect to e then*

- *Event e_w is a wait event on semaphore S,*
- *The event e_w, which shadows e_s with respect to e, is unique. We define e_w to be the shadowing wait event corresponding to e_s, and*
- *The subsequence between e_w and e_s (in the same task) contains as many signal events as wait events on semaphore S.*

Proof: The proof is straightforward from the definitions.
□

Definition 16: *For any wait event $e \in E$, let*
$R_s(e) = \{e_s : e_s$ *is shadowed with respect to* $e\}$, *and*
$R_w(e) = \{e_w : \exists e_s \in R_s(e)$ *s.t.* e_s *is shadowed by* e_w *with respect to* $e\}$.

In the example shown in Figure 4.1, the signal event CS1 is shadowed by CW1 with respect to two wait events performed by task B.

Lemma 3: *For any wait event $e \in E$, the correspondence between shadowed signal and shadowing wait is one to one, i.e.,*

$$| R_s(e) | = | R_w(e) | .$$

Proof: Let $e \in E$ be a wait event on semaphore S. From Definitions 14 and 15, we know that any pair of corresponding shadowed signal and shadowing wait belongs to the same task.
Therefore, it is enough to show that the correspondence between shadowed signal and shadowing wait is one to one within each task T_i where $1 \leq i \leq n$.
Let $R^i_s(e)$ and $R^i_w(e)$ be the sets of shadowed signal events and shadowing wait events performed by task T_i with respect to e.

Equation 3.7 implies for some c

$$\overline{\max}(\tau_P(e^p), \tau^{\#}(e), \tau_P(e_s))[c] < \overline{\max}(\tau'(e^p), \tau^{\#}(e), \overline{\min}(\tau'(e_{s_1}), \ldots \tau'(e_{s_m})))[c]. \tag{3.8}$$

Equation 3.5 and 3.8 imply

$$\begin{aligned} \tau_P(e_s)[c] &< \overline{\min}(\tau'(e_{s_1}), \ldots \tau'(e_{s_m}))[c] && (3.9)\\ \tau_P(e_s)[c] &< \tau'(e_s)[c] && (3.10)\\ \tau_P(e_s) &\not\geq \tau'(e_s) && (3.11) \end{aligned}$$

Again 3.11 contradicts the assumption that e is the first event in the topological order of the partial order P such that $\tau_P(e) \not\geq \tau'(e)$.

Therefore, there is no event e in any execution P such that $\tau_P(e) \not\geq \tau'(e)$.

□

Theorem 4: *After rewinding, we have a partial order that is a safe order relation, i.e.*

$$\tau'(e_i) < \tau'(e) \Rightarrow e_i \prec e.$$

Proof: Let i be the task performing e_i, so $\tau'(e_i)[i] = \tau_P(e_i)[i] = \tau^{\#}(e_i)[i]$.

$$\begin{aligned} \tau'(e_i)[i] &\leq \tau'(e)[i] && \text{from the hypothesis} && (3.12)\\ \tau'(e_i)[i] &\leq \tau_P(e)[i] && \text{by Lemma 1} && (3.13)\\ \tau_P(e_i)[i] &\leq \tau_P(e)[i] && \text{since } e_i \in E_i && (3.14)\\ e_i &\xrightarrow{P} e && \text{for all } P \text{ and} && (3.15)\\ e_i &\prec e && \text{from the definition of } \prec. && (3.16) \end{aligned}$$

□

The rewinding process is based on the fact that any signal event might enable any wait event on the same semaphore. We may have lost some safe order relations during rewinding. As an example, in Figure 3.1, time vector τ' says that two W2 events and the W1 event in task A may happen concurrently with all of the events in task B and C. However, it is obvious that the W1 in task A must happen after the two S1 events in task B and C, and the second W2 in task A has to wait until all of the events in B and C have occurred. The final step in the algorithm will find some of the order relations lost during the rewinding procedure.

4 Expanding the Safe Order Relation

The result of the rewind step is a partial order that is a safe order relation. It is an overly conservative safe order relation because it assumed that any wait could happen immediately after any signal for the same semaphore. We now undertake a process to

From the inductive hypothesis

$$\tau'(e) \leq \overline{\min}(\tau'(e_1^S), \ldots, \tau'(e_k^S)),$$

and from the algorithm

$$\overline{\min}(\tau'(e_1^S), \ldots, \tau'(e_k^S)) < \tau'(\hat{e}).$$

Therefore $\tau'(e) < \tau'(\hat{e})$.

□

After rewinding, we have a partial order that is a safe order relation. If event e_i has an earlier time vector than e, we can say e_i will happen before e in all executions that are consistent with the given trace. Before we prove this in theorem 4 we first present one lemma used in the proof.

Lemma 1: *For any execution P consistent with a trace E and for all events $e \in E$*

$$\tau_P(e) \geq \tau'(e)$$

Proof: Assume to the contrary that there is an execution P and some event e such that $\tau_P(e) \not\geq \tau'(e)$.
In any topological ordering (with respect to partial order P) of events in E, let e be the first event in the topological ordering such that $\tau_P(e) \not\geq \tau'(e)$.
We consider two cases.

Case1: If e is not a wait event then from Algorithms 1 and 3:

$$\begin{aligned} \tau'(e) &= \overline{\max}(\tau'(e^p), \tau^{\#}(e)) && (3.3)\\ \tau_P(e) &= \overline{\max}(\tau_P(e^p), \tau^{\#}(e)) && (3.4) \end{aligned}$$

Note that $\tau_P(e^p) \not\geq \tau'(e^p)$ by our choice of e. This contradicts the assumption that e is the first event in the topological order of the partial order P such that $\tau_P(e) \not\geq \tau'(e)$.

Case2: Event e is a wait event. From the choice of e we get

$$\begin{aligned} \tau_P(e^p) &\geq \tau'(e^p) && (3.5)\\ \tau_P(e) &\not\geq \tau'(e) && (3.6) \end{aligned}$$

Substituting the definitions of τ_P and τ' into 3.6 gives:

$$\overline{\max}(\tau_P(e^p), \tau^{\#}(e), \tau_P(e_s)) \not\geq \overline{\max}(\tau'(e^p), \tau^{\#}(e), \overline{\min}(\tau'(e_{s_1}), \ldots \tau'(e_{s_m}))) \quad (3.7)$$

where e_s is the corresponding signal event of e and thus appears before e in the topological order, and each e_{s_i} for $1 \leq i \leq m$ is one of the m signal events for the semaphore waited on by e_w.

Figure 3.1: Rewinding the Time Vectors

Proof: It is enough to show that $\tau'(e)[i] \leq \tau'(\hat{e})[i] \Rightarrow \tau'(e) < \tau'(\hat{e})$. The opposite direction follows directly from the definition of vector comparison.

The proof is by induction on the number of updates made by Algorithm 3. As the base case, from theorem 2, the theorem holds for the initial τ' values.
Assume the theorem holds before some update. Consider two arbitrary events, $e \in E_i$ and $\hat{e} \in E_j$, after updating a single time vector.
Since Algorithm 3 does not change $\tau'(e)[i]$ and never increases time vectors, updating $\tau'(e)$ can not make $\tau'(e)[i] \leq \tau'(\hat{e})[i] \Rightarrow \tau'(e) < \tau'(\hat{e})$ false. Therefore, we consider three cases when $\tau'(\hat{e})$ was updated.

Case1: $i = j$
If $e = \hat{e}^p$ then from the algorithm $\tau'(e) < \tau'(\hat{e})$.
Otherwise, $\tau'(e) < \tau'(\hat{e}^p)$ which implies $\tau'(e) < \tau'(\hat{e})$.

Case2: $i \neq j$ and $\tau'(\hat{e})[i] = \tau'(\hat{e}^p)[i]$.
This implies $\tau'(e)[i] \leq \tau'(\hat{e}^p)[i]$. Since neither $\tau'(\hat{e}^p)$ nor $\tau'(e)$ changed, by the induction hypothesis $\tau'(e) \leq \tau'(\hat{e}^p)$, and the algorithm ensures that $\tau'(\hat{e}^p) < \tau'(\hat{e})$. Therefore $\tau'(e) < \tau'(\hat{e})$.

Case3: $i \neq j$ and $\tau'(\hat{e})[i] \neq \tau'(\hat{e}^p)[i]$. This implies that $\hat{e}$ is a wait event for some semaphore S.
Let $e^S_1 \ldots e^S_k$ be the signal events for the semaphore S. From the algorithm definition and the assumption we know

$$\tau'(e)[i] \leq \tau'(\hat{e})[i] = \overline{\min}(\tau'(e^S_1), \ldots, \tau'(e^S_k))[i].$$

3 Rewinding the Time Vectors

The result of the initialize step in the previous section is an unsafe order relation. It is unsafe because we assumed that the kth signal event for a particular semaphore was the one allowing the kth wait event to precede. The next step is to rewind the time vectors to account for the fact that any signal event might be the one that allowed any wait event on the same semaphore to complete. We use $\tau'(e)$ to represent the new time vector assigned to event e during and after the rewinding process. Initially τ' is the same as τ.

Suppose e is a wait event, and e_1 and e_2 are two signal events, either of which could have caused e to complete. In this case, we only know that either e_1 or e_2 must have happened before e. The trace might be in any of the forms:

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

However, we can conclude that the common ancestors of e_1 and e_2 must occur before e. Therefore if $e_a \prec e_1$ and $e_a \prec e_2$ then $e_a \prec e$. The rewind step defined below uses this fact to obtain a safe order relation.

Algorithm 3: Initially, $\forall e \in E, \tau'(e) = \tau(e)$.
Repeat the following procedure until no further changes are possible.

For all event $e \in E$, let

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

where if e is wait event on semaphore S:

$$\begin{array}{rcll} v_s & = & \overline{\min}(\tau'(e_1^s), \ldots, \tau'(e_k^s)) & (3.1) \\ & & \text{where } e_1^s \ldots e_k^s \text{ are all the signal events for the semaphore S.} & (3.2) \end{array}$$

otherwise v_s is the 0 vector.

End Algorithm 3.

Observe that the only difference between Algorithm 3 and Algorithm 2 (used to compute τ) is that for wait events in Algorithm 3, v_s is the minimum of a set of time vectors, which includes the time vector used for v_s in computing τ. Therefore the values of τ' will only get smaller as Algorithm 3 executes.

Theorem 3: *For any two distinct events $e \in E_i, \hat{e} \in E_j$,*

$$\tau'(e)[i] \leq \tau'(\hat{e})[i] \Longleftrightarrow \tau'(e) < \tau'(\hat{e})$$

2. If e_l is the corresponding signal event and $e_i \rightarrow e_l$ then $\tau(e_i)[i] \leq \tau(e_l)[i]$ and $e_l \rightarrow e$ $\Rightarrow \tau(e_l)[i] \leq \tau(e)[i]$ and the result follows.
3. If e_l is the corresponding signal event and $e_i \not\rightarrow e_l$ then

$$\begin{array}{rcll} e_i & \rightarrow & e^p & \text{Property 8} \quad (2.8) \\ \tau(e_i)[i] & \leq & \tau(e^p)[i] & \text{from the inductive hypothesis} \quad (2.9) \\ \tau(e^p)[i] & \leq & \tau(e)[i] & \text{from the definition of } \tau \quad (2.10) \end{array}$$

and the result follows.

□

Theorem 2: *For any two distinct events $e_i \in E_i, e \in E$,*

$$\tau(e_i)[i] \leq \tau(e)[i] \Longleftrightarrow \tau(e_i) < \tau(e)$$

Proof: The $\Leftarrow$ direction is trivial. For the $\Rightarrow$ direction, assume to the contrary that there are two events, $e_i \in E_i$ and $e \in E$ where $\tau(e_i)[i] \leq \tau(e)[i]$ but $\tau(e_i) \not< \tau(e)$. Thus there is some vector component c such that

$$\tau(e_i)[c] > \tau(e)[c]. \quad (2.11)$$

Let e_c be the event occurring in E_c with sequence number $\tau(e_i)[c]$ then

$$\begin{array}{rcll} \tau(e_c)[c] & = & \tau(e_i)[c]. & \quad (2.12) \\ e_c & \rightarrow & e_i & \text{from Theorem 1 and 2.12} \quad (2.13) \\ e_i & \rightarrow & e & \text{from the hypothesis and theorem 1} \quad (2.14) \\ e_c & \rightarrow & e & \text{from transitivity of } \rightarrow \quad (2.15) \\ \tau(e_c)[c] & \leq & \tau(e)[c] & \text{from 2.15 and Theorem 1} \quad (2.16) \end{array}$$

Combining 2.12 and 2.16 forms a contradiction with 2.11.

□

Corollary 1: *For any two distinct events $e \in E_i$, $\hat{e} \in E_j$, $i \neq j$:*

$$\tau(e)[i] > \tau(\hat{e})[i] \textit{ and } \tau(\hat{e})[j] > \tau(e)[j] \Longrightarrow e \parallel \hat{e}$$

The initialization process creates a partial ordering of the events in the trace. This partial ordering corresponds to an execution which is strongly consistent with the trace. It describes the *happened before* relation for the *canonical execution.*

Unfortunately, this partial order only gives the happened before relationships between events for the canonical execution, i.e. it is an unsafe order relation. The kth signal event in one execution might not necessarily be the kth signal in some other execution. Therefore, event e may not happen before $\hat{e}$ in some other execution even if it did in this execution. Even when $\tau(e) < \tau(\hat{e})$ we cannot say e *must happen before* $\hat{e}$.

Proof: From Properties 3 and 7 we know that if either side holds then e_i appears before e in the trace. Therefore, it suffices to prove that whenever the algorithm assigns a time vector to some event e, and e_i is any event appearing earlier in the trace (and thus already assigned a time vector by the algorithm) the two conditions are equivalent. We prove this by induction on the position of e in the trace.

After the first event is assigned a time vector, the theorem trivially holds as no distinct pairs of events have been assigned time vectors. We now show that the time vector assigned to the next event, e, satisfies the theorem assuming that the time vectors assigned to all events appearing before e in the trace satisfy the theorem.

We first show that assuming $\tau(e_i)[i] \leq \tau(e)[i]$ then $e_i \rightarrow e$. If $e \in E_i$, so that the two events are in the same task, T_i, the implication follows because the selected vector component is the event count for task T_i. Otherwise the events occur in different tasks and

$$\tau(e)[i] = \tau(\hat{e})[i]$$

where

$$\hat{e} \text{ is either} \begin{cases} e^p & \text{or possibly} \\ e_j & \text{if } e \text{ is a wait event and } e_j \text{ is the corresponding signal event.} \end{cases}$$

In either case $\hat{e}$ has previously been assigned a time vector and

$$\begin{array}{rcll} \tau(e)[i] & = & \tau(\hat{e})[i] & \text{by the definition of } \tau \quad (2.1) \\ \tau(e_i)[i] & \leq & \tau(\hat{e})[i] & \text{from the assumption} \quad (2.2) \\ \hat{e} & \rightarrow & e & \text{from the definition of } \hat{e} \quad (2.3) \end{array}$$

Either $e_i = \hat{e}$ and the theorem is proven or by the induction hypothesis $e_i \rightarrow \hat{e}$, and by transitivity $e_i \rightarrow e$.
To prove that $e_i \rightarrow e \Rightarrow \tau(e_i)[i] \leq \tau(e)[i]$ we consider three cases.

Case1: If $e \in E_i$, so that the two events are in the same task, the result follows from Properties 3 and 4.

Case2: If e is not a wait event then

$$\begin{array}{rcll} \tau(e)[i] & = & \tau(e^p)[i] & \quad (2.4) \\ e_i & \rightarrow & e^p & \text{Property 8} \quad (2.5) \\ \tau(e_i)[i] & \leq & \tau(e^p)[i] & \text{from the hypothesis} \quad (2.6) \\ \tau(e_i)[i] & \leq & \tau(e)[i]. & \quad (2.7) \end{array}$$

Case3: If e is a wait event then we have three subcases:

1. If e_i is the corresponding signal event then the result trivially holds.

Algorithm 2: To compute initial time vectors, $\tau(e_i)$, from a trace E use algorithm 1 with the following modifications.

- The kth wait event on semaphore S (in trace order) corresponds to the kth signal event on S.
- The events are assigned time vectors in the order they appear in the trace.

End Algorithm 2.

For the given trace, Figure 1.1(a) shows the result of the initialization procedure.
The time vectors computed for the canonical execution have the following properties:

Property 4: *If e and $\hat{e}$ are two events in the same task T_i and e occurred before $\hat{e}$ in the trace, then $e \rightarrow \hat{e}$ and $\tau(e) < \tau(\hat{e})$.*

Property 5: *If e and $\hat{e}$ are the corresponding signal/wait pair (the kth signal and the kth wait on the same semaphore S in the trace), then $e \rightarrow \hat{e}$ and $\tau(e) < \tau(\hat{e})$.*

Property 6: *At any point in the trace, the maximum value of any time vector component is the number of events performed by the corresponding task up to that point.*

Property 7: *If $e \in E_i$ and $\tau(e)[i] \leq \tau(\hat{e})[i]$ then either $e = \hat{e}$ or e appears before $\hat{e}$ in the trace.*

Because an event is only constrained to follow its predecessor in the same task, and in the case of wait events, the corresponding signal, the following property holds.

Property 8: *If $e \rightarrow \hat{e}$ then one of the following is true:*

1. *$e = \hat{e}^p$,*
2. *$e = \hat{e}_s$ where $\hat{e}$ is a wait event and $\hat{e}_s$ is the corresponding signal event,*
3. *$e \rightarrow \hat{e}^p$ or*
4. *$e \rightarrow \hat{e}_s$ where $\hat{e}$ is a wait event and $\hat{e}_s$ is the corresponding signal event.*

Given the correspondence between signal and wait events for execution P, events can be assigned time vectors by using Algorithm 1. Mattern [Mat88] has shown that the time vector τ_P correctly represents the partial order relation $\overset{P}{\rightarrow}$, i.e., for any pair of distinct events $e_i \in E_i$ and $e \in E$,

$$\tau_P(e_i)[i] \leq \tau_P(e)[i] \Longleftrightarrow e_i \overset{P}{\rightarrow} e.$$

For completeness, we now prove that the initial time vectors, τ, correctly represent the happened before relation for the canonical execution.

Theorem 1: *For any pair of distinct events $e_i \in E_i$ and $e \in E$,*

$$\tau(e_i)[i] \leq \tau(e)[i] \Longleftrightarrow e_i \rightarrow e$$

1.3 An Overview of the New Algorithms

In the following sections we will introduce a series of algorithms to calculate different time vectors for trace events. By comparing their final time vectors, we can distinguish many *ordered* events from the *unordered*, potentially concurrent, events. Our goal is a set of time vectors where if event e_1 has an earlier time vector than e_2, then e_1 will happen before e_2 in *all* executions that are consistent with the given trace[3].

The three phases of the algorithm are "initialize", "rewind", and "expand". The initialization uses Algorithm 1. The resulting partial order is similar to that computed by the algorithm of [Fid88]. This partial order is shown to be equivalent to the "happened before", $\overset{P}{\rightarrow}$, relation for a canonical execution P. Note that the canonical execution is in general not the same execution which generated the trace. The result of the rewinding phase is a partial order that is a subrelation of the $\prec$ relation. Unfortunately this safe order relation is overly conservative, in that there may be many "must happen before" relations that it does not include. The third and final phase results in a safe partial order that is closer to the "must happen before" relation.

2 Initializing the Vectors

Before giving the algorithm for computing the initial time vectors, we define a canonical execution that will be used to verify the "correctness" of the time vectors.

Definition 13: *Given a trace E with the total ordering of events, $<_E$, the partial order $\overset{P}{\rightarrow}$ corresponding to the* canonical *execution P is constructed by selecting and taking the transitive closure of the following subrelation of $<_E$.*

- *If e_i and e_j are two events from the same task and $e_i <_E e_j$ then $e_i \overset{P}{\rightarrow} e_j$.*
- *If e_i and e_j are the kth signal and wait events respectively on the same semaphore, then $e_i \overset{P}{\rightarrow} e_j$.*

In the remainder of the paper we will use $\rightarrow$ to mean $\overset{P}{\rightarrow}$ where P is the canonical execution defined above.

Property 3: *If $e \rightarrow \hat{e}$ then e appears before $\hat{e}$ in the trace.*

[3]Given a specific input and a trace, there are in general executions which are not consistent with that trace, however, any such execution will contain a race if and only if a race occurred in the execution that generated the trace [AP87].

length n, where n is the total number of tasks.[2] Each task T_i has its own vector component $C_i[i]$ which guarantees a strict temporal ordering of events occurring in that task. A local event counter which is incremented each time an event occurs in the task can be used as the local clock.

Before presenting the algorithms for computing time vectors from a trace, we need to define some notation.

Definition 9: *For an event $e \in E_i$, e^p is the previous event performed by the same task T_i if such an event exists.*

Definition 10: *For an event $e \in E_i$, $\tau^{\#}(e)$ is the time vector containing the local event count for e in the ith position and zeros elsewhere.*

Definition 11: *For any two time vectors u, v in Z^n*

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

Definition 12: *For any k time vectors $v_1, \ldots, v_k$ of Z^n*

- $\overline{\min}(v_1, \ldots, v_k)$ *is a vector of Z^n whose ith component is* $\min(v_1[i], \ldots, v_k[i])$, *and*
- $\overline{\max}(v_1, \ldots, v_k)$ *is a vector of Z^n whose ith component is* $\max(v_1[i], \ldots, v_k[i])$.

The following algorithm (derived from [Mat88, Fid88]) computes time vectors for the events in an execution. This algorithm requires the correspondence between signal and wait events. The time vectors produced reflect the execution's partial order.

Algorithm 1: Given the correspondence between signal and wait events for execution P, events are assigned time vectors, $\tau_P(e_i)$, in topological order.

$$\tau_P(e_i) \quad = \quad \overline{\max}(v_t, v_s, \tau^{\#}(e_i))$$

where

$$v_t \quad = \quad \begin{cases} \tau_P(e_i^p) & \text{if } e_i \text{ has a predecessor} \\ \text{the 0 vector} & \text{otherwise} \end{cases}$$

$$v_s \quad = \quad \begin{cases} \tau_P(\hat{e}) & \text{if } e_i \text{ is a wait event and} \\ & \hat{e} \text{ is the corresponding signal event} \\ \text{the 0 vector} & \text{if } e_i \text{ is not a wait event} \end{cases}$$

End Algorithm 1.

[2]We use an integer valued clock in our discussion although a real number valued clock can also be used.

Definition 4: *An execution is* strongly consistent *with a trace if it is consistent and the total order specified by the trace is an extension of the partial order specified by the execution.*

For example, consider the trace {AS1, CW1, CS1, CS2, BW1, BS1, BS2, AW2, AW2, AW1}. Event AS1 means task A performs a signal(S_1), AW1 means task A performs wait(S_1) etc. Figure 1.1 shows the four executions which are consistent with this trace. In addition, the executions (a) and (b) are strongly consistent with the trace.

Definition 5: *Consider the correspondence between signal and wait events in execution P and two distinct events e, e'. If $e \overset{P}{\not\to} e'$ and $e' \overset{P}{\not\to} e$ then events e and e' are concurrent, and thus can happen at the same time, in the execution.*

Definition 6: *The symbol "$\|$" is used to represent the concurrent relationship between events. Two events e and e' are concurrent, i.e. $e \| e'$, if they can happen at the same time in some execution which is consistent with the trace.*

Definition 7: *The symbol "$\prec$" is used to represent the* must happen before *relationship between events. Given two events e and e', if $e \prec e'$, then event e will happen before e' in all executions that are consistent with the given trace. Events e and e' are* ordered *if $e \prec e'$ or $e' \prec e$, otherwise, they are* unordered.

Concurrent events are always unordered, but unordered events need not be concurrent. For example, see events BW1 and CW1 in Figure 1.1.

Notice that, in general, $e \prec e'$ is different from the relation $e \xrightarrow{P} e'$ for any choice of P. The former relation tells us that e *must happen before* e' in all executions consistent with the trace being analyzed, while the later says that e happened before e' in the execution represented by the partial order P. If $e \prec e'$ then $e \xrightarrow{P} e'$ for all consistent executions P. But the converse condition does not hold.

In Figure 1.1, CS1 $\xrightarrow{P}$ BW1 if P is the execution (a). However, if P is the execution (c), BS1 $\xrightarrow{P}$ CW1, and BW1 $\xrightarrow{P}$ CS1 by transitivity. Event AS1 happens before BW1 and CW1 in all executions consistent with the trace, therefore AS1 $\prec$ BW1 and AS1 $\prec$ CW1. There is no order relation between event CS2 and BW1 in execution (a). Therefore, they can happen concurrently, i.e., CS2 $\|$ BW1.

Definition 8: *A partial ordering R on the events is a* safe *order relation if $e_i \; R \; e_j \Rightarrow e_i \prec e_j$. If R is not safe, then R is* unsafe.

1.2 Virtual Time

The concept of virtual time for distributed systems was introduced by Lamport in 1978 [Lam78]. The time vectors we compute in this paper are an extension of the time vectors of Fidge [Fid88] and Mattern [Mat88]. There, each task T_i has a clock C_i which is a vector of

Trace = {AS1, CW1, CS1, CS2, BW1, BS1, BS2, AW2, AW2, AW1}

Figure 1.1: Trace, Executions, and Time Vectors

- and a (positive integer) sequence number equal to one plus the number of previous operations performed by the task.

In order to perform the final race analysis, it must be possible to determine from a trace what shared objects are referenced between any two synchronization events. This can be done by additionally associating with each event the source line number of the statement generating the event. From this the path between two adjacent events can be determined and the variables referenced along the path can be computed [McD89].

Many other kinds of synchronization operations can be simulated by using 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 create an equivalent a trace containing only semaphore events.

In each execution every wait event has a corresponding signal event. We use this correspondence to define a partial order representing that execution.

Definition 1: *An* execution *of a parallel program is a partial ordering of the events performed. This partial order is the transitive closure of edges from each event to the next event performed by the same task and edges to each wait event from the corresponding signal event.*

The relation defined by the partial order P representing an execution is called the *happened before* relation and is denoted with the symbol $\xrightarrow{P}$. Our definition of "happened before" is consistent with that of Lamport[Lam78].

Definition 2: *A trace of an execution is an interleaving of the local sequences of events E_i for $1 \leq i \leq n$ where for every prefix of the trace and every semaphore S, the prefix contains at least as many signal(S) events as wait(S) events.*

Every trace must satisfy the following properties:

Property 1: *No two events in the trace have both the same task id and the same sequence number.*

Property 2: *If there is an event with task id t and sequence number k, then for every $1 \leq i < k$, there is an event with task id t and sequence number i appearing earlier in the trace.*

A single execution usually has many possible traces. Similarly, a single trace could have been generated by any one of a number of executions. (Figures 1.1(a) and 1.1(b) show two different executions for the same trace).

Definition 3: *An execution is* consistent *with a trace if the local sequences of trace events E_i for each task $1 \leq i \leq n$ is the same as in the execution.*

occur" execution order. Our algorithms appear to be more efficient and may find more guaranteed order relations.

Netzer and Miller [NM89] present a formal model of a program execution based on Lamport's model of concurrent systems [Lam86]. 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, and a *feasible data race*, which is a data race that could have been exhibited due to timing variations. They show how to characterize each detected data race as either being feasible, or as belonging to a set of data races such that at least one data race in the set is feasible. They rely on the trace for their ordering information. As an example, when two tasks try to enter some critical regions surrounded by some binary semaphore S, their algorithm will say that these two tasks are ordered when accessing these regions. Under their definitions there is neither an actual nor feasible data race even if two tasks write to some shared variable in this case. We view the ordering relationships in the trace with suspicion, and wish to generate race reports in this situation.

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 present 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, including IBM Parallel Fortran [IBM88], and Cedar Fortran [GPH*88].

1.1 Description of the Model

We view a parallel program as a finite set of *tasks* $T_1, \ldots, T_n$ where n is the number of tasks in the system. These tasks perform synchronization and computation operations, including computation on shared data[1]. In an execution, each task T_i is a sequential entity characterized by a local sequence E_i of events. Different tasks may perform operations concurrently. We assume, for convenience, that each task has a unique identifier.

In our model, programs synchronize using only counting semaphores which are assumed to be initialized to zero. Therefore, each event is a tuple containing:

- the operation completed (wait or signal),
- the semaphore affected,
- the id of the task that performed the operation,

[1]Although operations on shared data can be used for synchronization [Dij65], we only consider explicit synchronization operations as capable of generating synchronization events.

1 Introduction

One of the fundamental problems encountered when debugging a parallel program is determining the race conditions in the program. A race condition may exist when two or more parallel tasks access shared data in an unspecified order and at least one of the accesses is a write access. Notice that races include both accesses that may occur "at the same time" and accesses that must occur sequentially but the order is unspecified (e.g. accesses protected by a lock). One approach to determining potential races is based on computing all of the reachable concurrent states of the program [McD89, Tay84]. The major disadvantage of this approach is that the number of concurrent states may become prohibitively large. Another approach to determining potential races is based on analyzing a trace from an execution of the program [EP88, EGP89, NM89]. This approach has the disadvantage that a trace must be recorded, and is limited to determining races that can occur given the input data used. Even for the given data, it may not be possible to determine all races [AP87]. Nevertheless, this later approach can provide important information to help in debugging parallel programs and is the subject of this paper.

A *trace* specifies a total ordering of the events performed by the program. For our purposes, the trace reflects only one of the orders in which the events could have occurred. A more restrictive definition that is difficult to achieve in practice would be for a trace to specify the exact order in which the events did occur. Since traces are only approximations of executions, there are usually several executions that are consistent with a given trace. What we want to compute is the orderings between pairs of events that *must occur* in all executions which are consistent with the trace. In general this will be a partial order. If the partial order contains all orderings that must occur, then a pair of events not ordered by this "*must occur*" partial ordering can potentially execute in either order.

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 edge is then added from the closest common ancestor of these `Post` events to the `Wait` event node. The idea of the algorithm is very simple, but it may be computationally complex. Also some of the guaranteed order relations may be missed by their algorithm. Rather than repeatedly computing the common ancestor information, we use time vectors to calculate the guaranteed "must

Analyzing Traces with Anonymous Synchronization

David P. Helmbold
Charles E. McDowell
Jian-Zhong Wang

UCSC-CRL-89-42
December, 1989

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

ABSTRACT

In a parallel system, events can occur concurrently. However, programmers are often forced to rely on misleading sequential traces for information about their program's behavior. We present a series of algorithms which extract ordering information from a sequential trace with anonymous semaphore-style synchronization.

We view a program execution as a partial ordering of events, and define which executions are consistent with a given trace. Although it is generally not possible to determine which of the consistent executions occurred, we define the notion of "safe orderings" which are guaranteed to occur in every execution which is consistent with the trace.

The main results of the paper are algorithms which determine many of the "safe orderings". The first algorithm starts from a sequential trace and creates a partially ordered canonical execution. The second algorithm strips away the ordering relationships particular to the canonical execution, so that the resulting partial order is safe. The third algorithm increases the amount of ordering information while maintaining a safe partial order. All three algorithms are accompanied by proofs of correctness.

keywords: virtual time, program tracing, parallel processing, debugging

This work was supported by IBM under agreement SL 88096.