Timestamps for Programs Using Messages and Shared Variables

Alessio Bechini

Dipartimento di Ingegneria dell'Informazione
Facoltà di Ingegneria-Università di Pisa
via Diotisalvi, 2 56100 Pisa, Italy
alessio@pical3.iet.unipi.it

Kuo-Chung Tai

Department of Computer Science
North Carolina State University
Raleigh, North Carolina 27695-7534, USA
kct@csc.ncsu.edu

Abstract

Algorithms for vector timestamps have been developed to determine the "happened before" relations between events of an execution of a message-passing program. Many message-passing programs contain variables shared by multiple processes (including threads). Such programs need to have vector timestamps for send, receive, read and write events. In this paper, we define two "happened before" relations, called strong happened-before (SHB) and weak happened-before (WHB), between events of an execution involving send, receive, read and write statements. We then present two timestamp assignment algorithms, one for SHB and the other for WHB, and show how to use such timestamps to determine the SHB or WHB relation between any two events of an execution involving send, receive, read and write statements. For a program containing n processes, the size of a vector timestamp for SHB or WHB is n, regardless of the number of shared variables in the program. Finally, we show how to apply WHB timestamps to perform race analysis for programs using messages and shared variables.

1. Introduction

Traditionally, processes in a distributed program communicate with each other through message-passing, and processes in a parallel program through shared variables. Due to the use of threads and/or shared distributed memory, many distributed programs contain variables shared by multiple processes (including threads). On the other hand, many parallel programs use message passing for communication and synchronization. Thus, programs using both messages and shared variables are becoming popular. One example is that a program runs on multiple sites, with processes in each site communicating with each other through shared variables. A more general example is that processes in a program communicate with each other through both messages and shared variables, regardless of the locations of processes and shared variables.

Lamport defined the "happened before" relation for events of an execution involving send and receive statements [5]. Vector timestamps have been used to determine the "happened before" relations between events of an execution involving send and receive statements [3, 6, 9]. For programs using both messages and shared variables, if all accesses to shared variables are synchronized by using messages, then the "happened before" relations between read and write accesses for shared variables can be determined by the "happened before" relations between their corresponding send and receive events.

In this paper, we consider concurrent programs that use both messages and shared variables and do not necessarily synchronize all accesses to shared variables by using messages. We show how to extend the classical "happened before" relation to define two "happened before" relations for events of an execution. These two "happened before" relations have different applications for analysis, testing and debugging of concurrent programs. We also show how to assign vector timestamps to send, receive, read and write events and how to use these timestamps to determine event ordering.

The paper is organized as follows. Section 2 reviews previous work on timestamps. Section 3 defines

This work was performed when the first author was a visitor at NCSU, supported by a fellowship from the University of Pisa; the work was supported in part by MURST, Italy. This work was supported in part by NSF grant CCR-9320992

0-8186-8292-2/98 $10.00 © 1998 IEEE

two "happened before" relations for send, receive, read and write events. Section 4 shows the assignment of vector timestamps for send, receive, read and write events and the use of vector timestamps to determine event ordering. Section 5 describes how to apply WHB timestamps to perform race analysis for programs using messages and shared variables. Section 6 concludes this paper.

3. Previous works on timestamps

Lamport proposed the assignment of *integer timestamps* to events of an execution involving send and receive statements [5]. Integer timestamps can be used to produce totally ordered sequences of events of an execution involving send and receive statements such that these sequences do not violate the "happened before" relations among events of the execution (i.e. if event e "happened before" event f, then e appears before f in any of these totally ordered sequences). However, integer timestamps cannot be used to determine the "happened before" relation between two events of the same execution. To solve this problem requires the use of *vector timestamps*, each consisting of n values, where n is the number of processes involved in an execution. How to assign vector timestamps for events of an execution involving asynchronous communication (i.e. non-blocking send and blocking receive) is shown in [6]. The assignment of vector timestamps for events of an execution involving asynchronous and/or synchronous communication is described in [3]. A technique for improving the implementation of vector timestamps for message-passing programs was proposed in [10]. Netzer considered optimal tracing and replay of parallel programs that contain accesses to shared variables, but do not contain messages [7]. He presented an algorithm that uses vector timestamps for read and write events on shared variables in a parallel program.

During an execution of a parallel program, the number of parallel threads is usually not a constant. A number of timestamp techniques for a parallel program avoid the use of vector timestamps with their size being the total number of parallel threads in the program. Dinning and Schonberg considered parallel programs that use doall-endall statements for parallelism and some coordination statements for synchronizing accesses to shared variables [2]. A block of a parallel program is defined as an instruction sequence, executed by a single thread, that does not include doall, endall, or any coordination statements. A technique, called *task recycling*, assigns a vector timestamp to a block, where the size of this vector timestamp is the maximum number of parallel threads in the outermost doall-endall statement that contains the block. Audenaert considered parallel programs that use fork and join statements for parallelism and send and receive statements for synchronizing accesses to shared variables [1]. He described a technique that assigns a *clock tree*, which is a tree of vector timestamps, to a fork, join, send or receive event. The average size of a clock tree is much smaller than the size of a vector timestamp based on task recycling.

The above two timestamp techniques for parallel programs assume that all accesses to shared variables are synchronized by message-passing and other statements. Therefore, they do not consider read and write events in the derivation of timestamps. Such timestamp techniques can be applied to detect the existence of race conditions for shared variables in a parallel program satisfying the above-mentioned assumption. However, such timestamp techniques cannot be applied to solve the problem addressed in this paper. In this paper we combine Fidge's assignment of vector timestamps to send and receive events and Netzer's assignment of timestamps to read and write events.

3. "Happened before" relations for send, receive, read and write events

In this section, we define two "happened before" relations, called strong happened-before (SHB) and weak happened-before (WHB), for send, receive, read and write events. In the following discussion, we consider a concurrent program containing processes that communicate with each other by using messages and shared variables. A send statement is either blocking or non-blocking, and a receive statement is blocking. Asynchronous message-passing refers to the passing of a message from a non-blocking send to a receive, and synchronous message-passing refers to the passing of a message from a blocking send to a receive. A read or write operation is assumed to be atomic. If shared distributed memory is used, strict or sequential consistency for the shared memory is assumed [14]. Thus, the sequence of read and write events on a shared variable during an execution is a totally ordered sequence. For two read/write events e and f on a shared variable V, $e \xrightarrow{d(V)} f$ denotes that e occurs before f on V.

Fig. 1 shows a graphical representation of a sequence of read and write events on shared variable V by processes P_1, P_2, P_3 and P_4. In fig. 1, each process or shared variable is denoted by a vertical line. A read event on V by a process is denoted by a horizontal line from the vertical line for V to the vertical line for the process. A write event on V by a process is denoted by a horizontal line from the vertical line for the process to the vertical line for V.

3.1 Strong happened-before

Below are the rules for defining the *strong happened-before* relation (SHB or $\xrightarrow{S}$) for events of an execution involving send, receive, read and write statements.

SHB.1 If e and f are events on the same process such that e occurs before f, then $e \xrightarrow{S} f$.

SHB.2 If event e is a non-blocking send and event f is the corresponding receive, then $e \xrightarrow{S} f$.

SHB.3 If events e and f form a synchronous message-passing (i.e. one of them is a blocking send and the other is the corresponding receive), then for event g such that $e \xrightarrow{S} g$, we have $f \xrightarrow{S} g$, and for event h such that $h \xrightarrow{S} f$, we have $h \xrightarrow{S} e$.

SHB.4 For two different events e and f on shared variable V such that at least one of them is a write event, if $e \xrightarrow{ob(V)} f$ then $e \xrightarrow{S} f$.

SHB.5 For events e, f and g, if $e \xrightarrow{S} f$ and $f \xrightarrow{S} g$, then $e \xrightarrow{S} g$.

To simplify our notation, in the following we will use $\rightarrow$ for $\xrightarrow{S}$, if there is no ambiguity. Rules SHB.1, SHB.2, SHB.3 and SHB.5 are used in classical "happened before" [3], while rule SHB.4 is added to explicitly deal with read and write events. Notice that the new rule does not affect other rules. In [3] additional rules are used for process creation and termination. For the sake of simplicity, these rules are not mentioned here.

For events e and f of an execution, if neither $e \rightarrow f$ and nor $f \rightarrow e$, then e and f are said to be *concurrent,* indicated by $e \parallel f$. Thus, for two read events e and f in different processes on the same variable V, if no write event on V occurs between e and f and no "happened-before" relation between e and f exists due to messages or accesses to other variables, then $e \parallel f$.

For the events in fig. 1, according to SHB, $b \parallel c$, $a \rightarrow b$, $a \rightarrow c$, $a \rightarrow d$ and $b \rightarrow e$. Notice that $a \rightarrow d$ does not imply that event a in P_3 causally affects event d in P_1, since there is no flow of information from P_3 after event a to P_1 before event d. Similarly, $b \rightarrow e$ does not imply that event b in P_2 causally affects event e in P_3. According to the classical happened-before relation for send and receive events, event u happens before event v if and only if u causally affects v. However, this is not true for SHB, which covers send, receive, read and write events. Below there are two major considerations for defining a happened-before relation:

- **Causality:** the ability to determine the set of events that causally affects a given event.

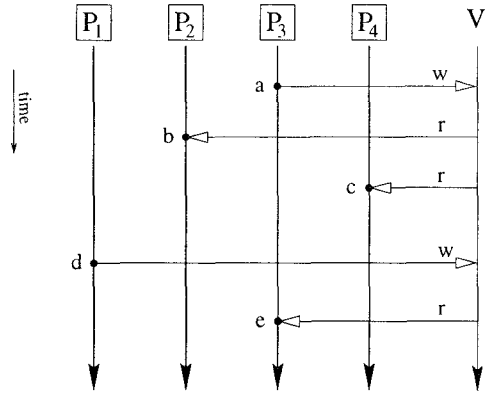

***Fig. 1* - Representation of a sequence of read and write events on the shared variable V.**

- **Reproducibility:** the ability to repeat a previous execution and thus produce the same results.

A happened-before relation is said to support causality if, for any event e in an execution, the set of events happening before e in this execution is exactly the set of events causally affecting e in this execution. A happened-before relation is said to support reproducibility if repeating the happened-before relations for all events in an execution guarantees repeating this execution. The classical happened-before relation supports both causality and reproducibility for programs using messages. SHB supports reproducibility, but not causality, for programs using both messages and shared variables. Below we define a different happened-before relation that supports causality, but not reproducibility, for programs using both messages and shared variables.

3.2 Weak happened-before

As mentioned earlier, the sequence of read and write events on a shared variable during an execution is a totally ordered sequence. Thus, the version number of a shared variable V during an execution can be defined as follows. Initially, the version number of V is zero. A write event on V increases the version number of V by one, while a read event on V keeps the version number of V intact. Let $v(V,e)$ denote the version number of V immediately after event e on V. According to SHB, the following two properties hold:

3.2.1 For a write event e and a read event f on variable V such that $v(V,e) = v(V,f)$, $e \rightarrow f$.

3.2.2 For events e and f on variable V such that $v(V,e) < v(V,f)$, $e \rightarrow f$.

Property 3.2.1 is needed for supporting causality, but property 3.2.2 is not. Below are the rules for defining the *weak happened-before* relation (WHB or $\xrightarrow{W}$) for

events of an execution involving send, receive, read and write statements.

WHB.1-3: same as SHB.1-3, except that $\xrightarrow{S}$ is replaced by $\xrightarrow{W}$.

WHB.4 For a write event e and a read event f on variable V such that $v(V,e) = v(V,f)$, $e \xrightarrow{W} f$.

WHB.5: same as SHB.5, except that $\xrightarrow{S}$ is replaced by $\xrightarrow{W}$.

WHB.4 is equivalent to the following rule, which does not use version numbers of events:

WHB.4* For a write event e and a read event f on variable V such that $e \xrightarrow{ob(V)} f$, if there is no write event w on V such that $e \xrightarrow{ob(V)} w \xrightarrow{ob(V)} f$, then $e \xrightarrow{W} f$.

In the following, we will use the symbol $\rightarrow$ for either WHB or SHB, if what it represents is clear from the context. For the events in fig. 1, according to WHB, $b \parallel c$, $a \rightarrow b$, $a \rightarrow c$, $a \parallel d$ and $b \parallel e$. The set of events happening before event c, according to SHB, is $\{a, b, c, d\}$, and the set of events happening before event c, according to WHB, is $\{d\}$. According to WHB, event e happens before event f if and only if e causally affects f. Thus, WHB supports causality for programs using both messages and shared variables. For the execution shown in fig. 1, repeating the WHB relations for all events in this execution does not guarantee that the final value of V be the value written by event d. Therefore, WHB does not support reproducibility for programs using both messages and shared variables.

4. Vector timestamps for SHB and WHB

In section 4.1, we show how to assign vector timestamps to send, receive, read and write events for SHB and WHB respectively. The assignment of timestamps is done on-the-fly (i.e. during the execution of these events). In section 4.2, we discuss how to use such timestamps to determine the existence of a SHB or WHB relation between two events. In the following discussion, we consider a concurrent program P containing processes $P_1, P_2, \ldots, P_n$, which communicate with each other by using messages and shared variables.

4.1 Timestamp assignment algorithms for SHB and WHB

For a send, receive, read, or write event e, let $T(e)$ contain n elements $T(e)[1], T(e)[2], \ldots, T(e)[n]$. For a message m, let $T(m)$ be the vector timestamp associated with m. For each process P_i, $1 \le i \le n$, we maintain C_i as the vector clock of P_i. Each C_i, $1 \le i \le n$, has a vector of zeros as its initial value. For each shared variable V, we keep two vector timestamps $T\_LastWrite(V)$ and $T\_Curr(V)$, which are defined as follows:

- $T\_LastWrite(V)$ contains the vector timestamp of the last write event on V.
- $T\_Curr(V)$ is the vector clock of V, which contains the up-to-date information for V.

Each of $T\_LastWrite(V)$ and $T\_Curr(V)$ has a vector of zeros as initial value. For two vector timestamps T and T', $T^* = max(T, T')$ is defined as
$T^*[i] = max(T[i], T'[i])$ for $1 \le i \le n$.

Algorithm SHB_VTS

This algorithm assigns vector timestamps for SHB. When process P_i, $1 \le i \le n$, is to execute an event e, it performs the following operations:

$C_i[i] = C_i[i]+1$;
$T(e) = C_i$;

Moreover, depending on the type of e, P_i performs the following operations:

1. If e is an internal event, P_i performs event e.
2. If e is a non-blocking send, P_i performs the send by sending a message and C_i, with C_i as the timestamp for the message.
3. If e is a receive with the receipt of a message m from a non-blocking send, P_i performs the following operations after event e:
   3.1. $C_i = max(C_i, T(m))$;
   3.2. $T(e) = C_i$;
4. If e is a blocking send with the corresponding receive in process P_j, P_i performs the send by sending a message and C_i, with C_i as the timestamp for the message. Then P_i performs the following operations:
   4.1. receive C_j from P_j;
   4.2. $C_i = max(C_i, C_j)$;
   4.3. $T(e) = C_i$;
5. If e is a receive with the receipt of a message m from a blocking send in process P_j, P_i performs the following operations after event e:
   5.1. send C_i to P_j;
   5.2. $C_i = max(C_i, T(m))$;
   5.3. $T(e) = C_i$;
6. If e is a write on shared variable V, P_i performs the following operations after event e:
   6.1. $C_i = max(C_i, T\_Curr(V))$;
   6.2. $T(e) = C_i$;
   6.3. $T\_LastWrite(V) = C_i$;
   6.4. $T\_Curr = C_i$;

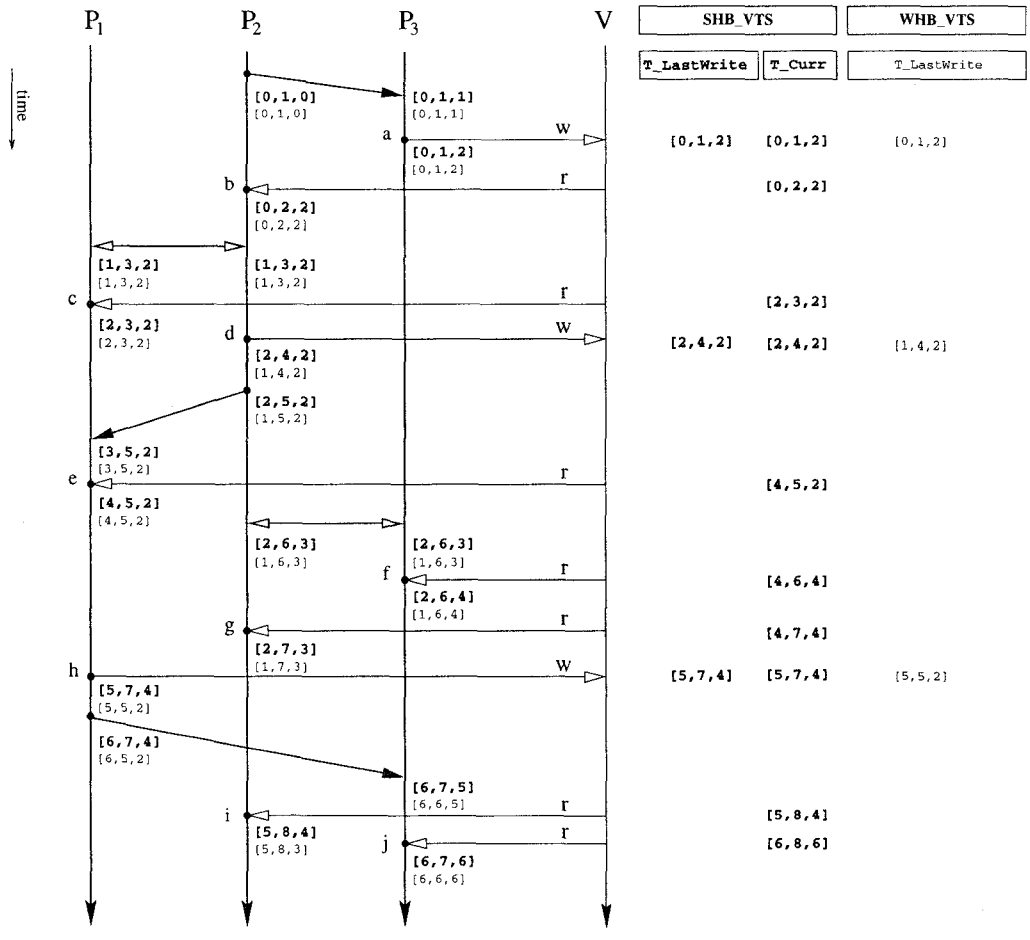

***Fig. 2* - An example of SHB and WHB vector timestamp assignment.**

7. If e is a read on shared variable V, P_i performs the following operations after event e:
   7.1. $C_i = max(C_i, T\_LastWrite(V))$;
   7.2. $T(e) = C_i$;
   7.3. $T\_Curr(V) = max(C_i, T\_Curr(V))$;

Rules 2 through 5 are equivalent to the rules in [3] for assigning vector timestamps to send and receive events. Rules 6 and 7 are equivalent to the rules used in [7] for assigning timestamps to read and write events. These two sets of rules can be combined without creating any conflicts.

For a write event (in rule 6), C_i is set to $max(C_i, T\_Curr(V))$. For a read event (in rule 7), C_i is set to $max(C_i, T\_LastWrite(V))$. The reason for the difference is the following. A write event on V happens before all following read and write events on V. A read event on V is concurrent with other read events on V that happens after the most recent write event on V and before the next write event on V, if there are no happened-before relations between these read events due to messages or accesses to other shared variables.

Algorithm WHB_VTS

This algorithm assigns vector timestamps for WHB. In WHB, a write event on a shared variable V happens before all following read events on V before the next write event on V. However, a write event on V is concurrent with other write events on V, if there are no happened-before relations between these events due to messages or accesses to other shared variables. Therefore, $T\_Curr(V)$ is not needed for V. Algorithm WHB_VTS is the same as algorithm SHB_VTS except that rules 6 and 7 are changed as follows:

6. If e is a write on shared variable V, P_i performs the following operations after event e:
   6.1. $T\_LastWrite(V) = C_i$;
7. If e is a read on shared variable V, P_i performs the following operations after event e:
   7.1. $C_i = max(C_i, T\_LastWrite(V))$;
   7.2. $T(e) = C_i$;

Fig. 1 shows vector timestamps, according to algorithms SHB_VTS and WHB_VTS, for events in an execution involving asynchronous and synchronous message-passing and accesses to shared variable V. In fig. 2, an arrow with one black head denotes an asynchronous message-passing, a horizontal arrow with one white head on each side denotes a synchronous message-passing, and a horizontal arrow with one white head denotes a read or write operation on a shared variable. Each event in fig.2 has two vector timestamps based on SHB and WHB respectively, with the SHB-based timestamp in boldface. Values of *T_LastWrite(V)* and *T_Curr(V)* for each read or write event on V are also shown in fig. 2.

For a concurrent program P with n processes and s shared variables, the size of a vector timestamp for an event in P is n. Algorithm SHB_VTS requires the use of a vector clock for each process and two vector timestamps for each shared variable. Assume that the timestamps for events of an execution are logged into a file, not save in memory. For an execution of P, the space complexity of algorithm SHB_VTS is $n(n + 2s)$. In contrast, the space complexity of algorithm WHB_VTS is $n(n + s)$. Algorithms SHB_VTS and WHB_VTS assign timestamps on-the-fly, and they can be modified to assign timestamps in post-mortem fashion (i.e. after the collection of events).

4.2 Use of SHB and WHB timestamps for event ordering

Below we summarize the results of using vector timestamps to determine the classical happened-before relation between two events of an execution involving asynchronous and synchronous message-passing [3, 6]. We claim that the same results hold for using vector timestamps to determine the SHB or WHB relation between two events of an execution involving asynchronous and synchronous message-passing and accesses to shared variables. The proofs for our claim are omitted in this paper.

Definition 4.1 For vector timestamps u and v of dimension n,

- $u \le v$ iff $u[k] \le v[k]$ for $k \in [1, \ldots, n]$
- $u < v$ iff $u \le v$ and $u \ne v$
- $u \parallel v$ iff $\neg(u < v)$ and $\neg(v < u)$

Observation 4.1 For events e and f of the same execution, $T(e) = T(f)$ iff $e = f$ or e and f are involved in the same synchronous message-passing.

Theorem 4.1 Assume that e and f are events of the same execution.

a) $e \rightarrow f$ iff $T(e) < T(f)$
   $e \parallel f$ iff $T(e) \parallel T(f)$
b) If e and f are events of processes P_i and P_j respectively,
   $e \rightarrow f$ iff $T(e)[i] \le T(f)[i]$ and $T(e)[j] < T(f)[j]$
c) If e and f are events of processes P_i and P_j respectively such that $i \ne j$ and e and f are not involved in the same synchronous message-passing,
   $e \rightarrow f$ iff $T(e)[i] \le T(f)[i]$

5. Applications of SHB and WHB timestamps

Vector timestamps based on the classical happened-before relation have been used in many techniques for analysis, testing and debugging of message-passing programs. Examples of such techniques are message-race detection [8], data-race detection [2], detection of global predicates [4], execution replay [8], deterministic testing [11], and reachability testing [13]. (See [1, 9] for more references). SHB and WHB vector timestamps can be used in similar techniques for analysis, testing and debugging of programs using either shared variables only or both messages and shared variables. However, for an analysis, testing or debugging technique, we may choose the use of SHB or WHB vector timestamps according to whether causality or reproducibility is needed. (This problem does not exist for message-passing programs since the classical happened-before relation supports both causality and reproducibility). For example, a debugging technique may use SHB timestamps to perform execution replay, and it may use WHB timestamps to determine the set of events that causally affect the value of a shared variable.

In section 5.1, we briefly describe an analysis technique, called *race analysis*, for message-passing programs. In section 5.2, we show how to extend this technique to programs using messages and shared variables by using WHB timestamps.

5.1 Race analysis for message-passing programs

Let P be a message-passing program containing processes $P_1, P_2, \ldots, P_n$, $n > 1$. A (partially-ordered) trace of P contains one totally-ordered sequence of events for each process in P. Let Q be the trace of an execution of P with input X. $Q = (Q_1, Q_2, \ldots, Q_n)$, where Q_i, $1 \le i \le n$, is a sequence of send events with P_i as the sender and receive events with P_i as the receiver. The race set for a receive event *rc* in Q is the set of messages in Q that could be received at *rc* during an execution of P that repeats all events causally affecting *rc* in

Q. The race sets for receive events in Q are useful for testing and debugging P. Below are some examples:

- An execution of P with input X is deterministic (i.e. no message race) if and only if the race set for each receive event in Q is empty.
- For a receive event in Q, determine whether its race set contains unintended messages.
- The race sets for receive events in Q can be used to construct prefixes of other possible traces of P with input X. Such prefixes can be used for testing P [11, 13].

Race analysis of a trace of send and receive events refers to analysis of the trace in order to determine the race set for each receive event in the trace. A race analysis algorithm was given for a trace of send and receive events involving asynchronous message-passing [12]. Two additional race analysis algorithms were also given for traces based on two special types of asynchronous message-passing [12]. These algorithms use vector timestamps based on the classical happened-before relation.

5.2 Race analysis for programs using messages and shared variables

Let P be a program using messages and shared variables. Assume that P contains processes $P_1, P_2, \ldots, P_n$, $n > 1$. Let Q be the trace of an execution of P with input X. $Q = (Q_1, Q_2, \ldots, Q_n)$, where Q_i, $1 \le i \le n$, is a sequence of send events with P_i as the sender, receive events with P_i as the receiver, and read and write events as P_i as the executing process. The race set of a read event rd in Q is the set of write events such that their values could be read by rd during an execution of P that repeats all events causally affecting rd in Q. The race set for a receive event in Q is the same as that given in section 5.1. An execution of P with input X is deterministic (i.e. no message or data race) if and only if the race set for each read or receive event in Q is empty.

For a read event rd on shared variable V in Q, let $race(Q,rd)$ denote the race set for rd and let w be the corresponding write event in Q, i.e. $v(V,rd) = v(V,w)$. During an execution of P that repeats all events causally affecting rd in Q, consider a write event w' on V in Q, where $w' \neq w$, for the following two cases:

(a) w' is concurrent with rd in Q according to WHB. In this case, the value of w' could be read by rd during this execution.

(b) w is not concurrent with rd in Q according to WHB. In this case, the value of w' can never be read by rd during this execution.

Thus we have the following theorem.

Theorem 5.1 Let Q be the trace of an execution of a program using messages and shared variables. For a read event rd in Q on shared variable V,

$race(Q,rd)$ = {write events w on V in Q such that $w \parallel rd$ according to WHB}.

According to Theorem 5.1, we need to use WHB timestamps to construct the race set for a read or receive event. Below we show an algorithm for determining the race sets for read events in a trace of a program using messages and shared variables. This algorithm can be combined with the algorithms in [12] to determine the race sets for all read and receive events in a trace.

For an even e in Q, let $T(e)$ denote the WHB vector timestamp for e. Let Q[i,j] refer to the j^{th} event of Q_i. In the following algorithm, for a read event Q[i,j], we search for write events in Q_k, where $1 \le k \le n$, and $k \neq i$, that write on V and are concurrent with Q[i,j]. For such a write event Q[k,s],

$\neg(Q[k,s] \rightarrow Q[i,j])$ and $\neg(Q[i,j] \rightarrow Q[k,s])$.

According to Theorem 4.1 c),

$T(Q[k,s])[k] > T(Q[i,j])[k]$ and $T(Q[i,j])[i] > T(Q[k,s])[i])$.

Therefore, for events in Q_k, we check event Q[k,s] with $s = T(Q[i,j])[k]+1, T(Q[i,j])[k]+2, \ldots,$ until either $T(Q[k,s])[i] \ge T(Q[i,j])[i]$ or the end of Q_k is reached. If a checked event is a write on V, then this write event is added to the race set for Q[i,j].

Algorithm Race_Read

```
<for each read event r in Q> {race(Q,r)=<empty set>;}
for (i=1; i<= n; i++) {
  for (j=1; j<=length(Qi); j++) {
    if ( <Q[i,j] is a read event on V> ) {
      for (k=1; k<=n; k+=(k!=i-1)?1:2 ) {
        s:=T(Q[i,j])[k]+1;
        while(T(Q[i,j])[i]>T(Q[k,s])[i]
              && s<length(Qk)) {
          if ( <Q[k,s] is a write on V> ) then {
            <add Q[k,s] to race(Q,Q[i,j])>;
          }
          s++;
} } } } }
```

Let Ne and $Ne(i)$ be the numbers of events in Q and Q_i, respectively. Similarly, let Nr and $Nr(i)$ be the numbers of read events in Q and Q_i, respectively. The time complexity of this algorithm is

$$O\left(\sum_{i=1}^{n}\left(Ne(i)+Nr(i)\sum_{k\neq i}Ne(k)\right)\right)$$

According to algorithm Race_Read, the race sets for some read events in fig. 3 are given below: $race(Q,c) = \{d\}$, $race(Q,e) = \{\}$, $race(Q,f) = \{h\}$.

One idea for reducing the time complexity of algorithm Race_Read is to keep all write events on the same shared variable as a list in the order of accesses to the shared variable. Let such a list for shared variable V be

referred to *WList*(V). Such lists can be constructed by modifying rule 6 in algorithm WHB_VTS as follows.

6. If e is a write on shared variable V, P_i performs the following operations after event e:
   6.1. T_LastWrite(V) = C_i;
   6.2. add (i,C_i[i]) to the end of *WList*(V);

Below we present a revised version of algorithm Race_Read, called Race_Read_WList, by using the lists of write events for shared variables.

Algorithm Race_Read_WList

```
<for each read event r in Q> {race(Q,r)=<empty set>;}
for (i=1; i<= n; i++) {
  for (j=1; j<=length(Qi); j++) {
    if ( <Q[i,j] is a read event on V> ) {
      for (k=1; k<=length(WList(V)) {
        <let the kth element in WList(V) be (u,v)>
        if( u != i && T(Q[u,v])[u] > T(Q[i,j])[u]
           && T(Q[i,j])[i] > T(Q[u,v])[i] ) {
           <add [u,v] to race(Q,Q[i,j]).
} } } } }
```

The time complexity of algorithm Race_Read_WList is *O(Ne·max(length(WList(V))* for each shared variable V))

6. Conclusions

In this paper we have extended the classical happened-before to define two happened-before relations, called strong happened-before (SHB) and weak happened-before (WHB), for programs using messages and shared variables. SHB and WHB differ in that the former supports reproducibility, but not causality, and the latter supports causality, but not reproducibility. We have presented timestamp assignment algorithms for SHB and WHB, and shown the use of SHB and WHB timestamps to determine event ordering. In addition, we have described how to apply WHB timestamps to perform race analysis for programs using messages and shared variables. We are investigating other applications of SHB and WHB timestamps to solve problems in analysis, testing and debugging of programs using messages and shared variables.

Acknowledgments

The authors are grateful to the anonymous reviewers for their helpful comments on the previous version of this paper.

References

[1] K. Audenaert, "Clock Trees: Logical Clocks for Programs with Nested Parallelism," *IEEE Trans. On Software Eng.*, Vol. 23, No. 10, Oct. 1997

[2] A. Dinning and E. Schonberg, "An Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection," *ACM Symp. on Principles and Practice of Parallel Programming*, March 1990, pp. 1-10

[3] C. J. Fidge, "Logical Time in Distributed Computing Systems," *IEEE Computer*, Aug. 1991, pp. 28-33

[4] V. K. Garg and B. Waldecker, "Detection of Strong Unstable Predicates in Distributed Programs," *IEEE Trans. On Parallel and Distributed Systems,* Vol. 7, No. 12, Dec. 1996, pp. 1323-1333

[5] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. ACM*, July 1978, pp. 558-565

[6] F. Mattern, "Virtual Time and Global States of Distributed Systems," Parallel and Distributed Algorithms (M. Cosnard et al. Eds.), Elsevier Science, North Holland, 1989, pp. 215-226

[7] R. H. B. Netzer, "Optimal Tracing and Replay for Debugging Shared-Memory Parallel Programs," *3rd ACM/ONR Workshop on Parallel and Distributed Debugging*, May 1993

[8] R. H. B. Netzer and B. P. Miller, "Optimal Tracing and Replay for Debugging Message-Passing Parallel Programs," *The Journal of Supercomputing,* Vol. 8, No. 4, 1994, pp. 371-388

[9] R. Schwarz and F. Mattern, "Detecting Causal Relationships in Distributed Computations: in Search of the Holy Grail," *Distributed Computing* (1994) 7, pp. 149-174 52

[10] M. Singhal and A. Kshemkalyani, "An efficient implementation of vector clocks," *Information Processing Letters,* Vol. 43, 1992, pp. 47-52

[11] K. C. Tai and R. H. Carver, "Testing of Distributed Programs," chapter 33 of *Handbook of Parallel and Distributed Computing*, ed. A. Zomaya, McGrow-Hill, 1996, pp. 955-978

[12] K. C. Tai, "Race Analysis of Traces of Asynchronous Message-Passing Programs," *Proc. of IEEE 17th Intern. Conf. on Distributed Computing System*, May 1997, pp. 261-268

[13] K. C. Tai, "Reachability Testing of Asynchronous Message-Passing Programs," *Proc. IEEE Int. Workshop on Software Engineering for Parallel and Distributed Systems,* May 1997, pp. 50-61

[14] A.S. Tanenbaum, *Distributed Operating Systems,* Prentice-Hall, 1995