A Theory of Data Race Detection

Utpal Banerjee, Brian Bliss, Zhiqiang Ma, Paul Petersen

Intel Corporation

{utpal.banerjee, brian.e.bliss, zhiqiang.ma,paul.petersen}@intel.com

ABSTRACT

This paper presents a rigorous mathematical theory for the detection of data races in threaded programs. After creating a structure with precise definitions and theorems, it goes on to develop four algorithms with the goal of detecting at least one race in the situation where the history kept on previous memory accesses is limited. The algorithms demonstrate the tradeoff between the amount of access history kept and the kinds of data races that can be detected. One of these algorithms is a reformulation of a previously known algorithm; the other three are new. One of the new ones is actually used in the tool called Intel® Thread Checker.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; D.1.3 [**Programming Techniques**]: Concurrent Programming

General Terms

Algorithms, Theory, Reliability

Keywords

Data Race, Access Conflict, Thread, Synchronization, Vector Clock, Happens-before, Dependence

1. INTRODUCTION

Consider a programming environment where a number of threads are active simultaneously. The instructions in these threads can be arbitrarily interleaved. If two threads access the same location x in shared memory and at least one of the accesses is a 'write,' then the final outcome of the program may depend on the order of the two accesses. This is another way of saying that a *data race* may exist in x.

A number of researchers have worked on data race detection. Netzer and Miller gave formal definitions and characterizations of different types of races in [6]. They also show that the problem of detecting data races in a program is NP-hard. There is no efficient tool that can detect all potential races in the source code of a given parallel program. There is an *on-the-fly* approach to race detection that tries to find at least one data race that actually happens during a given execution of a program [8]. In such an approach, one keeps records on accesses to various memory locations. When a given location is accessed, the record on that access is compared with records on previous accesses to the same location to determine if a data race exists for that location.

A thread is divided into parts called *segments*, and segments of different threads are partially ordered based on the way threads communicate among themselves. We keep track of this partial order by assigning an integer vector called *vector clock* to each segment. Whether or not there is a *data race* between two segments in a memory location they both access is then determined by comparing the vector clocks of the segments. Due to practical considerations, it is not possible to always carry the information on all segments that have already accessed a given memory location. This implies that we may not be able to detect *all* races in that location. The focus then shifts to the question: *Can we report that there exists at least one race when races are present?*

Algorithms for detecting at least one data race have been given in [3], [7], [8], and [9]. In this paper, we present a rigorous mathematical framework in which one can study the tradeoff between the amount of access history kept and the kinds of data races that can be detected. We derive four algorithms based on theorems that show step by step how our prediction capability improves by keeping more and more extensive history. The first two algorithms on race detection can detect races in many situations, and the last two algorithms can detect races in every situation. Our basic approach is similar to that in [3], [7], [8], and [9]. The fourth algorithm on race detection (Algorithm 5) is actually the algorithm used in [3] and [8], stated in our framework. It is described in Section 3.4.

An on-the-fly technique reports only races that happen in a particular execution of the given program. A technique based on *model checking* searches exhaustively for races and can find those in execution paths rarely taken. However, the search space for such technique could be huge. A hybrid scheme is presented in [10] that tries to combine the advantages of both approaches while minimizing the difficulties of both.

For threaded programming, Intel has produced a suite of tools one of which is the Intel® Thread Checker. It is a dynamic analysis tool for detecting violations in the use of threading and synchronization API's, including incorrect argument usage, incorrect lock usage, and incorrect usage of shared variables. The last item gives rise to data races caused by the lack of synchronization. The second algorithm on race detection (Algorithm 3) presented in this paper is the one that the Thread Checker uses. Implementation issues for this algorithm are discussed in [1].

Section 2 builds up the basic structure of the theory, while Section 3 studies in detail the problem of race detection with limited history. We describe the prior work relevant to this paper in Section 4, and finally, some conclusions are given in Section 5. Theorems, lemmas, and remarks are numbered consecutively in the same sequence. Algorithms are numbered separately.

2. BASIC CONCEPTS

A *thread* in a program is a sequence of instructions that can be run independently. We consider an execution of a program on a multiprocessor system in an environment where multiple threads can be run in parallel. Typically, the program starts out as a single thread which then creates several other threads, and as execution continues more threads are created and terminated. The goal of this section is to explain the theoretical basis of a structure that enables us to detect data races in a multi-threaded program.

In the subsections that follow, we define precisely the parts of a thread called *segments*, the precedence order between segments, and the *vector clocks* of segments that represent this order. We give an algorithm for the computation of vector clocks, and derive the condition (involving vector clocks) under which two segments are parallel. We define *data races* and derive conditions for their existence.

2.1 Segments of a Thread

Creation or termination of threads and communication among threads are all controlled by executing certain special sequences of instructions called *synchronization operations*, or *sync ops* for short. It is convenient to distinguish between two types of sync ops: *posting* and *receiving*. A sync op executed on a thread T is a *posting* sync op, if it posts some information carried by T that can be read later by T itself or by some other thread. A sync op executed on a thread T is a *receiving* sync op, if it reads the information already posted by one or more posting sync ops. A given sync op cannot both post and receive. However, the information posted by a single posting op can be read by more than one receiving op, and a given receiving op may read information posted by more than one posting op.

We assume that when a thread T_0 creates a thread T, a posting sync op A_0 is executed on T_0 and a receiving sync op A is executed on T. Thus, T starts with a receiving sync op. (In the special case when T is the main thread with which a given program starts, the role of its creator T_0 is played by the operating system.) When T ends, its last action is to execute a posting sync op. At any point during its lifetime, T may execute a posting or a receiving sync op to communicate with other threads (or itself).

The sync ops on a given thread are executed in a definite order. These operations are used to partition the thread into parts called *segments*. A *segment* of a thread is a maximal sequence of instructions containing exactly one sync op that ends the sequence. If a segment S ends with a sync op A, we say that S is *defined* by A. Also, S is a *posting* or a *receiving* segment, if A is a posting or a receiving sync op, respectively. In the previous paragraph, the segment of T_0 that ends with A_0 is a posting segment, and the first segment of T is a receiving segment consisting only of the receiving sync op A. Also, the last segment of any thread is a posting segment.

Consider a thread T and label the distinct sync ops on it by $A_1, A_2, \ldots, A_n$ in the order of their execution. Let S_k denote the segment defined by A_k, where $1 \leq k \leq n$. For $2 \leq k \leq n$, the segment S_k consists of the sequence of instructions between A_{k-1} and A_k, including A_k and excluding A_{k-1}. The first segment S_1 consists only of the instructions belonging to the sync op A_1.[1] The segments of T are executed in the order: $S_1, S_2, \ldots, S_n$. In a given segment S_k, the instructions are executed sequentially. The sync op A_k is always executed last, but the other instructions in S_k, if any, are executed in an unspecified order. No instructions of S_k are executed before the execution of A_{k-1} is complete.

A segment on a thread knows exactly how many segments on that thread have already executed. However, it has only a partial knowledge of how many segments on a *different* thread have already executed. Any such knowledge that the segment has is based upon the information, if any, that its thread has already received by executing receiving sync ops. Since a global clock is not available, we have to base our analysis on this incomplete knowledge.

We formalize the ideas of the previous paragraph by defining a relationship $\prec$ among segments of the entire program as follows. Let S denote a segment on a thread T and S' a segment on a thread T'. We have $S \prec S'$, if one of the following holds:

1. $T = T'$ and S is executed before S'.

2. S is a posting segment on T defined by a sync op A, S' immediately follows a receiving segment on T' defined by a sync op B, and B reads the information posted by A. (See Figure 1.)

3. There exists a finite sequence of segments $S_0, S_1, S_2, \ldots, S_m$ in the program with
$$S = S_0 \prec S_1 \prec S_2 \prec \cdots \prec S_{m-1} \prec S_m = S',$$

such that for $0 \leq p \leq m-1$, the relation $S_p \prec S_{p+1}$ holds in the sense of either Condition 1 or Condition 2.[2] If $S \prec S'$, the segments S and S' must be distinct. As usual, the notation $S \preceq S'$ means either $S \prec S'$ or $S = S'$. It is clear that $\preceq$ is a partial order on the set of segments in the program.

We have collected below some statements that should clear up possible misconceptions on the definition of the relation $\prec$.

Remarks 1

1. If $S \prec S'$, then segment S must finish executing before segment S' starts (in the particular program execution under consideration).

[1]This is not unique to S_1. An arbitrary segment S_k need not contain any instructions other than those belonging to its sync op A_k.

[2]Condition 3 can now be taken as the general definition of $\prec$, since the other two conditions are its special cases.

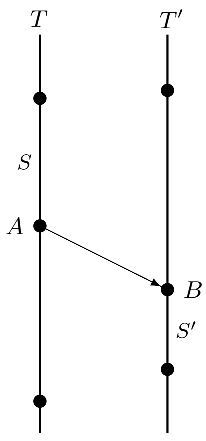

Figure 1: Segment S of Thread T precedes Segment S' of Thread T'.

2. Even if S finishes executing before S' starts, the relation $S \prec S'$ may or may not be true.

3. If S and S' overlap or S finishes before S' starts, then $S' \prec S$ is false.

4. The relation $\prec$, when restricted to segments of a fixed thread, is a total order. If S and S' are both segments of the same thread, then it is true that $S \prec S'$ if and only if S is executed before S'.

We often find it convenient to say in words that "*S precedes S'*" to indicate that $S \prec S'$. From now on, *precedence* among segments should always be understood in this narrowly restricted sense.

We end this subsection by introducing the concept of two segments being *parallel.* This is crucial in the analysis for data race detection. Two distinct segments S and S' in the given program are *parallel*, and we write $S \parallel S'$, if they are not comparable in terms of the relation $\prec$. In other words, we have $S \parallel S'$ if and only if $S \prec S'$ *and* $S' \prec S$ are both false. It is worthwhile to remember that when two segments are parallel, they are necessarily distinct and they belong to two distinct threads.

2.2 Vector Clock of a Segment

We need a numerical representation for the partial order $\preceq$, so that given two thread segments one can easily decide whether or not they are parallel. To this end, we associate an integer vector called *vector clock* with each segment S; it keeps track of how many posting segments on various threads precede S.

Consider any segment S on any thread T in the program. Let $\mathcal{T}_S$ denote the set of all threads *known* to S. This set consists of T itself, and every other thread T' with a posting segment S' such that $S' \prec S$. The *vector clock* of S is a function $V_S : \mathcal{T}_S \to \{0, 1, 2, \ldots, \infty\}$ defined as follows: For all $T' \in \mathcal{T}_S$,

$$V_S(T') = \text{[Number of posting segments } S' \text{ on } T' \text{ such that } S' \prec S].$$

We adopt the convention that $V_S(T') = 0$ if T' is not known to S (i.e., if $T' \notin \mathcal{T}_S$). Then, the vector clock $V_S(T')$ is defined for each segment S and each thread T'. This makes it possible to compare two vector clocks whose domains of definition were originally different.

The vector clock of a segment is determined by the vector clocks of its immediate predecessors, if any. We now give a detailed discussion of this relationship that will lead to an algorithm for the computation of vector clocks.

As before, S is a segment of a thread T. First, let S be the first segment of T. Then, there are no segments that precede S and the only thread known to S is T. This means $\mathcal{T}_S = \{T\}$ and $V_S(T) = 0$.

Assume next that S is not the first segment of T. Let S_1 denote the segment of T that immediately precedes S. Two cases arise.

First, suppose S_1 is a posting segment. On T, the number of posting segments preceding S is one more than the number of posting segments preceding S_1, since S_1 itself is a posting segment and it precedes S. Hence, $V_S(T) = V_{S_1}(T) + 1$. Any information about other threads that T may have at the start of S was already available at the start of S_1. A posting segment S' on a thread $T' \neq T$ precedes S iff it precedes S_1. Hence, a thread $T' \neq T$ is known to S iff it is known to S_1, and on such a thread the set of posting segments preceding S is identical to the set of posting segments preceding S_1. In other words, $\mathcal{T}_S = \mathcal{T}_{S_1}$ and we have $V_S(T') = V_{S_1}(T')$ for each $T' \in \mathcal{T}_S$ other than T.

Second, suppose S_1 is a receiving segment. Computation of V_S is now a little more complicated. Let $\mathcal{P}_S$ denote the set of all immediate predecessors of S. (This set, of course, contains S_1.) A thread is known to S if and only if it is known to at least one member of $\mathcal{P}_S$. Hence, we have $\mathcal{T}_S = \bigcup\{\mathcal{T}_{S'} : S' \in \mathcal{P}_S\}$. We need to compute the value of V_S on each thread in $\mathcal{T}_S$.

On T itself, the set of posting segments that precede S is now exactly the same as the set of posting segments that precede S_1, since S_1 is not a posting segment. Hence, we have $V_S(T) = V_{S_1}(T)$.

Next, let T' denote any thread other than T known to S. The value $V_S(T')$ is the number of posting segments of T' that precede S. Each immediate predecessor of S brings a count of such segments. We will get $V_S(T')$ by taking the maximum of all such counts. Let S' denote any immediate predecessor of S. If S' is on T', then the number of posting segments on T' that precede S is at least one more than the number of posting segments that precede S', since S' itself is a posting segment on T' preceding S. Thus, the contribution of S' to $V_S(T)$ in this case is $[V_{S'}(T') + 1]$. If S' is not on T', then its contribution to $V_S(T)$ is simply $V_{S'}(T')$. Taking account of all these contributions, we get

$$V_S(T') = \max\{V_{S'}(T') + \delta(S', T') : S' \in \mathcal{P}(S)\},$$

where $\delta(S', T') = 1$ if S' is on T', and $\delta(S', T') = 0$ otherwise.

The above discussion is encapsulated in the form of an algorithm.

Algorithm 1 (Computation of Vector Clocks) This algorithm shows how to compute the vector clock of any segment of any thread in a given program. It has to be applied recursively starting with the first segment of the original master thread that starts the program. The set of threads known to a segment S is denoted by $\mathcal{T}_S$. The vector clock of S is denoted by V_S; it is a mapping of $\mathcal{T}_S$ to the set of nonnegative integers. For any given segment S, we find the set $\mathcal{T}_S$, and compute the value of V_S at each member T' of $\mathcal{T}_S$. This is done by processing the corresponding items of the immediate predecessors of S.

```
If $S$ is the first segment of a thread $T$, then
    $\mathcal{T}_S \leftarrow \{T\}$
    $V_S(T) \leftarrow 0$
else
    $S_1 \leftarrow$ the immediate predecessor of $S$ on $T$
    if $S_1$ is a posting segment, then
        $\mathcal{T}_S \leftarrow \mathcal{T}_{S_1}$
        $V_S(T) \leftarrow V_{S_1}(T) + 1$
        $V_S(T') \leftarrow V_{S_1}(T')$ for every other $T' \in \mathcal{T}_S$
    else
        $\mathcal{P}_S \leftarrow$ the set of immediate predecessors of $S$
        $\mathcal{T}_S \leftarrow \bigcup\{\mathcal{T}_{S'} : S' \in \mathcal{P}_S\}$
        $V_S(T) \leftarrow V_{S_1}(T)$
        for $T'(\neq T)$ in $\mathcal{T}_S$
            $V_S(T') \leftarrow \max\{V_{S'}(T') + \delta(S', T') : S' \in \mathcal{P}_S\}$
```

(As defined above, $\delta(S', T') = 1$ if S' is on T', and $\delta(S', T') = 0$ otherwise.) Note that according to our convention $V_{S'}(T')$ is always defined: if a thread $T' \in \mathcal{T}_S$ is not known to some predecessor S' of S, we take the default value 0 for $V_{S'}(T')$.

Remarks 2 Note that by counting only posting segments in the definition of a vector clock, we keep the values of vector clocks relatively low. But this also means that two consecutive segments on the same thread may have identical vector clocks. Let S_1 and S denote two consecutive segments on a thread T, such that S comes after S_1. Suppose S_1 is a receiving segment, and the sync op defining it has only read information posted by a sync op executed on T itself a while back. Then we have $\mathcal{T}_S = \mathcal{T}_{S_1}$, and $V_S(T') = V_{S_1}(T')$ for each $T' \in \mathcal{T}_S$.

The following theorem gives a necessary and sufficient condition for a segment S to precede another segment S' in terms of their vector clocks.

Theorem 3. *Let T and T' denote two distinct threads, S a segment on T, and S' a segment on T'. Then $S \prec S'$ if and only if $V_S(T) < V_{S'}(T)$.*

Proof. The 'only if' Part. Let $S \prec S'$. We show that $V_S(T) < V_{S'}(T)$. By definition, there exists a finite sequence of segments $S_0, S_1, S_2, \ldots, S_m$ in the program, such that $S_0 = S$, $S_m = S'$, and

$$S = S_0 \prec S_1 \prec S_2 \prec \cdots \prec S_{m-1} \prec S_m = S',$$

where each of the relations $S_p \prec S_{p+1}$ holds in the sense of either Condition 1 or Condition 2 in the definition of $\prec$.

Since $S = S_0$ is on thread T and $S' = S_m$ is not, there exists a smallest integer p such that the segment S_p is on T, but the segment S_{p+1} is not. Then for $S_p \prec S_{p+1}$ to hold, S_p must be a posting segment. From the relations $S \preceq S_p \prec S_{p+1} \preceq S'$, it follows that the number of posting segments on T preceding S' is at least one more than the number of posting segments on T preceding S. Therefore, $V_S(T) < V_{S'}(T)$.

The 'if' Part. Let $V_S(T) < V_{S'}(T)$. We need to show that $S \prec S'$. Let $S_1, S_2, \ldots, S_n$ denote all the posting segments on T, labelled such that $S_1 \prec S_2 \prec \cdots \prec S_n$. Write $k = V_S(T)$, so that there are exactly k posting segments on T preceding S. Since these must be the *first* k posting segments on T, we have

$$S_1 \prec S_2 \prec \cdots \prec S_k \prec S \preceq S_{k+1}. \tag{1}$$

Since $V_{S'}(T) \geq V_S(T) + 1 = k + 1$, there are at least $(k+1)$ posting segments on T preceding S'. Again, this means that the *first* $(k+1)$ posting segments on T must precede S', so that

$$S_1 \prec S_2 \prec \cdots \prec S_k \prec S_{k+1} \prec S'. \tag{2}$$

From (1) and (2) we get $S \preceq S_{k+1} \prec S'$, that is, $S \prec S'$. ■

Corollary 1. *A segment S on a thread T is parallel to a segment S' on a thread T', iff $V_S(T) \geq V_{S'}(T)$ and $V_{S'}(T') \geq V_S(T')$.*

Proof. Note that $S \parallel S'$ holds iff the segments lie on distinct threads, and both $S \prec S'$ and $S' \prec S$ are false. However, by Theorem 3, $S \prec S'$ is false iff $V_S(T) \geq V_{S'}(T)$, and $S' \prec S$ is false iff $V_{S'}(T') \geq V_S(T')$. ■

2.3 Data Races

During its lifetime, a typical thread accesses (reads and writes) several locations in the shared memory. A race condition may be present when there is nothing to prevent two threads from accessing the same location simultaneously. There is a *data race* between two segments, if the segments are parallel, and if there is a location in shared memory accessed by both such that one of the accesses is a 'write.' Data races in a given threaded program depend very much on the particular execution of the program being considered.

We state now a set of necessary and sufficient conditions for two segments to have a data race, in terms of their vector clocks and the sets of locations read and written by them.

Theorem 4. *Let S denote a segment on a thread T and S' a segment on a different thread T'. Let R_S denote the set of memory locations read and W_S the set of locations written by the segment S, and similarly for the segment S'. Then there is a data race between S and S', iff the following two conditions hold:*

1. *$V_S(T) \geq V_{S'}(T)$ and $V_{S'}(T') \geq V_S(T')$;*
2. *Either $[R_S \cup W_S] \cap W_{S'} \neq \emptyset$, or $[R_{S'} \cup W_{S'}] \cap W_S \neq \emptyset$.*

Proof. By definition, there is a data race between S and S', iff (a) $S \parallel S'$, and (b) either S' writes at least one shared memory location that is accessed by S, or S writes at least one location that is accessed by S'. Condition 1 in the theorem represents (a) by the corollary of the last section, and Condition 2 is just the symbolic version of (b). ■

We now look at the race detection problem from the perspective of a single location in shared memory. A *data race* in a shared memory location x exists, if there are two parallel segments that access x and at least one of the accesses is a 'write.' The race detection scheme used by the Thread Checker is based on the following result that is derived from Theorem 4.

Theorem 5. *Let x denote a location in the shared memory. Assume that x has been accessed first by a segment S on a thread T, and then by a segment S' on a different thread T'. Then there is a data race in x, if*

1. $V_S(T) \geq V_{S'}(T)$ *and*

2. $x \in W_S \cup W_{S'}$.

Proof. Note that $S' \prec S$ must be false in this case, since otherwise S' would have finished executing before S started (see Remark 1.1.). So, by Theorem 3, we must already have $V_{S'}(T') \geq V_S(T')$. Thus, the second inequality in the first condition of Theorem 4 is already satisfied, and we need only check the first inequality.

Also, we already know here that $x \in R_S \cup W_S$ and $x \in R_{S'} \cup W_{S'}$. So, we need only check if $x \in W_S$ or $x \in W_{S'}$.■

In the next section, we will analyze a race in x in much finer detail. To that end, we distinguish between data races of different types. We need some definitions to classify the various ways in which the accesses to x by two segments can be related. First, there is an *access conflict* in x, if there are two parallel segments such that they both access x. Since an access is either a 'read' or a 'write,' there are four *types* of access conflicts. A *flow conflict* in x exists, if there are two parallel segments S and S', such that x is first written by S and then read by S'. The other three types of conflict in x are defined similarly in terms of two parallel segments that access x: An *anti conflict* exists if x is first read by S and then written by S', an *output conflict* exists if x is written first by S and then by S', and we get an *input conflict* if x is read first by S and then by S'. When we define a data race in x, input conflicts are not considered. Two segments accessing x cause a data race in x, if they cause an access conflict, and that conflict is of type: flow, anti, or output. Sometimes, for emphasis, we may refer to a flow conflict as a *flow dependence race*, and similarly for the other two types of conflicts: anti and output. (There is no such thing as an *input dependence race*.)

If it is important to focus on the two segments that are involved in a conflict, we may use language like "the segments S and S' *cause* a flow conflict in x," or "there is an output conflict *between* the segments S and S'," and so on.

Theorem 5 can be modified in an obvious way to give a set of sufficient conditions for an access conflict of any given type. For example, we have the following result: Assume that x has been accessed first by a segment S on a thread T, and then by a segment S' on a thread T'. Then these segments cause a flow conflict in x, if

1. $V_S(T) \geq V_{S'}(T)$ and

2. $x \in W_S \cap R_{S'}$.

3. RACE DETECTION WITH LIMITED HISTORY

In this section, we study the practical question of how much can be said (in terms of data races) when the record kept about segments accessing a memory location is less than complete.

Throughout this section, we focus on a given location x in shared memory that is accessed at least once during an execution of the given program. Let there be n accesses to x during this execution, and let $\{S_1, S_2, \ldots, S_n\}$ denote the chronological sequence of segments (of threads in the program) from which these accesses came. Each entry in this sequence corresponds to one access, so that if, for example, the 4th and 5th accesses to x both came from the same segment S, we will have $S_4 = S_5 = S$.

Suppose that at some point during execution, S_j is the current segment that has just accessed x. If we have available now the record of a segment S_i that accessed x previously, then Theorem 5 can be applied to decide if S_i and S_j cause a data race in x. Thus, if we always keep the records of all segments that have accessed x, then we would definitely know whether or not a data race in x exists, and would also be able to find all pairs of segments that cause such races.

Due to space constraints, however, it may not be possible to keep all the segments that have already accessed x at each point during program execution. Then, typically, when we find a segment S_j that has just accessed x, only a few of the segments $S_1, S_2, \ldots, S_{j-1}$ are available for comparison. Since we cannot expect in this situation to be able to capture *all* data races that may be present in x, the important question now is: "When there are data races in x, are we always able to report that at least one such race exists?." The answer, of course, depends on *which* segments from the set $\{S_1, S_2, \ldots, S_{j-1}\}$ are available for comparison with S_j.

It is the algebra of parallel segments that makes a limited record-keeping scheme worth considering. It turns out that even if two segments causing a race are 'far apart' in the sequence $\{S_k\}$, they may force a race between two segments that are often relatively 'close.' Thus, when the goal is not to find all races but to detect if there is at least one race, it is enough to look for races between close pairs of segments. This could be possible by making available at each step only a few segments that have accessed x in the 'recent' past. These vague ideas are made precise below when we describe three different algorithms based on three different record-keeping schemes. The first scheme is very simple, the second extends the first, and the third extends the second.

The following lemma describes a crucial property of the algebra of parallel segments that is used repeatedly in the results and conclusions to follow.

Lemma 6. *Let $\{S_1, S_2, \ldots, S_n\}$ denote the chronological sequence of segments that have accessed a memory location x. Let $i, q_1, q_2, \ldots, q_t, j$ denote integers such that $1 \leq i < q_1 < q_2 < \cdots < q_t < j \leq n$. If the segments S_i and S_j are parallel, then the segments in at least one of the $(t+1)$ pairs:*

$$(S_i, S_{q_1}), (S_{q_1}, S_{q_2}), \ldots, (S_{q_{t-1}}, S_{q_t}), (S_{q_t}, S_j)$$

are parallel.

In particular, if i, q, j denote integers such that $1 \leq i < q < j \leq n$, and the segments S_i and S_j are parallel, then either S_i and S_q are parallel, or S_q and S_j are parallel (or both).

Proof. We prove the special case; the general version follows easily by induction.

Since x is accessed by S_i before it is accessed by S_q, we cannot have $S_q \prec S_i$ (Remark 1.1). Hence, there are two distinct possibilities: either $S_i \preceq S_q$ or $S_i \parallel S_q$. Similarly, for S_q and S_j, there are two distinct possibilities: either $S_q \preceq S_j$ or $S_q \parallel S_j$.

If $S_i \parallel S_q$ and $S_q \parallel S_j$ are both false, then we have $S_i \preceq S_q$ and $S_q \preceq S_j$, so that $S_i \preceq S_j$. This means S_i and S_j cannot be parallel, which is a contradiction. Hence, at least one of $S_i \parallel S_q$ and $S_q \parallel S_j$ is true. ■

3.1 Adjacent Conflict Detection

In this subsection, we study access conflicts.[3] As before, $\{S_1, S_2, \ldots, S_n\}$ denotes the chronological sequence of segments that have accessed a given location x.

An access conflict in x exists between two segments S_i and S_j, where $1 \leq i < j \leq n$, if the segments are parallel. An access conflict between S_i and S_j is called an *adjacent access conflict*, or simply an *adjacent conflict*, if the segments are also consecutive in the sequence $\{S_k\}$, that is, if $i = j - 1$. The terms *adjacent race*, *adjacent flow conflict*, etc. are defined similarly.

The following theorem shows that an access conflict between two segments that could be far apart in $\{S_k\}$ always forces an access conflict between two segments that are next to each other.

Theorem 7. *If there is an access conflict in x, then there must exist at least one adjacent conflict in x.*

Proof. Let S_i and S_j, where $1 \leq i < j \leq n$, denote two segments that cause an access conflict in x. Then, by definition, S_i and S_j are parallel.

If $i = j - 1$, then the access conflict between S_i and S_j is an adjacent conflict, and we have nothing left to prove. So, assume that $i < j - 1$. Taking $t = j - i - 1$ and

$$q_1 = i + 1, q_2 = i + 2, \ldots, q_{j-i-1} = j - 1$$

in Lemma 6, we see that the segments in at least one of the $(j - i)$ pairs:

$$(S_i, S_{i+1}), (S_{i+1}, S_{i+2}), \ldots, (S_{j-2}, S_{j-1}), (S_{j-1}, S_j)$$

are parallel. Since those two consecutive parallel segments cause an adjacent conflict in x, there is an adjacent conflict in x. ■

Note that the above theorem deals with parallel segments only; it does not guarantee that the *type* of the derived adjacent conflict will be the same as that of the original access conflict.

We can now state our most basic algorithm for detecting races in a location x. To apply it we need to keep the information on only one segment: the most recent segment to access x.

Algorithm 2 (Adjacent Conflict Detection Algorithm) This algorithm detects all adjacent conflicts in a location x in shared memory that is accessed by the given program. The requirement is that the information about the last segment to access x is always available.

Whenever a segment S_j is found to access x, compare it with the segment S_{j-1} that was the last segment to access x. If the segments are parallel, then there is an adjacent conflict between them. In this case, note also the type of the conflict. If this is an adjacent conflict of type flow, anti, or output, then it represents a data race in x.

[3]Remember that a race is an access conflict, but an access conflict need not be a race (when it is an input conflict).

If, at the end of program execution, no adjacent conflicts are reported, then there is no access conflict and hence no data race in x. If we report at least one adjacent conflict of type flow, anti, or output, there is a data race. If only adjacent input conflicts are reported, the algorithm is inconclusive.

Remarks 8 This algorithm fails to predict a race in x, when there are races, but no adjacent races (only adjacent input conflicts). To see this clearly, consider the example where the first three segments S_1, S_2, S_3 are such that $S_1 \prec S_2$, $S_2 \parallel S_3$, and $S_1 \parallel S_3$. Assume that S_1 writes x, S_2 reads x, and S_3 reads x. This algorithm will detect the adjacent input conflict between S_2 and S_3, but not the flow dependence race between S_1 and S_3.

We see in the next subsection that by keeping a little more history, we can improve this algorithm to a large extent.

3.2 Local Conflict Detection

In this subsection, we study access conflicts of different types. For each type, we find a result that shows how an access conflict of that type between two arbitrary segments forces an access conflict of the same or a different type, between two segments that are close in the sequence $\{S_k\}$ in a well-defined sense.

In the sequence of segments $\{S_1, S_2, \ldots, S_n\}$ that access x, a member S_k is a *read-segment* if it reads x, or a *write-segment* if it writes x. Consider now any fixed member S_j for $1 < j \leq n$. Segment S_i in the subsequence $\{S_1, S_2, \ldots, S_{j-1}\}$ is the *last-read* segment of S_j, if S_i reads x and the segments $S_{i+1}, S_{i+2}, \ldots, S_{j-1}$, if any, do not. Similarly, S_i is the *last-write* segment of S_j, if S_i writes x and the segments $S_{i+1}, S_{i+2}, \ldots, S_{j-1}$, if any, do not.

An access conflict in x between two segments S_i and S_j, where $1 \leq i < j \leq n$, is a *local access conflict*, or simply a *local conflict*, if S_i is either the last-write or the last-read segment of S_j. The terms *local race*, *local flow conflict*, etc. are defined similarly. Note that S_i is the last-write segment of S_j for local flow and output conflicts, and S_i is the last-read segment of S_j for local anti and input conflicts. An adjacent conflict of any given type is also a local conflict of the same type, since S_{j-1} is always either the last-read or the last-write segment of S_j. The converse, of course, is false.

We present below four theorems that show how an arbitrary access conflict of a given type leads to a local conflict of the same or a different type.

Theorem 9. *If there is an output conflict in x, then there must exist at least one local output conflict in x.*

Proof. Let there be an output conflict in x. Then there are two segments S_i and S_j in the sequence $\{S_k\}$, such that $i < j$, $S_i \parallel S_j$, and both segments write x.

If S_i is the last-write segment of S_j, then the output conflict between them is local, and we have nothing left to prove. So, assume $i < j - 1$ and that some of the segments $S_{i+1}, S_{i+2}, \ldots, S_{j-1}$ write x. Let there be exactly t such segments: $S_{q_1}, S_{q_2}, \ldots, S_{q_t}$, where $1 \leq i < q_1 < q_2 < \cdots < q_t < j \leq n$. Since S_i and S_j are parallel, it follows from Lemma 6 that the segments in at least one of the $(t+1)$ pairs:

$$(S_i, S_{q_1}), (S_{q_1}, S_{q_2}), \ldots, (S_{q_{t-1}}, S_{q_t}), (S_{q_t}, S_j)$$

are parallel. Those two segments cause an output conflict in x and that conflict is local. This completes the proof. ■

Theorem 10. *If there is an input conflict in x, then there must exist at least one local input conflict in x.*

PROOF. The proof of this theorem is exactly similar to the proof of Theorem 9; we just focus on read-segments instead of on write-segments. ■

Theorem 11. *If there is a flow conflict in x, then there must exist at least one local output or one local flow conflict in x.*

PROOF. Let there be a flow conflict in x. Then there are two segments S_i and S_j in the sequence $\{S_k\}$, such that $i < j$, $S_i \parallel S_j$, S_i writes x, and S_j reads x. After this point, we proceed exactly as in the proof of Theorem 9. The only difference now is that while the segments $S_i, S_{q_1}, S_{q_2}, \ldots, S_{q_t}$ write x, the segment S_j reads it. So, if it turns out that S_{q_t} and S_j are parallel, then the conflict between them would be a local flow conflict. If the segments in any of the first t pairs are parallel, they will represent a local output conflict. ■

Theorem 12. *If there is an anti conflict in x, then there must exist at least one local input conflict or one local anti conflict in x.*

PROOF. The proof is similar to the proofs of the previous theorems and is omitted. ■

Remarks 13

1. We can combine theorems 9 and 11 to say that a race of type output or flow between two arbitrary segments will always force a local conflict of type output or flow. So, in order to detect the possible existence of an output or flow dependence race, it is sufficient to check for all local output and flow conflicts.

2. The case for an anti dependence race is more complicated. Such a race may not produce a local conflict of type output, flow, or anti. If we see a local anti conflict, then we have detected a race. But, if we see a local input conflict, then it may or may not indicate an anti conflict. To see this better, consider the example where the first three segments to access x are as follows: $S_1 \parallel S_2, S_2 \prec S_3, S_1 \parallel S_3$, S_1 and S_2 read x, and S_3 writes x. (See Figure 2.) There is an anti dependence race here between S_1 and S_3, but it is not a local race (since S_1 is not the last-read segment of S_3). There is a local input conflict between S_1 and S_2, but that does not qualify to be a race.

We state next our second algorithm for detecting races in a location x. To apply it we need to keep the information on only two segments: the last segment to read x and the last segment to write x.

Algorithm 3 (Local Conflict Detection Algorithm) This algorithm detects all local conflicts in a location x in shared memory that is accessed by the given program. The requirement is that the information about the last segment to read x and the last segment to write x is always available.

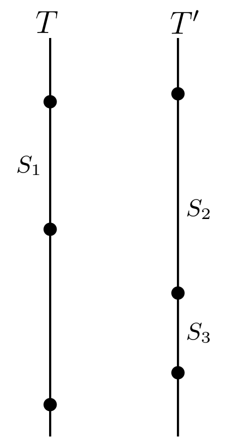

Figure 2: S_1 and S_2 read x, S_3 writes x; anti conflict in x, also a local input conflict, but no local race.

Whenever a segment S_j is found to access x, compare it first with its last-write segment to see if the segments cause a local output or flow conflict. Then, compare S_j with its last-read segment to see if the segments cause a local anti or input conflict.

If, at the end of program execution, no local conflicts are reported, there is no data race in x. If at least one local conflict of type flow, anti, or output is reported, there is a data race. If only local input conflicts are reported, then the algorithm is inconclusive, since a discovered local input conflict may or may not signal an anti conflict between another pair of segments.

Remarks 14 This algorithm fails to predict a race in x, when there are races, but no output conflicts of any kind, no flow conflicts of any kind, and no local anti conflicts. This algorithm will detect the flow dependence race between S_1 and S_3 in the example of Remark 8 (that Algorithm 2 failed to detect), but will not detect the anti dependence race between S_1 and S_3 in the example of Remark 13.2.

The current version of the Intel Thread Checker uses the Local Conflict Detection Algorithm.

3.3 Data Race Detection

In order to remedy the deficiency of the Local Conflict Detection Algorithm, we need to explore what, if any, clues are given by an anti conflict when it does not force a local race. This leads us to define a special class of anti conflicts that are more general than adjacent conflicts. An anti conflict in x between two segments S_i and S_j, where $1 \leq i < j \leq n$, is *near-adjacent*, if for each k in $i < k < j$, the segment S_k reads x and is parallel to S_i. An adjacent anti conflict is clearly near-adjacent, since this condition is then satisfied vacuously.[4]

Theorem 15. *If there is an anti conflict in x, then there must exist at least one local output conflict or one near-adjacent anti conflict in x.*

PROOF. Let there be an anti conflict in x. Then there are two segments S_p and S_r, such that $1 \leq p < r \leq n$, $S_p \parallel S_r$, S_p reads x, and S_r writes x. We need to find two segments

[4]A *local* anti-conflict may not be near-adjacent.

S_i and S_j with similar properties, such that the segments S_k between them, if any, read x and are parallel to S_i.

First, assume that there are t segments $S_{q_1}, S_{q_2}, \ldots, S_{q_t}$ between S_p and S_r that write x, where $1 \le p < q_1 < q_2 < \cdots < q_t < j \le n$. (If there are no such segments, then take $q_1 = r$ and skip to the next paragraph.) Since S_p and S_r are parallel, it follows from Lemma 6 that the segments in at least one of the $(t+1)$ pairs:

$$(S_p, S_{q_1}), (S_{q_1}, S_{q_2}), \ldots, (S_{q_{t-1}}, S_{q_t}), (S_{q_t}, S_r)$$

are parallel. If the segments in one of the last t pairs are parallel, we have a local output conflict, and we are done. So, assume that the segments S_p and S_{q_1} in the first pair are parallel.

Let $j = q_1$. Then, $S_p \parallel S_j$ and the segments $S_p, S_{p+1}, \ldots, S_{j-1}$ all read x. Let i denote the largest integer such that $p \le i < j$ and $S_i \parallel S_j$. Such an integer always exists, since $S_p \parallel S_j$. It is clear that there is an anti conflict between the segments S_i and S_j. We claim that this anti conflict is near-adjacent. If $i = j-1$, then this is an adjacent anti conflict, and we are done. So, assume $i < j-1$. Take any k such that $i < k < j$. Since $S_i \parallel S_j$, it follows from Lemma 6 that either $S_i \parallel S_k$, or $S_k \parallel S_j$. But, due to the way i was chosen, S_k cannot be parallel to S_j. Hence, S_i is parallel to S_k. Since S_k obviously reads x, this completes the proof. ■

Since S_i is parallel to each of the segments $S_{i+1}, S_{i+2}, \ldots, S_{j-1}$, to make sure that we have S_i available for comparison with S_j, it suffices to keep all read-segments parallel to the last-read segment (between two consecutive write-segments).

Theorems 9, 11, and 15 show that if there is a race in x (of any kind) between any pair of segments, then there must also be either a local output conflict, or a local flow conflict, or a near-adjacent anti conflict. The third algorithm of Section 3 is the *Race Detection Algorithm.* It is designed to detect all local output and local flow conflicts in x, and also all near-adjacent anti conflicts. In this algorithm, we always keep the last segment to write x, the last segment to read x, and the set of all read-segments parallel to the last-read segment *since* the last-write segment. Whenever a segment S_j is found to access x, we compare it with the available segments to determine if there is a local output conflict, or a local flow conflict, or any near-adjacent anti conflicts.

This algorithm applied to a given location x in shared memory will always accurately predict whether or not there is at least one data race in x.

Algorithm 4 (Race Detection Algorithm) This algorithm is for a location x in shared memory that is accessed by the given program. It finds all conflicts of the following types: local output, local flow, and near-adjacent anti. It can always detect if there is a data race in x. (It may not, however, find *all* races.) At any point during execution of the program, let S^w denote the last segment to write x and S^r the last segment to read x after S^w, if any. Let $\mathcal{R}$ denote a set that consists of S^r and each read-segment between S^w and S^r, that is parallel to S^r. We keep track of the segment S^w and the set $\mathcal{R}$, but not the segment S^r. Initially, S^w is undefined and $\mathcal{R}$ is empty. Treat each notation of the form $S' \parallel S$ used below as an abbreviation of the clause: *S' is defined and parallel to S.*

```
Repeat until program execution comes to an end:
  $S \leftarrow$ the next segment to access $x$;
  If $S$ writes $x$
  then
     if $S^w \parallel S$
     then report a local output conflict in $x$,
     if $\mathcal{R} \neq \emptyset$
     then
        for each $S' \in \mathcal{R}$ such that $S' \parallel S$
           report a near-adjacent anti conflict in $x$,
        set $\mathcal{R} \leftarrow \emptyset$,
     set $S^w \leftarrow S$;
  else (i.e., if $S$ reads $x$)
     if $S^w \parallel S$
     then report a local flow conflict in $x$,
     delete each member of $\mathcal{R}$ that is not parallel to $S$,
     put $S$ in $\mathcal{R}$.
If at least one conflict of type output, flow, or anti has
been reported, then
  report that there is a data race in $x$,
else
  report that there is no data race in $x$.
```

Remarks 16

1. This algorithm will detect the anti dependence race between the segments S_1 and S_3 in the example of Remark 13.2 (that Algorithm 3 failed to detect).

2. When the set $\mathcal{R}$ has more than one member, any two distinct members are parallel.

3. The size of $\mathcal{R}$ cannot exceed the number of mutually parallel read-segments between two consecutive write-segments.

4. If we keep fewer than the read segments required in this algorithm, then we may not always be able to report a race when there is one. Consider the following example:

   Assume that the first 99 segments $S_1, S_2, \ldots, S_{99}$ to access x are such that every two segments in the sequence are parallel and each segment reads x. Now, suppose x is written by segment S_{100}, such that the segment S_1 is parallel to S_{100} and each of the segments $S_2, S_3, \ldots, S_{99}$ precedes S_{100}. There is only one race here: the anti-dependence race between S_1 and S_{100}. To capture this race, we need to have S_1 available when S_{100} is found to access x. So, we cannot afford to drop S_1 as long as it is parallel to the last-read segment at each step (since the last-write segment). The number 100 chosen here is arbitrary.

3.4 Current Conflict Detection

In this section, we describe the race detection algorithm published in [3, 8]. The description is given in the framework we have developed in this paper, so that this algorithm can be easily compared with the ones we have already presented.

The previous three algorithms in Section 3 have been described in terms of the chronological sequence of segments $S_1, S_2, \ldots, S_n$ that access a given memory location x in an execution of the program. Adjacent, local, and near-adjacent conflicts were defined in a temporal sense in terms of this sequence. The threads to which these segments belong have been implicit. Now we describe an algorithm where we always keep the last segment to write x and the last segment to read x for *each* active thread. Like Algorithm 4, this algorithm is able to detect at least one race if there are races. The storage requirements are different however; we comment on that issue in Remark 18 below. First, we need some definitions and results.

We say that there is a *current* flow conflict in x between two segments S_i and S_j, where $1 \leq i < j \leq n$, if at some given point, S_i is the last segment to write x on its thread, S_j is the last segment to read x on its thread, and $S_i \parallel S_j$. Current conflicts of the other three types are defined in a similar way. A *current* race in x is a current conflict of type flow, anti, or output. Our algorithm in this subsection is based on the following theorem.

Theorem 17. *If there is an access conflict in x of a given type, then there must be a current conflict in x of the same type.*

Proof. The particular type of conflict chosen is irrelevant here. For definiteness, assume that there is a flow conflict in x between two segments S_i and S_j such that $1 \leq i < j \leq n$. Focus on the moment when S_j becomes the last segment to read x on its thread. By hypothesis, S_i has already written x. If it is currently the last segment to write x on its thread, then we have a current flow conflict, and there is nothing left to prove.

Assume then that S_i is not currently the last segment to write x on its thread. Let S_q denote the last segment to write x on the thread of S_i, when S_j is the last segment to read x on the thread of S_j. Note that we have $i < q < j$. By hypothesis, S_i and S_j are parallel. Lemma 6 then implies that either the segments S_i and S_q are parallel, or the segments S_q and S_j are parallel. The first possibility is ruled out since S_i and S_q belong to the same thread. Hence, S_q and S_j are parallel. These two segments cause a current conflict in x. ■

Corollary 2. *If there is a race in x, then there must be a current race in x.*

Proof. A race is an access conflict of type flow, anti, or output. If there is an access conflict of one of these three types, then there must also be a current conflict of the *same* type which is a current race. ■

Algorithm 5 (Current Conflict Detection Algorithm) This algorithm detects all current conflicts in a location x in shared memory that is accessed by the given program. It can always detect if there is a data race in x, although it may not find *all* races. The requirement is that at any given point and for each active thread T, the information about the last segment on T to read x and the last segment on T to write x are always available.

Whenever a segment S_j is found to access x, compare it first with the last-write segment of each thread to see if the segments cause a current output or flow conflict. Then, compare S_j with the last-read segment of each thread to see if the segments cause a current anti or input conflict.

If, at the end of program execution, we report at least one current conflict of type flow, anti, or output, then there is a data race in x. Otherwise, there is no data race in x.

Remark 18 Algorithms 4 and 5 have similar capabilities: Each will predict at least one race if there are races, and each will say there are no races when there are none. For a given memory location x, Algorithm 5 needs to keep $2t$ segments at each step where t is the number of active threads: 2 segments for each thread. Algorithm 4, on the other hand, needs to keep only $(p+1)$ segments: 1 write-segment and p read-segments that are mutually parallel. Thus, Algorithm 5 needs to keep $(2t-p-1)$ more segments than Algorithm 4. Since these p read-segments are mutually parallel, they must come from different threads, so that $t \geq p$. Hence, we have $(2t-p-1) \geq t-1$. This difference is huge when there are a large number of threads.

4. PREVIOUS WORK

The fundamental concept used in this paper is that of the precedence relationship '$\prec$' between segments. This is the *happens-before* relation '$\rightarrow$' introduced by Lamport in [4]. He also gives the definition of parallel (concurrent) segments. Logical clocks defined by Lamport in the same paper do not, however, quite capture the parallel relationship '$\parallel$' between segments. For that we had to use vector clocks.

Vector clocks have been used empirically by several researchers to solve specific problems since the early 1980's. Fidge [2] and Mattern [5] are generally credited for introducing them as a formal concept and stating their basic properties in 1988, independently of each other.

Our race detection method (based on vector clocks) is similar to the one described in [3], [7], [8], and [9]. A hybrid detection method is described in [7] that combines two techniques one of which is based on the *happens-before* relation. Unlike the approach taken in [9], we do not use two passes, and we keep the values of vector clocks relatively low by counting only the *posting* segments that precede a given segment.

The algorithmic approach in [3] and [8] is described in detail in Section 3.4, rephrased in our framework to allow direct comparison with the other algorithms in this paper. Our Algorithm 4 and Algorithm 5 of [3, 8] both solve the entire problem of detecting at least one data race when races are present. Neither one will report a race when there are none. However, the storage requirements of the two are different. Let t denote the total number of threads and N the total number of memory locations accessed. Then, Algorithm 4 needs to keep $(1+p)N$ segments, where $p \leq t$, and Algorithm 5 needs to keep $2tN$ segments.

Algorithm 3 (used by the Intel Thread Checker) may sometimes fail to detect a race in rare situations, but it needs to keep only $2N$ segments. An experimental discussion of how rare these situations are in real programs is beyond the scope of the present paper.

5. CONCLUSIONS

We have provided a solid mathematical foundation for the theory of data race detection. All four race detection algorithms can be used in a practical situation with different goals in mind. The Intel Thread Checker uses Algorithm 3. Practical applications of this algorithm have been possible due to the way it handles the tradeoff between the ability to always detect data races and the need to conserve memory usage. By keeping the history of only two previous accesses to a memory location, the Thread Checker manages to detect the existence of a data race in a vast majority of situations. Algorithm 4 or Algorithm 5 can be used to gain further improvement. Either one is capable of detecting at least one data race if races are present. Also, they will not report a race if there are none. While the two algorithms have similar capabilities, our Algorithm 4 needs less storage than the previously published Algorithm 5.

6. REFERENCES

[1] U. Banerjee, B. Bliss, Z. Ma, and P. Petersen. Unraveling Data Race Detection in the Intel® Thread Checker. Presented at The First Workshop on Software Tools for Multi-core Systems (STMCS), in conjunction with IEEE/ACM International Symposium on Code Generation and Optimization (CGO), March 26, 2006, Manhattan, New York, NY.

[2] C. J. Fidge. Timestamps in Message-Passing Systems that Preserve the Partial Ordering. *Proceedings of the 11th Australian Computer Science Conference (ACSC'88),* K. Raymond (Editor), February 1988, pp. 56–66.

[3] A. Itzkovitz, A. Schuster, and O. Zeev-Ben-Mordehai. Towards Integration of Data Race Detection in DSM Systems. *Journal of Parallel and Distributed Computing,* Vol. 59, No. 2, November 1999, pp. 180–203.

[4] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM,* Vol. 21, No. 7, July 1978, pp. 558–565.

[5] F. Mattern. Virtual Time and Global States of Distributed Systems. *Proceedings of the International Workshop on Parallel and Distributed Algorithms,* Elsevier Science Publishers, M. Corsnard et al. (Editors), Chateau de Bonas, France, October 1988, pp. 215–226.

[6] R. H. B. Netzer and B. P. Miller. What are Race Conditions? Some Issues and Formalizations. *ACM Letters on Programming Languages and Systems,* Vol. 1, No. 1, March 1992, pp. 74–88.

[7] R. O'Callahan and J.-D. Choi. Hybrid Dynamic Data Race Detection. *Proceedings of the 2003 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,* ACM Press, San Diego, California, June 2003, pp. 167–178.

[8] E. Pozniansky and A. Schuster. Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs. *Proceedings of the 2003 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,* ACM Press, San Diego, California, June 2003, pp. 179–190.

[9] M. Ronsse and K. de Bosschere. RecPlay: a Fully Integrated Practical Record/Replay System. *ACM Transactions on Computer Systems,* Vol. 17, No. 2, May 1999, pp. 1–17.

[10] O. Shacham, M. Sagiv, and A. Schuster. Scaling Model Checking of Data Races Using Dynamic Information. *Proceedings of the 2005 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,* ACM Press, Chicago, Illinois, June 2005, pp. 107–118.