Unraveling Data Race Detection in the Intel® Thread Checker

Utpal Banerjee Brian Bliss Zhiqiang Ma Paul Petersen
{utpal.banerjee, brian.e.bliss, zhiqiang.ma, paul.petersen}@intel.com
Software and Solutions Group
Intel Corporation

ABSTRACT

The Intel® Thread Checker is a runtime analysis tool for automatically finding threading related defects in multithreading applications. An unsynchronized memory reference which causes non-deterministic behavior – a problem that is notoriously hard to find – is one such defect which the Thread Checker finds with ease. These data races are pinpointed for the user at specific locations in the application's source code.

Modern threading APIs (such as Windows or POSIX threading APIs) have a rich set of threading and synchronization constructs which allow for complex interactions between threads – more than simply locks protecting a critical section. The Thread Checker uses an analysis algorithm which allows the transitive effects of these APIs to be reflected.

This paper will use a motivating example to illustrate the operation of the Intel® Thread Checker. The usage model, instrumentation approach, and runtime analysis algorithm will be presented.

1 INTRODUCTION

The ability to observe a problem is the first step towards resolving the problem. To sequential programs, it is possible to make arbitrarily detailed observations. Developers can use interactive breakpoint debuggers to observe program behaviors by single stepping the program execution, by examining the contents of the registers, by inspecting data structures to see if they are still in a consistent state. This is possible because the memory used by the program is the state of the computation. This state can be inspected and modified. By controlling the execution the developers control how the state to be modified.

Developers of threaded programs also are expected to use breakpoint debuggers to solve the problems encountered in threaded applications. This works up to a point. The debuggers do allow the current state of the program to be inspected and allow some control over moving to a new state by single stepping the execution. However, the most difficult problems to diagnose are not easily solved through the use of a breakpoint debugger.

Concurrency introduces non-determinism into the execution of the program by having a set of threads that can be chosen to be executed at any point in time. On a uni-processor the operating system decides which of these threads to execute. On a multi-processor, the operating system may choose several of these threads to execute at the same time. If the instructions executed by each thread don't share any of the same objects in use by the program, then this non-determinism can not introduce visible side-effects to indicate that they were not executed in some sequential order. But this is rarely the case. The use of threads almost always involves the threads cooperating for some purpose, sharing resources and object references. Once a potential exists for simultaneous access to the same object or resource correct control access to this resource is needed in order to make sure the intended computation occurs.

The Intel® Thread Checker (referred to in this paper as the Thread Checker) is designed to observe the execution of a program and to inform the user of places in the application where problems may exist. The problems detected are specific to the use of threads. These include incorrect use of the threading and synchronization API functions (for example passing the wrong type of synchronization object handle to a synchronization API). The use of synchronization APIs can be correct in a specific instance but incorrect in aggregate. For example, deadlocks can occur when synchronization resources are acquired in inconsistent orders by different threads. Finally, the problem may be the lack of synchronization. This manifests itself as a data-race. A data-race occurs when two threads attempt to access the same object, where one of the threads is attempting to modify the object, and the order of these two

accesses is not controlled by the synchronization logic implemented in the program.

1.1 The Problem

Let's discuss data-races in a bit more depth. Are data-races good or bad? It depends on how they affect the computation. If a thread attempts to read the value of a computation before the computation has generated a result, it will either read an un-initialized or obsolete value. If two threads are attempting to accumulate partial results into a finally result, the result will not be correct if the result of one thread overwrites the result of the other thread rather than merging the two results. If a thread attempts to use an object before it is fully initialized it may cause the program to abort.

However in some circumstances, a data-race may be benign and a natural part of an algorithm. A common case is redundant initialization, or redundant signaling. If multiple threads all write the same value, then the order of the writes is not significant to an observing thread. This assumes that the memory system can guarantee the atomic modification of the value so that intermediate states are not observable.

A class of algorithm in which data-races can be benign is with memory based synchronization. A common pattern is the "double-check" pattern. This pattern is similar to the redundant initialization pattern described above, except that a flag is speculatively checked to see if the initialization has finished and if not, then a lock is acquired. Once the lock is acquired, then the flag is checked again to ensure that an earlier thread did not initialize the object. Finally the object is initialized, the flag is set, and the lock is released. If the "double-check" pattern is correctly implemented, a benign data-race exists between the check of the flag outside of the critical section, and the modification of the flag inside the critical section. The reason this works is that this modification of the flag is atomic (guaranteed by the memory system) from the observers perspective, and the sequence of modifications is monotonic. Since the sequence is monotonic, all previous states can be inferred by observing the current state. Once the object has been initialized it can never become uninitialized, with the speculation being that two threads may potentially see it as uninitialized, this conflict is resolved through the use of the critical section to validate if the first speculative read was correct.

Throughout this paper we will assume that the user of the Thread Checker should be informed of all data-races detected in the application. The user can then inspect each data-race to determine if each instance can cause a defect in the execution of the application, or if the data-race is benign, and is an expected part of the algorithm used by the application.

1.2 Organization

The remainder of this paper explores the implementation of the Thread Checker by showing how it handles specific situations which rise from the instrumentation and analysis of the example used by this paper. In section 2, we introduce the example to be used throughout the paper and show what the user sees as the result of the Thread Checker analysis. Also section 2 dives into the implementation of the Thread Checker to see how the major components are organized. In section 3, we start with the first phase: Instrumentation. The instrumentation is the core feature that allows the programs behavior to be observed. Observing thread creation and destruction, thread synchronization, and memory accesses collects information for the data-race analysis. Section 4 progresses on to explain the analysis which is performed on the information generated by the execution of the instrumented application. Details include a short background on the theoretical basis of the analysis, and continuing on to describe the central algorithm used to detect data-races. As the paper progresses, we move to Section 5 which discusses consequences of the design used in the Thread Checker, and shows the significance of this problem area through a short tour of alternate solutions. Finally, we reach the conclusion of the paper in Section 6, where we recap what we have learned through this exposition. We hope that this short walk through the implementation of the Thread Checker will prove to be interesting and entertaining.

2 EXAMPLE AND OVERVIEW OF INTEL® THREAD CHECKER

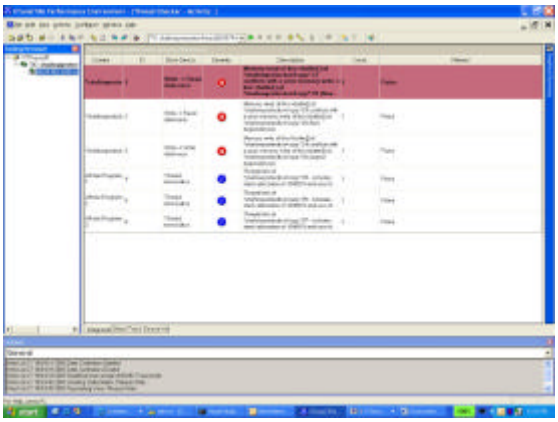
In this section, we will first use an example to show what the Thread Checker can do and then give a brief overview of how it works.

2.1 An Example

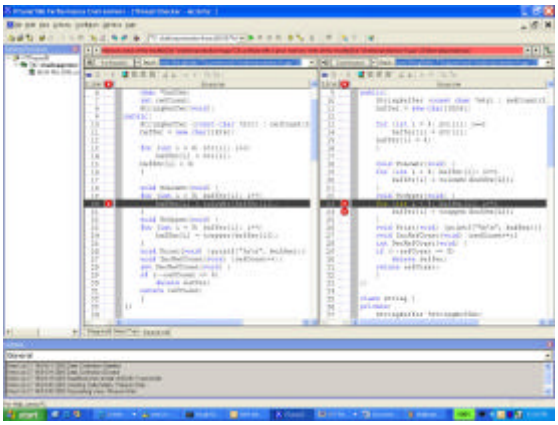
The example shown in the appendix presents a simple imperfect C++ string implementation.

The author of the example intended to implement a copy-on-write string. There are 2 classes in this implementation: `String` and `StringBuffer`. The `StringBuffer` implements a character array as a back store, which can be shared among different strings. The `String` is a string type making use of the `StringBuffer`. The idea is that different strings can share the same `StringBuffer` and that copying from one `StringBuffer` to another `StringBuffer` should be performed only if a string is to be written.

Unfortunately, the implementation is not quite correct. We use the Thread Checker to analyze the example and get the following report.



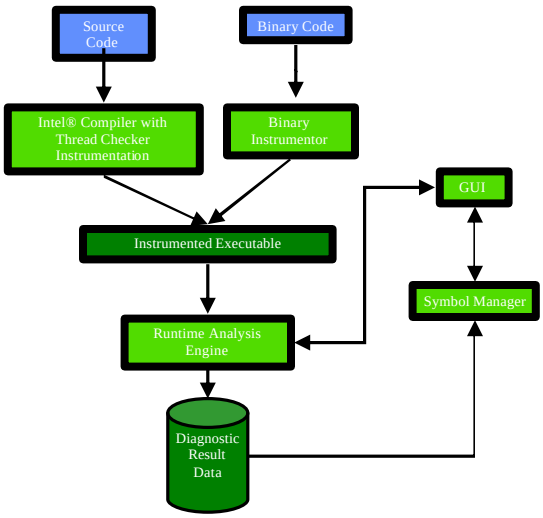
Thread Checker reports three data race diagnostics. If we highlight the first diagnostic and click the "Source View" tab, we will get:



The first diagnostic reports that while one thread was converting the `StringBuffer` to lowercase at line 20, another thread was reading the same `StringBuffer` at line 23. The second and third diagnostics report that while one thread was converting the `StringBuffer` to lowercase at line 20, another thread was converting the same `StringBuffer` to uppercase at line 24. Apparently, the copy-on-write is not correctly implemented.

2.2 Overview of Intel® Thread Checker

Following is an architectural overview of the Intel® Thread Checker.



The Thread Checker begins by instrumenting either source code or binary code of the program. It instruments every memory reference instruction and every thread synchronization primitive in the program. When the instrumented program is executed, the runtime analysis engine monitors every memory reference event and every thread synchronization event and analyzes if there is a data races. It writes the diagnostic results to a file and to the GUI for real-time display. The GUI asks the symbol manager to translate any address in the diagnostics to human readable symbolic symbols and source line information and present the resolved diagnostics to the user.

3 INSTRUMENTATION

In order to analyze a multithread program, the Thread Checker needs to observe the runtime behavior of each thread in the program. Instrumentation is the process of inserting extra code in the original program for this purpose. During runtime when the instrumented program is executed, the extra code inserted will pass what it has "observed" to the runtime analysis engine for analysis.

A thread is a sequence of instructions that can run independently. From a programmer's perspective, two kinds of instructions, memory instructions and synchronization instructions, are of interest to us. In the example,

```
buffer[i] = 0;
--RefCount;
```

are both memory instructions and

```
CreateThread(0,0,ConvertToLower,…);
WaitForSingleObject(lock, INFINITE);
ReleaseMutex(lock);
```

are all synchronization instructions.

To watch the runtime behavior of each thread, we need to instrument both memory instructions and synchronous instructions.

Though compiler instrumentation and binary instrumentation are implemented differently, they both follow the same principles. Unless it is explicitly pointed out, the following discussion is applicable to both instrumentation schemes.

3.1 Memory Instructions

The Thread Checker instruments every memory instruction. To each memory instruction, extra code is inserted to catch a memory access record consisting of:

(1) the memory access type: read, write or update
(2) the relative virtual instruction address
(3) the virtual memory address the instruction performs at and
(4) the size of the memory the instruction performs on.

In the example,

```
buffer[i] = 0;
```

is a memory instruction, The instrumented pseudo code will look like (assume the relative virtual instruction address is 100000):

```
MemAccessType = WRITE;
VirtualInstrAddress = 100000;
VirtualMemAddress = &(buffer[i]);
MemSize = 4;
Call AnalyzeMemInstr(
       MemAccessType,
       VirtualInstrAddress,
       VirtualMemAddress, MemSize);
buffer[i] = 0;
```

When the above code is executed, the memory access type, the relative virtual instruction address, the virtual memory address and the memory size are collected and passed to the runtime analysis engine by calling routine `AnalyzeMemInstr()`.

Memory instruction instrumentation is very expensive, To reduce the cost, we buffer as many access records as possible and we allow one invocation of `AnalyzeMemInstr()` to pass multiple access records to the runtime analysis engine. To further reduce the cost, we allow the user to turn off memory instruction instrumentation completely in a function, a file (if compiler instrumentation is used) or a module.

3.2 Synchronization Instructions

In order to correctly analyze the whole program, the observation and analysis order of the synchronization instructions by the runtime analysis engine has to match the execution order of the same instructions (see section 4 for explanation). If a thread *T* executes `ReleaseMutex()` releasing a mutex first and a thread *T'* executes `WaitForSingleObject()` acquiring the same mutex later, the runtime analysis engine needs to observe and analyze the call to the `ReleaseMutex()` function before the call to the `WaitForSingleObject()` function call.

On the other hand, a synchronization instruction is usually in fact a threading API or a function call to the underlying threading library or to the operating system, though it can be viewed as a single instruction from a programmer's perspective. So for simplicity we instrument synchronization instructions by function replacement instead of simply inserting extra code in the original program.

For each threading API which is deemed as a synchronization instruction from a programmer's perspective, the Thread Checker replaces it with a different function which guarantees the observation and analysis order is the same as the execution order of the replaced API.

In the example, function

```
void Lock(void) {
   WaitForSingleObject(lock, INFINITE);
}
```

executes a synchronization instruction or calls a threading API WaitForSingleObject();

The Thread Checker replaces WaitForSingleObject(); call with a different call provided by the runtime analysis engine. The instrumented pseudo code will look like:

```
void Lock(void) {
   ExecuteAndAnalyzeWaitForSingleObject(
     lock, INFINITE);
}
```

This new function and WaitForSingleObject() have the same signature. It is provided by the runtime analysis engine

```
ExecuteAndAnalyzeWaitForSingleObject(
  HANDLE hHandle, DWORD dwMilliseconds) {

    PreAnalyzeWaitForSingleObject();

    // critical section begins
    WaitForSingleObject(
      hHandle, dwMilliseconds);
    PostAnalyzeWaitForSingleObject();
    // critical section ends
}
```

In this new function, execution of the threading API `WaitForSingleObject()` and the routine defined by the runtime analysis module `PostAnalyzeWaitForSingleObject()`

are in a critical section such that we can guarantee that the execution order is the same as the analysis order of the synchronization instruction.

3.3 Other Instructions

Besides memory instructions and synchronization instructions, there can be other instructions, for example, control transfer instruction in a thread. The Thread Checker also instruments some of those instructions. For example, the Thread Checker instrument function call and return instructions for keeping a call stack of a thread during runtime. Discussion these instructions and their instrumentations are out of the scope of this paper.

4 ANALYSIS

We introduce how Thread Checker analysis engine works in this section. We start by defining a few terms.

4.1 Segment and Partial Order Graph

Synchronization instructions partition a thread into segments. A segment of a thread is a maximal subsequence of instructions ending with a synchronization instruction. In the previous example, the following code is a segment of the first child thread.

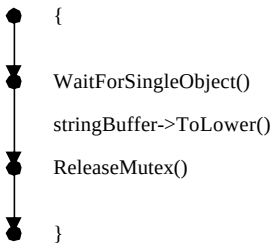
```
stringBuffer->ToLower();
Unlock();
```

A segment may or may not contain memory instructions.

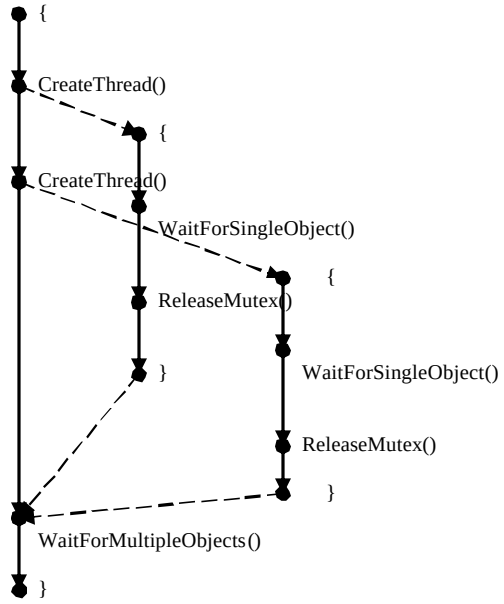
We further classify synchronization instructions into posting synchronization instructions and receiving synchronization instructions. A synchronization instruction is a posting synchronization instruction if it posts the occurrence of an event or condition to some other thread or itself. A synchronization instruction is a receiving synchronization instruction if it reads the occurrence of an event or condition posted before. In our example, `ReleaseMutex()` is a posting synchronization instruction posting that the mutex is now released and available for acquisition by another thread. `WaitForSingleObject()` is a receiving synchronization instruction reading the occurrence of the mutex releasing posted before by the releasing thread. Accordingly, we define a segment ended by a posting synchronization instruction to be a posting segment and a segment ended by a receiving synchronization instruction to be a receiving segment. For example, the segment listed above is a posting segment.

A thread starts from a receiving synchronization instruction reading the occurrence of its being created by another thread executing posting synchronization instruction `CreateThread()` or by the operating system executing a `CreateThread()`-equivalent posting synchronization instruction. A thread ends with a posting synchronization instruction posting the occurrence of its death for another thread to read by executing a receiving synchronization instruction. Accordingly, a thread starts with a receiving segment containing no memory instructions and ends with a posting segment.

We use a sequence of arrowed line segments connected by dots to represent a thread's execution. Each arrowed solid line segment represents a segment of the thread and each dot represents a synchronization instruction. For example, the first child thread can be represented by:



We capture the fact that a posting synchronization instruction must execute before a corresponding receiving synchronization instruction by drawing a dotted arrowed line segment from the dot representing the posting synchronization instruction to the dot representing the receiving synchronization instruction. By doing so, we have a directed acyclic graph representing the whole program execution.



We can see from the graph that segments of different threads execute in partial orders even though segments

of a particular thread execute sequentially in a definitive order, from the first segment to the last segment. The directed acyclic graph is a partial order graph.

Theorem 1: Let S_1 and S_2 denote 2 different segments in the partial order graph. S_1 definitively executes before S_2 if and only if there exists a path from the starting of S_1 to the ending of S_2 and both S_1 and S_2 are on the path.

Proof: omitted.

We define $S_1 \prec S_2$ if S_1 definitively executes before S_2.

We say S_1 and S_2 are parallel or $S_1 \parallel S_2$ if and only if neither $S_1 \prec S_2$ nor $S_2 \prec S_1$.

For any 2 different segments S_1 and S_2 in the graph, we have one of:

(1) $S_1 \prec S_2$;
(2) $S_2 \prec S_1$;
(3) $S_1 \parallel S_2$.

4.2 Vector Clocks

Given the partial order graph, we are able to determine the ordering of any two segments by finding if there is a path from one segment to the other. However, the partial order graph of a program can be too large for practice and finding a path from one segment to another is not cheap. Vector clocks provide a simple numerical method for computing the ordering of any two segments.

A vector clock Vs of segment S is an integer vector associated with segment S. Given the set of the threads T in the partial order graph, for each $T \in T$, we define

$V_S(T)$ = [number of posting segments S' on T such that $S' \prec S$]

Theorem 2: Let S denote a segment on thread T and S' a segment on a different thread T'. $S \prec S'$ iff $V_S(T) < V_{S'}(T)$.

Proof: omitted.

4.2.1 Propagating Vector Clocks

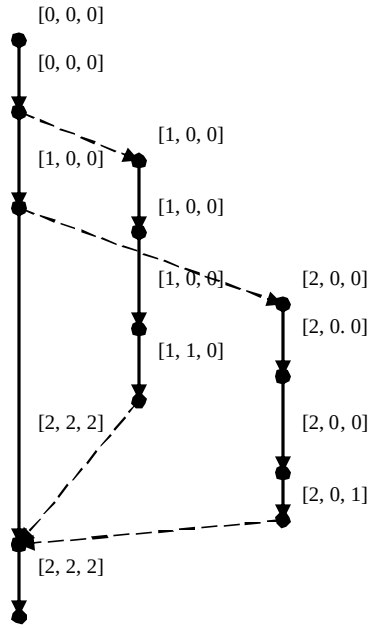
When a thread executes a synchronization instruction, it does so on a synchronization object. The mutex `String::lock` in our example is a mutex synchronization object and both `WaitForSingleObject()` and `ReleaseMutex()` perform on it. When a thread executes `ReleaseMutex()`, it changes the state of `String::lock`. When a second thread later executes `WaitForSingleObject()`, it will see the state change made by the first thread. What this means is that a synchronization object can be used as a carrier to propagate vector clocks from one thread to another thread for vector clock calculation.

For each synchronization object such as `String::lock` in the example, the Thread Checker keeps a vector clock K_O for vector clock propagation purpose.

A thread is also a synchronization object. Both the `CreateThread()` posting synchronization instruction executed by the creating thread and the receiving synchronization instruction starting the created thread perform on the synchronization object of the creating thread. On the other hand, the posting synchronization instruction ending a thread performs on the synchronization object of the thread being ended.

Whenever a thread T executes a posting synchronization instruction on a synchronization object O ending the current segment S_i and starting a new segment $S_{(i+1)}$, Thread Checker first calculates the vector clock $V_{S_{(i+1)}}$ from the vector clock V_{S_i}, then it passes the value of $V_{S_{(i+1)}}$ to K_O. Later, when a thread T' executes a receiving synchronization instruction on synchronization object O ending the current segment S'_j and starting a new segment $S'_{(j+1)}$, Thread Checker calculates the vector clock $V_{S'_{(j+1)}}$ from the vector clock $V_{S'_j}$ and the vector clock K_O.

Following is the partial order graph of the example annotated with vector clock of each segment.



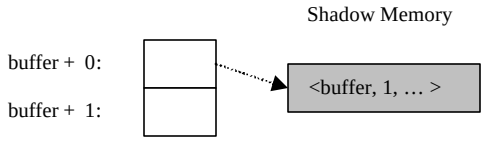
4.3 Shadow Memory

Normally, when a thread accesses a memory location, it accesses a region of memory. For example, if a thread accesses an integer at memory location *x*, it accesses a memory region of 4 bytes on an IA-32 platform starting at address *x*. We introduce the notation *M(x, z)* to represent a region of memory with size *z* starting at address *x*.

In the Thread Checker, all of the shared memory regions are shadowed. The shadow memory of a shared memory region is a collection of shadow memory cells. Each shadow memory cell shadows a sub-region of the shared memory region.

A shadow memory cell is an 8-element tuple $<m, z, t_r, c_r, l_r, t_w, c_w, l_w>$, in which *m* is the starting address of a sub-region, *z* is the size of the sub-region, t_r is the identifier of thread most recently reading *M(m, z)*, c_r is the $V_S(t_r)$ when t_r most recently read *M(m, z)* in segment *S*, l_r the location of instruction executed by thread t_r most recently reading *M(m, z)*, t_w the identifier of the thread most recently writing *M(m, z)*, c_w is $V_{S'}(t_w)$ when t_w most recently wrote *M(m, z)* in segment *S'*, and l_w is the location of instruction executed by thread t_w most recently writing *M(m, z)*.

A representation of the shadow memory cell of memory region *M(buffer, 1)* is shown below.



4.4 Detecting Data Races

Now we are ready to discuss how the Thread Checker detects data races. We start from a theorem.

Theorem 3: Let *x* denote a shared memory location. 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'*. There is a data race on *x* if $V_S(T) >= V_{S'}(T)$ and either *S* or *S'* or both write to *x*.

Proof: omitted.

The following data race detection algorithm used by Thread Checker is based on this theorem.

Let *T* denote the set of threads ever created. Initially,

$T = \emptyset$

The Thread Checker monitors 6 kinds of events in each thread: synchronization object creation, memory read, memory write, memory update, posting synchronization instruction and receiving synchronization instruction. For each thread *T*, the Thread Checker runs the following algorithm and takes appropriate actions upon each event.

INITIALIZE
  SET integer $i = 0$;
  $T = T \cup T$;
  CREATE vector clock V_{S_i};
  FOR each $T' \in T$ DO
    $V_{S_i}(T') = 0$;

REPEAT
  ON creating synchronization object *O*:
    CREATE vector clock K_O for object *O*;
    FOR each $T' \in T$ DO
      $K_O(T') = 0$;

  ON receiving synchronization instruction on object $O_1, O_2 \ldots O_n$:
    CREATE vector clock $V_{S_{(i+1)}}$;
    FOR each $T' \in T$ DO
      $V_{S_{(i+1)}}(T') = \text{MAX}(V_{S_i}(T'), K_{O_1}(T'), K_{O_2}(T'), \ldots, K_{O_n}(T'))$;
    DISCARD V_{S_i}
    $i = i + 1$;

  ON posting synchronization instruction on object O:
    CREATE vector clock $V_{S_{(i+1)}}$;
    FOR each $T' \in T$ DO
      $V_{S_{(i+1)}}(T') = V_{S_i}(T')$;
    $V_{S_{(i+1)}}(T) = V_{S_i}(T) + 1$;
    DISCARD V_{S_i}
    FOR each $T' \in T$ DO
      $K_O(T') = \text{MAX}(K_O(T'), V_{S_{(i+1)}}(T'))$;
    $i = i + 1$;

  ON instruction i_r reading memory location *x*;
    IF *x* has already been shadowed THEN
      RETRIEVE the shadow memory *sm* of *x*;
    ELSE
      CREATE shadow memory *sm* for *x*;
      $sm.m = x$;
      $sm.t_r = sm.t_w = -1$;
      $sm.c_r = sm.c_w = -1$;

    IF $((sm.t_w > 0)$ AND $(V_{S_i}(sm.t_w) <= sm.c_w))$
THEN
      REPORT write-read race on *x* between thread *T* at i_r and thread $sm.t_w$ at $sm.l_w$;

    $sm.t_r = T$;
    $sm.c_r = V_{S_i}(T)$;
    $sm.l_r = i_r$;

  ON instruction i_w writing memory location *x*:

IF x has already been shadowed THEN
RETRIEVE the shadow memory sm of x;
ELSE
CREATE the shadow memory sm for x;
$sm.m = x;$
$sm.t_r = sm.t_w = -1;$
$sm.c_r = sm.c_w = -1;$

IF (($sm.t_w > 0$) AND ($V_{S_i}(sm.t_w) <= sm.c_w$)) THEN
REPORT write-write race on x between thread T at i_w and thread $sm.t_w$ at $sm.l_w$;
IF (($sm.t_r > 0$) AND ($V_{S_i}(sm.t_r) <= sm.c_r$)) THEN
REPORT read-write race on x between thread T at i_w and thread $sm.t_r$ at $sm.l_r$;

$sm.t_w = T;$
$sm.c_w = V_{S_i}(T);$
$sm.l_w = i_w;$

ON instruction i_u updating memory location x:
IF x has already been shadowed THEN
RETRIEVE the shadow memory sm of x;
ELSE
CREATE the shadow memory sm for x;
$sm.m = x;$
$sm.t_r = sm.t_w = -1;$
$sm.c_r = sm.c_w = -1;$

IF (($sm.t_w > 0$) AND ($V_{S_i}(sm.t_w) <= sm.c_w$)) THEN
REPORT write-read race on x between thread T at i_u and thread $sm.t_w$ at $sm.l_w$;
REPORT write-write race on x between thread T at i_u and thread $sm.t_w$ at $sm.l_w$;

IF (($sm.t_r > 0$) AND ($V_{S_i}(sm.t_r) <= sm.c_r$)) THEN
REPORT read-write race on x between thread T at i_u and thread $sm.t_r$ at $sm.l_r$;

$sm.t_r = T;$
$sm.c_r = V_{S_i}(T);$
$sm.l_r = i_u;$

$sm.t_w = T;$
$sm.c_w = V_{S_i}(T);$
$sm.l_w = i_u;$

UNTIL thread T exits;

For simplicity, we have omitted the memory region size in the shadow memory without sacrificing correctness. We also use "shadow memory sm of x" to refer to the collection of shadow memory cells shadowing shared memory region $M(x, z)$ for the same reason. Please keep in mind that whenever we say "instruction i reading/writing/updating memory location x", we mean "instruction i reading/writing/updating memory region $M(x, z)$". Similarly, when we say "RETRIEVE the shadow memory sm of x" or "CREATE the shadow memory sm for x", we mean "RETRIEVE the shadow memory sm of memory region $M(x, z)$" and "CREATE shadow memory sm for memory region $M(x, z)$".

A thread T starts with a receiving synchronization instruction which ends the first (empty) segment and starts the second segment of T. When it sees the first receiving synchronization instruction which starts thread T, the Thread Checker creates and initializes the vector clock of the first segment on T, then process the first receiving synchronization instruction event. In other words, the INITIALIZATION part and the processing of the first receiving synchronization instruction event both take place when the first receiving synchronization instruction event occurs.

Executing a synchronization instruction by a thread and processing the same synchronization instruction event by the Thread Checker are atomic. This atomicity is needed to guarantee that the Thread Checker observes the correct event ordering. If a thread T_1 first performs a posting synchronization instruction and later a thread T_2 performs a corresponding receiving synchronization instruction, the Thread Checker will observe and process the posting synchronization instruction event by thread T_1 first and the receiving synchronization instruction event by thread T_2 later.

A memory update event can be treated as a memory read event followed by a memory write event. Treating a memory read event followed by a memory write event as a memory update event, however, reduces the total number of events and helps performance.

5 Discussion

Two approaches have been used in dynamic data race detection: lockset[1] and happens-before[2]. Each approach has its own advantages and disadvantages.

5.1 Evaluation

Lockset has at least 2 major drawbacks. First, it only works with multithreaded programs using lock-based synchronization primitives. Secondly, it can generate many false positives because of its unawareness of casual ordering of events. In contrast, happens-before works with multithreaded programs using both lock-based and non-lock-based synchronization primitives and it doesn't generate false positives.

Modern operating systems and threading models support various kinds of synchronization primitives. For example, Microsoft Windows provides events, timers, threads, processes, semaphores, messages etc. in addition to mutex locks and critical sections for thread synchronization. POSIX threading models support condition variables, semaphores, reader-writer locks, barriers, etc. in addition to mutex locks for

thread synchronization. Most multithreaded programs today use various kinds of synchronization primitives. Because happens-before can model these primitives, we chose happens-before for Win32 and POSIX multithreaded programs (we use lockset for OpenMP programs) in the Thread Checker. We believe happens-before analyze provides a better choice for the data-race analysis of these threading models.

Happens-before requires logging the history of accesses to every shared memory variable. Instead of a complete access history, we only keep the most recently read and most recently written accesses. The tool is able to catch most data races, but it doesn't guarantee to catch all data races in a single run. We believe that, in practice, catching as many races of a program run as possible in a detailed manner is better than catching all races of a program run in a brief way. The Thread Checker expends significant cost to record the dynamic stack context detail of the detected race. Furthermore, it is typically required to run the program multiple times with different test cases (due to different dynamic code paths, and thread scheduling patterns) even if all possible data races are detected in a particular program run.

Dynamic data race detection tools also fall in 2 different categories: on-the-fly and post-mortem. Post-mortem requires 2 phases: tracing and analyzing. The drawback of post-mortem is that trace files can be incredibly huge and analysis result is only available after the program run is finished. Thread Checker performs data race analysis on-the-fly. All the data is analyzed in memory without the need to dump a huge trace file and any data races detected are reported to the user as they are found, while the program is still running.

5.2 Fellow Travelers

Several other data race detections tools are either commercially available or available open source forums.

JProbe Threadalyzer[3] from Quest Software uses both happens-before and lockset to detect data races in Java programs. Visual Threads[4] from HP uses lockset to find data races in POSIX threaded programs, and happens-before analysis to reduce false positives from thread creation. Both tools are commercially available.

Helgrind of the Valgrind[5] tool suite is an open-source data race detection tool. It is based on lockset and targeted for POSIX threaded programs on Linux.

We believe the Thread Checker is the first commercially available tool supporting both Microsoft Windows and Linux.

6 CONCLUSION

In this paper, we use an example illustrating how Intel® Thread Checker detects data races. As mentioned in the introduction, the ability to observe a problem is the first step towards fixing that problem. In support of this goal we have walked through the design and implementation of the Thread Checker focusing on the instrumentation mechanism and the analysis algorithms which allow the Thread Checker to produce diagnostics which can provide insight into problems in the user's application. The valuable insights provide by the Thread Checker give software developers more confidence to create quality multi-threaded applications.

7 REFERENCES

[1] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T., Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comp. Syst.* 15, 4 (Nov. 1997), 391-411.

[2] Ronsse, M., De Bosschere, K., RecPlay: a fully integrated practical record/replay system. *ACM Trans. Comp. Syst.* 17, 2 (May. 1999), 133-152

[3] Quest Software. JProbe Threadalyzer. Available online at http://www.quest.com/jprobe, 2003.

[4] Harrow, J. J., Runtime Checking of Multithreaded Applications with Visual Threads. In *Proc. of the 7th intl. SPIN Workshop on SPIN Model Checking and Software Verification* (Aug. 30 – Sept. 1, 2000). K. Havelund, J. Penix, W. Visser, Eds. Lecture Notes In Computer Science, vol. 1885. Springer-Verlag, London, 331-342.

[5] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *In Proceedings of the 3rd Workshop on Runtime Verification.* http://valgrind.kde.org/, 2003.

APPENDIX EXAMPLE

```
#include <windows.h>
#include <stdio.h>

class StringBuffer {
private:
    char *buffer;
    int refCount;
    StringBuffer(void);
public:
    StringBuffer (const char *str) :
refCount(1) {
        buffer = new char[1024];

        for (int i = 0; str[i]; i++)
            buffer[i] = str[i];
        buffer[i] = 0;
    }

    void ToLower(void) {
        for (int i = 0; buffer[i]; i++)
            buffer[i] =
tolower(buffer[i]);
    }
    void ToUpper(void) {
        for (int i = 0; buffer[i]; i++)
            buffer[i] =
toupper(buffer[i]);
    }
    void Print(void) {printf("%s\n",
buffer);}
    void IncRefCount(void) {refCount++;}
    int DecRefCount(void) {
        if (--refCount == 0)
            delete buffer;
        return refCount;
    }
};

class String {
private:
    StringBuffer *stringBuffer;
    HANDLE lock;
public:
    String(const char *str) {
        stringBuffer = new
StringBuffer(str);
        lock = CreateMutex(0, 0, 0);
    }
    String(String& str) {
        str.Lock();
        stringBuffer = str.stringBuffer;
        stringBuffer->IncRefCount();
        str.Unlock();
        lock = CreateMutex(0, 0, 0);
    }
    ~String(void) {
        Lock();
        stringBuffer->DecRefCount();
        Unlock();
        CloseHandle(lock);
    }
    void Print(void){stringBuffer-
>Print();}
    void ToUpper(void) {
        Lock();
        stringBuffer->ToUpper();
        Unlock();
    }
    void ToLower(void) {
        Lock();
        stringBuffer->ToLower();
        Unlock();
    }
    void Lock(void)
{WaitForSingleObject(lock, INFINITE);}
    void Unlock(void)
{ReleaseMutex(lock);}
};

DWORD WINAPI
ConvertToLower(void *arg) {
    ((String *)arg)->ToLower();
    return 0;
}

DWORD WINAPI
ConvertToUpper(void *arg) {
    ((String *)arg)->ToUpper();
    return 0;
}

String *aString, *bString, *cString;
int
main()
{
    HANDLE hThread[2];

    aString = new String("Mary has a
little lamb.");
    bString = new String(*aString);
    cString = new String(*aString);

    hThread[0] = CreateThread(0, 0,
ConvertToLower, bString, 0, NULL);
    hThread[1] = CreateThread(0, 0,
ConvertToUpper, cString, 0, NULL);

    WaitForMultipleObjects(2, hThread,
TRUE, INFINITE);
    CloseHandle(hThread[0]);
    CloseHandle(hThread[1]);
    aString->Print();
    bString->Print();
    cString->Print();
    delete aString;
    delete bString;
    delete cString;

    return 0;
}
```