RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking

Yuan Yu
Microsoft Research
1065 La Avenida Ave.
Mountain View, CA 94043
yuanbyu@microsoft.com

Tom Rodeheffer
Microsoft Research
1065 La Avenida Ave.
Mountain View, CA 94043
tomr@microsoft.com

Wei Chen
Computer Science Division
University of California
Berkeley, CA 94720
wychen@cs.berkeley.edu

ABSTRACT

Bugs due to data races in multithreaded programs often exhibit non-deterministic symptoms and are notoriously difficult to find. This paper describes RaceTrack, a dynamic race detection tool that tracks the actions of a program and reports a warning whenever a suspicious pattern of activity has been observed. RaceTrack uses a novel hybrid detection algorithm and employs an adaptive approach that automatically directs more effort to areas that are more suspicious, thus providing more accurate warnings for much less overhead. A post-processing step correlates warnings and ranks code segments based on how strongly they are implicated in potential data races. We implemented RaceTrack inside the virtual machine of Microsoft's Common Language Runtime (product version v1.1.4322) and monitored several major, real-world applications directly out-of-the-box, without any modification. Adaptive tracking resulted in a slowdown ratio of about 3x on memory-intensive programs and typically much less than 2x on other programs, and a memory ratio of typically less than 1.2x. Several serious data race bugs were revealed, some previously unknown.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging—*diagnostics, monitors*

General Terms: Reliability, Performance.

Keywords: Race detection, virtual machine instrumentation.

1. INTRODUCTION

A *data race* occurs in a multithreaded program when two threads access the same memory location without any intervening synchronization operations, and at least one of the accesses is a write. Data races almost always indicate a programming error and such errors are notoriously difficult to find and debug, due to the non-deterministic nature of multithreaded programming. Since the exact schedule of threads that trigger a data race is timing sensitive, data races are difficult to reproduce, even when the program is executed repeatedly with the same inputs. Furthermore, since a data race typically results in corrupted shared data structures rather than an immediate crash, programs may continue to execute, leading to mysterious failures later in unrelated code. Automatically finding data races in a program thus is widely recognized as an important research problem.

1.1 Related Work

Automated tools for race detection generally take either a *static* or a *dynamic* approach. For static race detection, the most common approach is to employ compile-time analysis on the program source, reporting all *potential races* that could occur in any possible program execution [10, 14, 32]. The primary drawback of this approach is excessive false positives (reporting a potential data race when none exists), since the compile-time analysis is often unable to determine the precise set of possible thread interleavings and thus must make a conservative estimate. Scaling is also difficult, since the entire program must be analyzed. Another approach is to augment the programming language's type system to express common synchronization relationships, so that any well-typed program is guaranteed to be free of data races [2, 11]. This language-level approach eliminates data races altogether, but requires a substantial amount of code annotation and also restricts the kinds of synchronization that can be expressed.

For dynamic race detection, the program is executed and a history of its memory accesses and synchronization operations is recorded and analyzed. Because the history is in fact a feasible execution, dynamic detectors typically suffer a lower false positive rate than static detectors. On the other hand, since not all possible execution paths are considered, the dynamic technique is not sound and thus cannot certify a program to be free of data races. The dynamic technique also imposes runtime overhead which must be kept within limits while still providing reasonably accurate race detection. The *post-mortem* approach records events during program execution and analyzes them later [1], or records only critical events and then carefully replays the program [5, 27]. This approach is unsuitable for long-running applications that have extensive interactions with their environment. The other approach is to record and analyze information as efficiently as possible *on-the-fly*.

Existing analysis techniques used in dynamic race detectors are *lockset analysis* and *happens-before analysis*. Lockset-based detectors verify that the program execution conforms to a *locking discipline*, a programming methodology that ensures the absence of data races. Eraser [28], for example, enforces the locking discipline that every shared variable is protected by some lock. Basically, each shared variable is associated with a *lockset* that keeps track of all common locks held during accesses, and a warning is issued when the lockset becomes empty. Later lockset-based detectors [21, 25] refine Eraser's lockset algorithm with more states and transitions to reduce the number of false positives.

Happens-before detectors [1, 5, 6, 7, 18, 27, 30] are based on Lamport's *happens-before* relation [15], which combines program order and synchronization events to establish a partial temporal ordering on program statements. A data race occurs when there is no established temporal ordering between two conflicting memory accesses—neither access "happens before" the other. Compared to the lockset method, happens-before detection is more general and can be applied to programs using non-lock-based synchronization primitives such as fork/join or signal/wait. Happens-before also never reports a false positive in the absence of a data race. However, it is less efficient to implement than lockset analysis and is much more likely to suffer false negatives (failing to detect a potential race) because of its sensitivity to the particular thread schedule. Many detectors attempt to get the advantages of both lockset and happens-before analysis by combining the techniques in some way [13, 22, 24, 33]; in fact, Dinning and Schonberg originated the idea of lockset-based race detection as an improvement to a happens-before detector [8].

Dynamic race detectors also vary in the technique used to monitor program execution. In one approach, the program binary is modified to instrument each load and store instruction that accesses global memory [13, 23, 28]. While this approach is language independent and requires no program source, it also results in high overhead in both time and space. Using Eraser, for example, applications typically run 10 to 30 times slower and a *shadow word* is required for each word in global memory to store lockset information. When applied to modern programming languages, binary instrumentation also suffers from an inability to utilize type information to generate more accurate reports and reduce execution overhead. Furthermore, hooks are needed into the runtime's memory allocator so that the shadow memory can correctly reflect the state of newly-allocated memory. A modern runtime that uses a copying garbage collector presents even more problems.

Race detectors for modern object-oriented programming languages have therefore generally adopted a higher-level approach [6, 21, 22, 24, 25]. These tools modify the source code, the compiler, intermediate bytecodes, or the virtual machine in order to instrument memory accesses and synchronization operations. This approach enables race detectors to choose the appropriate detection granularity based on language-defined data structures, instead of using a fixed-size detection unit such as a word. For object-oriented languages, fields are the natural choice for a detection unit, as each field represents a distinct program variable. However, since fully monitoring every field often introduces unacceptable overhead, many current tools place the detection granularity at the object level, assuming that accesses to different fields in an object will be protected by the same lock. This technique reduces overhead significantly, often lowering the running time to within a factor of two of the original program. The performance improvement, however, is achieved at the expense of detection accuracy, as the coarser detection unit leads to many false alarms. More advanced tools provide a mechanism for focusing the analysis. Posniansky and Schuster provide a parameterized detection granularity: the program is initially run using a generous detection granularity of say, 256 bytes, and if potential races are detected the program can be re-run using a smaller granularity to get more accuracy [24]. O'Callahan and Choi provide a two-pass solution: the program is initially run using lockset analysis to identify problematic fields ("simple mode") and if any are detected the program is re-run using a combined lockset and happens-before analysis for just those fields ("detailed mode") [22]. Choi *et al.* apply compile-time analysis techniques, such as points-to and escape analysis, to determine which accesses can never participate in data races and thus do not need to be monitored [4].

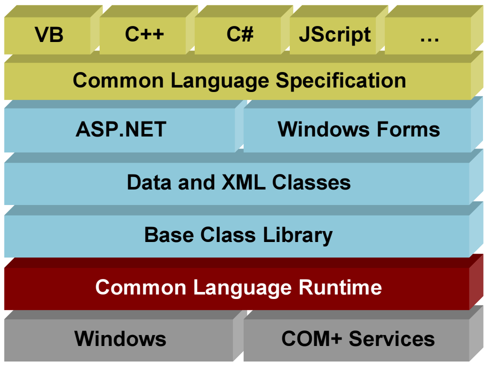

Figure 1: The .NET framework

1.2 The RaceTrack Approach

In this paper, we present RaceTrack, a practical, on-the-fly race detection tool for large multithreaded object-oriented programs on the Microsoft .NET platform. Figure 1 illustrates the .NET framework. At the core of the framework is the Common Language Runtime (CLR) [12], a language-independent layer that provides a managed execution environment for applications. Applications written in various languages are translated into the Common Intermediate Language (IL) [9], a stack-based language with memory access, arithmetic, and method call instructions and descriptive metadata containing type and version information. As an application executes, the IL is converted into native machine code by a just-in-time (JIT) compiler in the CLR. The CLR also manages the tasks of object allocation, thread creation, exception handling, and garbage collection.

Like some earlier Java-based race detectors [6, 21], RaceTrack performs instrumentation at the virtual machine level. RaceTrack uses a modified JIT compiler that emits inlined RaceTrack calls during native code generation and a modified garbage collector that allocates space with each object to store RaceTrack information. Applications run on top of the RaceTrack-modified runtime, with no user modification required. Also running on top of the CLR is a vast library of IL code that is also monitored by the RaceTrack-modified runtime.

To be practical for large, long-running programs, RaceTrack must limit itself to a reasonable overhead in processor and memory usage. To this end, RaceTrack employs a novel adaptive approach that dynamically adjusts both detection granularity and monitoring technique. When suspicious activity is detected, RaceTrack reports a warning. A post-processing warning analyzer correlates and ranks the reports.

RaceTrack does not attempt to detect all concurrent accesses. An access to location x is concurrent with another access to x if their threads hold no locks in common and neither access happens-before the other. Computing this on-the-fly requires remembering an access history for x that records a thread's lockset at the time of each access. This is in fact the basic approach of Dinning and Schonberg [8] and O'Callahan and Choi [22]. Actually, some accesses can be shown to be redundant, but you still have to keep a lot of history. For a long-running program, it becomes impractical to keep a sufficient access history for every monitored location. So instead, RaceTrack uses happens-before analysis to estimate the present number of concurrent accesses to location x. When RaceTrack monitors an access to x that it estimates is not concurrent with any prior access to x, it judges that no potential race is present and, furthermore, it resets the location's lockset to the lockset of the accessing thread. Pozniansky and Schuster introduced the idea of resetting a location's lockset at an explicit synchronization barrier [24]. RaceTrack improves on this by, in effect, dynamically estimating when a synchronization barrier has been crossed. RaceTrack's estimation technique permits it to monitor long-running programs without incurring excessive memory overhead. Of course, due to estimation, some races may be missed. The real test of whether the result is useful is if RaceTrack can direct a programmer's attention to serious race condition bugs that were otherwise unknown. We find in practice that this has been the case.

The adaptive approach of RaceTrack automatically concentrates effort to more accurately detect data races in areas that have raised suspicion, based on the assumption that a past data race access pattern is likely to be repeated in the future. However, suspicious activity of low accuracy is also reported, so that the post-process warning analyzer can incorporate it into its summary. Furthermore, at the per-field level RaceTrack also reports a subsequent access by a second or third thread, so that the warning analyzer can collate additional information about a possible race.

Specifically, RaceTrack offers the following benefits:

Coverage. Since instrumentation occurs at runtime, RaceTrack is language independent and automatically monitors any managed code that executes on top of the CLR. This includes a vast amount of library code. Due to heavy and pervasive use, a data race bug in a library is likely to be quite dangerous. Equally importantly, a data race inside library code is most likely caused by misuse of the library, a very common class of bugs in the application code. Using RaceTrack we have found several data race bugs in the .NET Base Class Library [19] (see Section 4). These bugs have been reported and corrected in a later release.

Accuracy. RaceTrack automatically adapts to monitor memory accesses at the field level, which makes its race reporting fundamentally more accurate than coarse-grained object race checkers. Furthermore, it can detect races on individual array elements, which many existing checkers avoid for performance reasons. RaceTrack also automatically adapts to employ happens-before checking, which permits filtering out lockset-only false positives that stem from fork/join parallelism and asynchronous method calls.

Performance. RaceTrack imposes a relatively low overhead due to its adaptive tracking techniques. Experimental results show that the overhead incurred by RaceTrack is well below that incurred by other dynamic race detectors. For example, on the SpecJBB benchmark, the lockset-based Java-runtime data race detector of Nishiyama suffers about 8x slowdown [21]; O'Callahan and Choi report a 2.9x slowdown in "simple mode" which has to be followed by a re-run in "detailed mode" that suffers a 2.2x slowdown [22]; whereas RaceTrack suffers only about 3x slowdown. We should also point out that RaceTrack monitored the entire 30 minute run of SpecJBB and the entire .NET library. For other benchmarks with less memory activity, RaceTrack typically suffers a slowdown of less than 1.3x. Regarding memory overhead, RaceTrack with adaptive granularity typically suffers a memory ratio of less than 1.2x. We were unable to find comparative numbers in the literature. Detailed performance numbers for RaceTrack appear in Section 4.2.

Scalability. Due to its runtime instrumentation and low overhead, RaceTrack can be applied to large, long-running, real-world applications. Implementing RaceTrack entirely inside the CLR, although an admittedly rather technical undertaking for a very sophisticated runtime, makes it completely transparent to applications. Many issues, such as interaction between managed and native code and object relocation by the garbage collector, are addressed cleanly in RaceTrack. Some other tools are unable to tolerate the actions of a copying garbage collector, and have to disable it [4, section 3.3]. This basically rules out running their race detectors on major applications.

The rest of the paper is organized as the follows. Section 2 describes RaceTrack's hybrid detection algorithms. Section 3 discusses our novel runtime instrumentation strategy, including a number of optimizations that further reduce RaceTrack's overhead. Section 4 reports the experimental results for the overhead and accuracy of our tools, also showcasing a number of real application bugs detected. Section 5 concludes the paper.

2. THE DETECTION ALGORITHM

Although lockset-based detection exhibits good performance and a relative insensitivity to execution interleaving, one well-observed shortcoming is its failure to take into consideration threading primitives such as fork/join and asynchronous calls. As a result, programs using such primitives typically exhibit many false alarms. Our new detection algorithm corrects this failure while retaining lockset's good properties. After describing the basic algorithm, we present the refinements necessary to make it practical for large, real applications.

Our improvement is based on the observation that a race on a variable can only occur if multiple threads are concurrently accessing the variable. So, in addition to tracking a lockset for each monitored variable, RaceTrack also tracks a

```
At $t$:Lock($l$):
    $L_t \leftarrow L_t \cup \{l\}$

At $t$:Unlock($l$):
    $L_t \leftarrow L_t - \{l\}$

At $t$:Fork($u$):
    $L_u \leftarrow \{\}$
    $B_u \leftarrow Merge(\{\langle u, 1\rangle\}, B_t)$
    $B_t \leftarrow Inc(B_t, t)$

At $t$:Join($u$):
    $B_t \leftarrow Merge(B_t, B_u)$

At $t$:Rd($x$) or $t$:Wr($x$):
    $S_x \leftarrow Merge(Remove(S_x, B_t), \{\langle t, B_t(t)\rangle\})$
    if $|S_x| > 1$
      then $C_x \leftarrow C_x \cap L_t$
      else $C_x \leftarrow L_t$
    if $|S_x| > 1 \wedge C_x = \{\}$ then report race
```

Figure 2: Pseudocode of the basic algorithm

set of concurrent accesses that we call a *threadset.* We use the popular *vector clock* technique [17] to determine which actions are ordered before other actions. Each thread has a private virtual clock that ticks at certain synchronization operations. Synchronization operations propagate information about clock values and each thread maintains the most recent clock value it knows about for each other thread. The clock values known by a thread comprise that thread's vector clock.

The threadset method works basically as follows. Each time a thread performs a memory access on a monitored variable, it forms a label consisting of the thread's identifier and its current private clock value and adds this label to the variable's threadset. The thread also uses its vector clock to identify and remove from the threadset labels that correspond to memory accesses that are ordered before the current access. Hence the threadset contains labels for accesses that are estimated to be concurrent. Whenever the threadset is a singleton, we conclude that the variable, at least for the moment, is no longer shared between threads. The following sections describe our method in detail.

2.1 Vector Clocks and Threadsets

Let T be the set of all threads. We use the letters t, u, and v to denote particular threads. Each thread has a private virtual clock whose value ranges over the set of natural numbers $\{1, 2, 3, ...\}$. When thread t performs a memory access, we label the access with the pair $\langle t, k\rangle$, where k is the current value of t's virtual clock. For brevity, we often refer to an access by its label.

Just because one access is executed *prior* to another does not necessarily mean the former is *ordered before* the latter. To determine which prior accesses are ordered before its next memory access, thread t maintains a *vector clock* called B_t. B_t maps thread identifiers to clock values. $B_t(u)$ is thread t's knowledge of the largest clock value for thread u such that any prior access $\langle u, B_t(u)\rangle$ is ordered before the next memory access to be executed by thread t. If thread t does not know anything about thread u, then $B_t(u) = 0$. We define $B_t(t)$ as the current value of the private virtual clock of thread t itself. This definition is consistent because any prior access performed by thread t has a label $\langle t, k\rangle$ where $k \leq B_t(t)$ and this prior access is ordered before the next memory access to be executed by thread t.

We use the letter x to denote a particular monitored variable. To keep track of which accesses to x are concurrent, x maintains a *threadset* called S_x. S_x maps thread identifiers to clock values. $S_x(u)$ is variable x's knowledge of the largest clock value for thread u such that an access $\langle u, S_x(u)\rangle$ of x has occurred and has not been ordered before any later-executed access of x. If x does not know of any such access by thread u, then $S_x(u) = 0$. RaceTrack considers the threadset S_x to be the present estimate of the set of concurrent accesses to x.

Note that even though their meanings are quite different, threadsets and vector clocks both have the same formal type: a map from threads to clock values. To simplify the description of the basic algorithm, we define the following operations on this type:

$$\begin{aligned}
|V| &\triangleq |\{t \in T : V(t) > 0\}| \\
Inc(V, t) &\triangleq u \mapsto \text{if } u = t \text{ then } V(u) + 1 \text{ else } V(u) \\
Merge(V, W) &\triangleq u \mapsto max(V(u), W(u)) \\
Remove(V, W) &\triangleq u \mapsto \text{if } V(u) \leq W(u) \text{ then } 0 \text{ else } V(u)
\end{aligned}$$

V and W denote maps from threads to clock values and t and u denote threads. For convenience, we consider a map V to be equivalent to the set of all pairs $\langle t, V(t)\rangle$ such that $V(t) > 0$. $|V|$ is the size of the set, $Inc(V, t)$ is the result of ticking thread t's virtual clock, $Merge(V, W)$ is the result of merging V and W to get the most up-to-date information, and $Remove(V, W)$ is the result of removing elements of V that are ordered before W.

2.2 The Basic Algorithm

For each thread t, RaceTrack maintains a lockset L_t and a vector clock B_t. L_t is the current set of locks held by thread t and B_t is thread t's most recent information about each thread's virtual clock. For each monitored variable x, RaceTrack maintains a lockset C_x and a threadset S_x. C_x is the current set of locks protecting variable x and S_x is the current set of concurrent accesses to x. When x is allocated, C_x is initialized to the set of all possible locks and S_x is initialized to $\{\}$.

Figure 2 gives pseudocode showing how the data structures are updated when relevant threading primitives and memory accesses are executed. t:Lock(l) and t:Unlock(l) denote the acquisition and release of lock l by thread t, t:Fork(u) denotes the creation of a new thread u by thread t, t:Join(u) denotes that thread t blocks until the termination of thread u, and t:Rd(x) and t:Wr(x) denote the read and write of x by thread t. For brevity, we show only the basic threading primitives. Asynchronous calls (BeginInvoke and EndInvoke) and condition variables (Pulse and Wait) require essentially the same updates as Fork and Join. Creating an initial thread is also essentially the same as Fork.

For t:Lock(l) or t:Unlock(l), we only need to update L_t accordingly.[1] As in past work that combines the lockset and happens-before techniques, we assume that lockset analysis

[1]Reentrant locks in languages such as C# and Java complicate the matter only slightly.

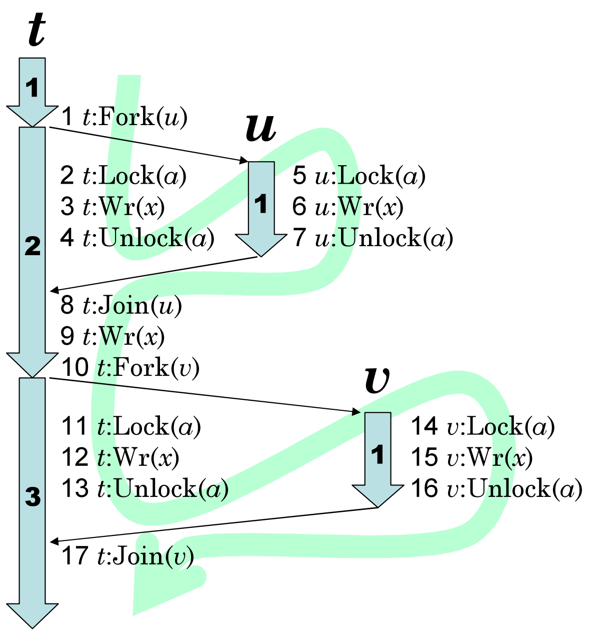

Figure 3: An example for the basic algorithm
Fat arrows show thread execution and skinny arrows show synchronization. Thread clock is indicated inside the fat arrow. Instruction numbers indicate sequential order of execution of an example interleaving.

covers the synchronization intended by Lock and Unlock and we ignore any happens-before ordering that results from these operations [8, 22].

For t:Fork(u), we initialize L_u to the empty set. The virtual clock of u is initialized to 1 and since any operation ordered before t's current point of execution is also considered to be ordered before u's current point of execution, B_t is merged in. The virtual clock of thread t then ticks to reflect the fact that any subsequent operation of t after forking u may be concurrent with subsequent operations of u. For t:Join(u), we merge B_u into B_t, since any operation ordered before u's current point of execution is also considered to be ordered before t's current point of execution. Note that we don't need to tick the virtual clock of thread t in this case.

The memory accesses t:Rd(x) and t:Wr(x) form the most interesting case. Each $\langle u, k\rangle \in S_x$ represents a prior access of x, and if $k \leq B_t(u)$, we know that this prior access is ordered before the current access, so it should be removed from S_x. Then to account for the current access, we merge $\{\langle t, B_t(t)\rangle\}$ into S_x. C_x is updated depending on the new value of S_x. If $|S_x| > 1$, we know that x is being accessed by multiple concurrent threads, so C_x is updated to the intersection of C_x and L_t, as in the normal operation of the lockset algorithm. However, if $|S_x| = 1$, we conclude that x is being accessed only by t, so C_x is simply set to L_t. We only issue a race condition warning when the size of the new S_x is greater than 1 and the new C_x is empty, which is to say that x is being accessed concurrently by multiple threads and there is no common lock protecting those accesses.

2.3 Example

Figure 3 shows an example execution of a simple program. Observe that variable x is initially protected under lock a, then there is a single-threaded phase where x is not protected at all, and then in a later phase x is again protected under lock a. This is a fairly typical behavior found in a parallel program. Since there is no common lock, the lockset method would report a false alarm. RaceTrack avoids this problem. Let us work through the updates.

Initially we would have $L_t = \{\}$, $B_t = \{\langle t, 1\rangle\}$, $C_x =$ the set of all locks, and $S_x = \{\}$. In step 1, t forks thread u, which results in $B_t = \{\langle t, 2\rangle\}$, $L_u = \{\}$, and $B_u = \{\langle t, 1\rangle, \langle u, 1\rangle\}$. Then in steps 2–4, t acquires a, writes x, and releases a. This updates the state for x to be $C_x = \{a\}$ and $S_x = \{\langle t, 2\rangle\}$. Then in steps 5–7, u acquires a, writes x, and releases a. This leaves C_x unchanged (since a is a common lock) and results in $S_x = \{\langle t, 2\rangle, \langle u, 1\rangle\}$. Although thread u knows that its write in step 6 is ordered after any $\langle t, 1\rangle$ access, there is no ordering with respect to the $\langle t, 2\rangle$ access, so both accesses must be left in S_x. Similar reasoning would apply if the interleaving happened to occur in the opposite order, so, in fact, no matter which of the accesses 3 or 6 happens to execute first, the results for C_x and S_x are the same.

Then in step 8, t joins thread u, which results in $B_t = \{\langle t, 2\rangle, \langle u, 1\rangle\}$. As a consequence, in step 9, thread t knows that its access is ordered after any $\langle u, 1\rangle$ access. This results in $C_x = \{\}$ and $S_x = \{\langle t, 2\rangle\}$. Although the lockset is empty, only one thread has a concurrent access to x, so RaceTrack gives no warning.

Looking ahead to steps 10–17, t forks thread v and then both threads access x under lock a. In the example, t makes the access first in step 12, resulting in $C_x = \{a\}$ and $S_x = \{\langle t, 3\rangle\}$. Note that C_x was set from L_t because the resulting S_x had only one element, indicating that no other accesses were concurrent. Then when v makes its access in step 15, S_x grows to two elements, indicating multiple concurrent accesses, but $C_x = \{a\}$, indicating that all concurrent accesses are protected by a common lock. Similar reasoning would apply if the interleaving happened to occur in the opposite order, so, in fact, no matter which of the accesses 12 or 15 happens to execute first, the results for C_x and S_x are the same.

2.4 Refinements

Maintaining the locksets and threadsets of variables during program execution as described in the preceding section can be very expensive in both time and space. In this section, we refine the basic algorithm to make it practical for real-world applications.

2.4.1 *Adaptive Threadset*

To exploit the common case in which a variable is either never shared, or, if shared, is always protected by the same lock, RaceTrack employs the *adaptive threadset* technique of not tracking the threadset until the lockset would report a race. Consequently, RaceTrack's state machine, shown in Figure 4, is more complex than that used by Eraser and some other similar tools. Considering a variable x, the states are described as follows.

Virgin. x has been allocated, but not yet accessed by any thread. On the first access we enter ***Exclusive0***.

Exclusive0. x has been accessed exclusively by one thread only. In this state, we track the thread id. When an access occurs from a different thread, we enter ***Exclusive1***.

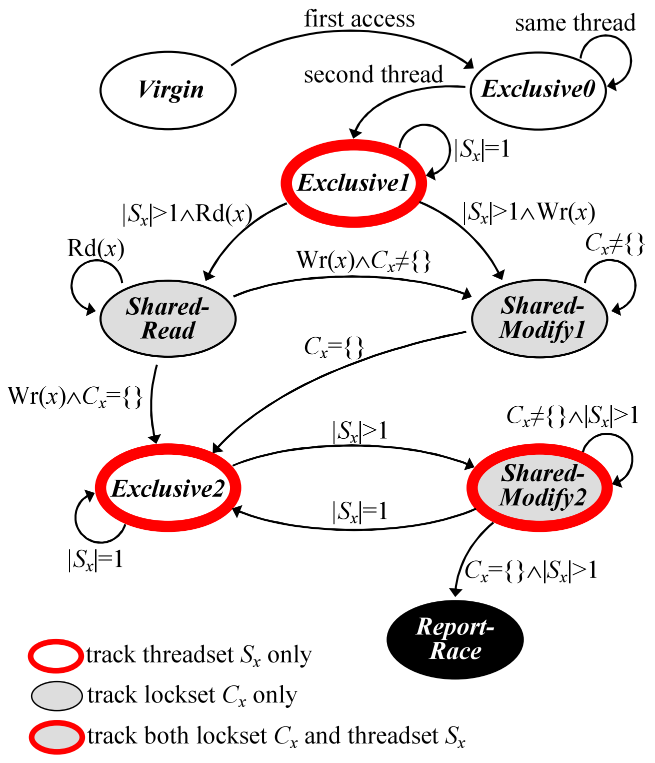

Figure 4: RaceTrack's state machine

Exclusive1. x is accessed exclusively by one thread. In this state, we track a singleton threadset $S_x = \{\langle t, k\rangle\}$ corresponding to the most recent access to x. We remain in this state as long as each successive access maintains $|S_x| = 1$, meaning that there are no concurrent accesses to x. Note that different threads may perform accesses—if, for example, x is passed from one thread to another—but all accesses must be ordered according to the threads' vector clocks. When an access occurs that would result in $|S_x| > 1$, we discard the threadset and enter ***Shared-Read*** or ***Shared-Modify1***.

Shared-Read. x is concurrently accessed by multiple threads, but all accesses are reads. We enter this state from ***Exclusive1*** when a read t:Rd(x) results in the detection of multiple concurrent accesses. In this state, we track only the lockset C_x, which is initialized to L_t.

Shared-Modify1. x is read and written concurrently by multiple threads. We enter this state either from ***Exclusive1***, when a write t:Wr(x) reveals multiple concurrent accesses to x, or from ***Shared-Read***, when a write t:Wr(x) occurs. As in ***Shared-Read***, we track only the lockset in this state. If we enter from ***Exclusive1***, the lockset C_x is initialized to L_t.

Exclusive2. x is accessed by multiple threads and the lockset discipline alone is not sufficient for protection. We track a singleton threadset in this state. The threadset S_x is initialized to contain only the thread and virtual clock of the access causing the transition. Similar to ***Exclusive1***, we remain in this state as long as $|S_x| = 1$. When $|S_x| > 1$ we enter ***Shared-Modify2***.

Shared-Modify2. x is concurrently read and written by multiple threads. Both lockset and threadset are tracked. Whenever S_x is reduced to contain only one element $\langle t, k\rangle$, we go back to ***Exclusive2***.

Report-Race. A potential race is detected and reported. This state is reached only from ***Shared-Modify2*** when $C_x = \{\}$ and $|S_x| > 1$, which means that x is concurrently accessed by multiple threads without a common lock.

The distinction between ***Exclusive0*** and ***Exclusive1*** enables a subtle optimization described later in Section 3.3. The use of the singleton threadset in ***Exclusive1*** efficiently tracks a common object-passing style. Harrow [13] described the same improvement in the context of Eraser, except that we implement it more efficiently by using vector clocks instead of a thread segment ordering graph. The idea of deferring happens-before analysis until lockset analysis proves insufficient appears in Pozniansky and Schuster [24]. RaceTrack adds the novel idea of returning to ***Exclusive2*** and resetting the lockset when the set of concurrent accesses falls to one.

2.4.2 *Adaptive Granularity*

Praun and Gross observed that the fields of an object are often treated as a single unit of protection and that detecting races at object granularity works reasonably well in practice [25]. The main benefit is that it enjoys a substantial reduction in memory overhead compared to tracking each field individually. However, coarse granularity can lead to false alarms when distinct fields of the same object are protected by different locks.

RaceTrack employs a novel *adaptive granularity* technique to avoid this problem. The basic idea is to start with object-level detection and automatically refine it to field-level detection when a potential race is detected. Essentially, we compose two copies of the state machine shown in Figure 4. RaceTrack first runs the detection algorithm at the object level, treating each object as a single variable and therefore tracking only one lockset and one threadset per object. If ***Report-Race*** is reached, RaceTrack then switches the object to field granularity and continues monitoring the program, now treating each individual field as a different variable. The same thing happens for arrays. RaceTrack treats an array just like an object, with each of the elements in the array being treated like a field.

2.4.3 *Dynamic Adaptation*

By dynamically adapting from object granularity to field granularity and from lockset detection to combined lockset/threadset detection, RaceTrack concentrates more effort to more accurately detect data races in areas that have raised suspicion, based on the assumption that a past data race access pattern is likely to be repeated in the future. Of course, if the access pattern is not repeated, the race will be missed by the more accurate methods. For this reason, RaceTrack generates warnings even when using methods of low accuracy. All of the warnings are collated and ranked by a post-processing warning analyzer, which is described later in Section 3.6.

3. IMPLEMENTATION

We have implemented RaceTrack as modifications to the current production version (v1.1.4322) of Microsoft's Common Language Runtime (CLR) [12]. RaceTrack can monitor any managed application out-of-the-box, with no application modifications required. We originally prototyped RaceTrack using the Shared Source Common Language Infrastructure (SSCLI) [20] and we plan to release this prototype in the near future.

3.1 Instrumenting the Virtual Machine

We implemented RaceTrack by modifying the virtual machine of the CLR runtime. The modifications, mainly in C++ and x86 assembly code, occurred in three main areas.

- To track the lockset and threadset for each thread, we modified the implementation of various threading primitives including thread Start, Join, and IsAlive; monitor Enter, Exit, Wait, and Pulse; reader-writer lock Acquire and Release; and asynchronous call BeginInvoke and EndInvoke. Reader-writer locks are handled using the technique introduced by Eraser [28]: when performing a write access, a thread is considered to hold only those locks it owns in write mode, since a read mode lock offers no protection against concurrent read accesses.
- To track the lockset and threadset for each variable, we modified the JIT compiler to emit extra code for various IL instructions including **`ldfld`**, **`stfld`**, **`ldsfld`**, **`stsfld`**, **`ldelem`**, and **`stelem`** (load and store operations on instance fields, static fields, and array elements). The extra code consists of a helper call to a RaceTrack function that manages the state machine.
- To store the lockset and threadset for each variable, we modified the memory allocator and garbage collector to reserve extra memory space and reclaim it when the variable is garbage collected. Modifications in the garbage collector were also needed to help manage two RaceTrack internal tables.

Once we attained a reasonable understanding of the CLR implementation, the modifications were fairly straightforward. Perhaps the most surprising item is the thread primitive IsAlive. In our early experiments with threadset, we encountered a program that used IsAlive to see if all its worker threads had terminated and, if so, proceeded to modify various shared variables and then fork another batch of workers. RaceTrack was giving false alarms on this program. After a little thought, we realized that when IsAlive returns false, it has the same meaning as Join. Accounting for this suppressed the false alarms. More interesting modifications were needed to store RaceTrack data structures, as discussed next.

3.2 Managing RaceTrack Data Structures

For simplicity, we describe the treatment of objects and their instance fields. Objects are allocated by the CLR on a garbage-collected (*managed*) heap. The treatment of static fields is technically different but conceptually the same. It is worth pointing out that our current implementation may not be the ideal solution, but was instead designed to avoid any major disruption to the existing runtime system.

Like Eraser, RaceTrack maintains a lockset table that contains all the distinct locksets ever created and identifies a lockset uniquely by its index in the table. Each lock has a corresponding internal unmanaged (and therefore immobile) runtime data structure and we use the address of this data structure to identify the lock.[2] For efficiency, lookups in the lockset table are lock-free while inserts are serialized by a mutex. When the lockset table fills up, we rehash into a bigger table. This leaves the problem of how to safely discard the old table, since lookups on other threads may still be accessing it. The solution is to employ the garbage collector. The CLR arranges for all other threads to remain at a "safe" point while the garbage collector scans for garbage and compacts the object store. During this time it is also safe to discard the old table.

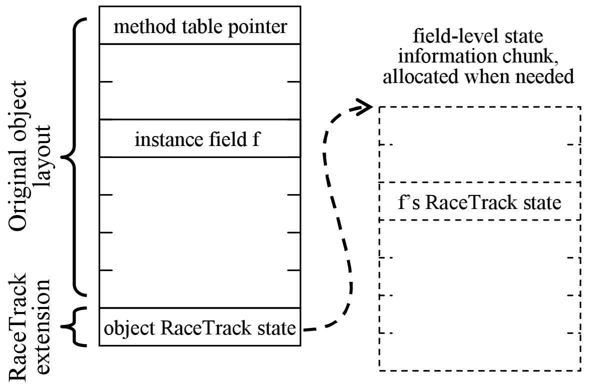

Figure 5: RaceTrack's object layout

In a similar fashion, RaceTrack maintains a threadset table and identifies a threadset uniquely by its table index. Now, whereas the set of distinct locksets ever created tends to be limited, this is not true of the set of distinct threadsets. Therefore RaceTrack uses a reference counting scheme to discard unneeded threadsets. Again, we employ the garbage collector to sweep the table at a safe point.

As described in Section 2.4, RaceTrack starts with object-level tracking and adapts to field-level tracking when needed. Figure 5 shows the object layout used by RaceTrack. For each object, RaceTrack adds an extension to store the object-level state information. The extension is always placed at the very end of an object in order to preserve the original object layout in memory. This is especially important in the presence of native methods that assume they know the object layout, and it is difficult to achieve with source-level instrumentation [4] because of subclassing. An alternative would be to place the RaceTrack extension at a negative offset in front of the original object. Although such an alternative might be simpler, we rejected it because it would interfere with the garbage collector's rather tricky use of that area as scratch space during compaction.

As illustrated in Figure 5, when RaceTrack refines the object to field-level tracking, an additional chunk of (unmanaged) memory is allocated to store the state information of the tracked fields of the object. The address of this chunk is stored in the object's extension.

[2]Actually, in the CLR the runtime lock data structure is allocated in a lazy manner and associated with an object only when needed. We had to disable a CLR optimization that disassociated the lock if it had been released for a long time.

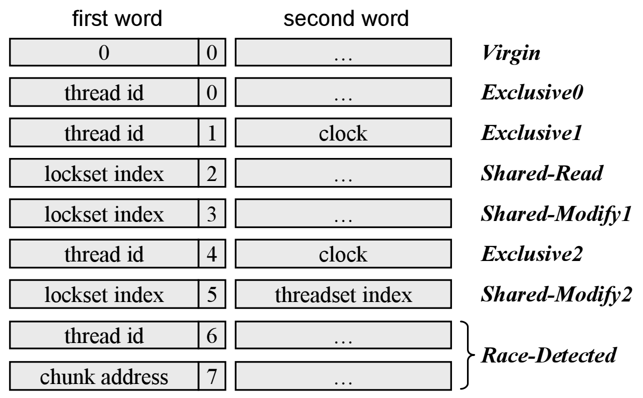

Figure 6: RaceTrack's state encoding

Obviously, an object's extension is garbage collected along with the object. For the chunk of field-level storage, we modified the garbage collector to free it, if it exists, at the time the object is determined to be unreachable.

The CLR supports finalization of unreachable objects. A special finalizer thread retrieves the object from a finalizer queue and executes its finalizer method, which can do anything with the object, including making it reachable again. Experience shows that finalizer methods are often a source of data races due to scheduling interactions between the finalizer thread, the garbage collector, and user threads; yet we know of no prior work that handles these methods properly. Our solution is simple and effective: before an unreachable object is finalized, RaceTrack resets the state of the object to ***Exclusive0***, indicating that the object is now exclusively "owned" by the finalizer thread.

3.3 Encoding RaceTrack State Information

RaceTrack stores the object-level or field-level state information in two 32-bit words, as illustrated in Figure 6. The first word contains three bits that encode the state (Figure 4) and 29 bits of other information. The interpretation of the remainder of the first word and of the second word depends on the state. ***Exclusive0*** records the thread id in the first word and ignores the second word. ***Virgin*** uses the same encoding as ***Exclusive0*** but employs a null thread id. ***Exclusive1*** and ***Exclusive2*** record the thread id in the first word and the thread's virtual clock in the second word. ***Shared-Read*** and ***Shared-Modify1*** record the lockset index in the first word and ignore the second word.[3] ***Shared-Modify2*** records the lockset index in the first word and the threadset index in the second word. In object-level state information, ***Race-Detected*** records the address of the additional chunk in the first word. In field-level state information, ***Race-Detected*** records the offending thread id in the first word.

To avoid unnecessary synchronization, updates of the state information are performed in a lock-free manner. RaceTrack reads both words using ordinary memory operations, computes what the new state information would be, and then atomically replaces the old information with the new information using the x86 64-bit interlocked compare-and-exchange instruction `cmpxchg8b`. A concurrent update might interfere with this process, causing the compare-and-exchange to fail, whereupon RaceTrack tries the process again. Failure is very infrequent unless the monitored program is pathological.

When RaceTrack reads the state words using ordinary memory operations, it might get an inconsistent combination: one word belonging to one atomic state information and the other belonging to another. So RaceTrack is careful when computing the new state information not to get confused because of this possibility, even though the eventual compare-and-exchange will fail. RaceTrack performs a subtle optimization in the case of a transition between states that both use only the first word. In such a case it is correct to use the 32-bit interlocked compare-and-exchange instruction `cmpxchg` to update just the first word and, furthermore, if the old and new values are identical, the update can be skipped entirely. ***Exclusive0*** is separated from ***Exclusive1*** precisely in order to exploit this optimization, and ***Shared-Read*** and ***Shared-Modify1*** also benefit. This subtle optimization improves performance by about 5% on some benchmarks.

3.4 Untracked Accesses

In theory, RaceTrack could track all static fields and all instance fields in all objects of all classes. In practice, certain classes of objects and accesses are not tracked.

- Field accesses by unmanaged native code are not tracked. A non-negligible number of methods in the CLI base library are implemented by unmanaged C++ and assembly code. Races involving memory accesses in these native methods will be missed.
- In the CLI, strings are immutable: the contents of a string cannot be modified. Furthermore, read-only fields cannot be modified and thread-local variables cannot be shared. Therefore we don't track these cases. We don't even allocate space for RaceTrack state information for untracked fields.
- Volatile fields and interlocked operations such as compare-and-exchange are designed to be used by lock-free code. To avoid spurious warnings, we decided not to track any memory access of a volatile field or interlocked operation.

3.5 Reducing Memory Usage

In implementing RaceTrack, we took care to minimize memory consumption. While the adaptive techniques of Section 2.4 have, to a large extent, addressed this problem, there are still cases in which RaceTrack could potentially cause excessive memory consumption. Below, we discuss two such cases along with our solutions. In both cases, we trade accuracy in favor of performance and scalability. In our experience, the accuracy loss occurs rarely in practice and thus these heuristics are justified.

First, it is easy to see from Section 3.2 that the memory overhead of RaceTrack could be worse than $8n$ bytes for an array of size n. This overhead looks particularly severe for large arrays. One approach is to give up on element-level detection when an array exceeds certain size. RaceTrack takes a variation of this approach by tracking the first, the middle, and the last m elements, for a given configuration

[3]As an optimization, if the lockset contains no more than two elements, the lock identifiers could be packed directly into the state information. This would require a mechanism to distinguish the packed format from the general case.

parameter m. If the size of an array is not greater than $3m$, all its elements will be tracked independently during field-level detection. However, for arrays having more than $3m$ elements, some of the elements will not be tracked and races could be missed because of this heuristic. In the current implementation, $m = 128$.

Second, as described in Section 2.2, RaceTrack maintains a vector clock for each thread. The vector clock can potentially grow very large in size. RaceTrack therefore imposes an upper bound on the size of a thread's vector clock. It also records the temporal ordering of the updates to the vector clock. So, when the size of the vector clock reaches the upper bound, RaceTrack shrinks it in half by removing the oldest elements in the vector clock. In the current implementation, the upper bound is set to 8192. By discarding some ordering information, it is possible that this heuristic may enlarge the threadsets of variables and hence cause false positives, but we have not noticed any problems in practice.

3.6 Improving Warning Analysis

Even once a potential race condition is reported, it is often difficult to analyze the program to determine if the race is real and what effect it might have. RaceTrack assists this task by issuing generous warnings and employing a post-process warning analyzer to collate and rank them all. Following are the important features of this process.

Multiple stack traces. It is valuable for debugging to have stack traces of each of the conflicting memory accesses that cause a potential race condition to be detected. Unfortunately, at the time of the earlier accesses, RaceTrack does not yet know that a warning will ensue, and the cost of saving every stack trace on speculation would be extremely high. We therefore took a different approach that incurs very little additional cost, based on the assumption that a past activity pattern is likely to be repeated in the future. When RaceTrack first detects a potential race on a variable, it reports the stack trace of the suspicious memory access, but it continues tracking subsequent accesses to that variable. If the variable is accessed again by a different thread, RaceTrack reports the stack trace of this new access. This approach may not always produce the best collection of stack traces, but we found it to be very useful in practice, for most of the time it does produce two distinct stack traces of suspicious access to the variable.

Ranking. RaceTrack ranks the warnings so that developers can focus attention on the most likely ones. As explained in Section 2.4, RaceTrack produces warnings with different levels of accuracy. We rank field-level warnings higher than object-level warnings, and warnings detected at ***Shared-Modify2*** higher than the ones detected at ***Shared-Modify1***. In addition, we found in practice that the warnings for which RaceTrack failed to produce multiple stack traces were less likely to be real, since the suspicious variables causing those warnings were not accessed again by another different thread. This observation led us to the heuristic of ranking them below the ones with multiple stack traces.

Classification. Often many identical or closely-related warnings are generated. For example, if the program contains a data race on an instance field, essentially identical warnings are likely to be reported for every object of that class. To deal with this issue, the postprocessor applies the techniques of fingerprint [26] and similarity analysis [3] to classify the warnings based on stack frame similarity. Such classification also helps to identify the relevant code so that the warnings can be directed to the responsible developers.

To minimize any performance impact on the CLR runtime, RaceTrack saves warnings in a file as they are detected. The warning analyzer then processes the file and generates a pretty-printed summary. Correlated warnings for the same variable are identified by a unique identifier assigned by RaceTrack,[4] and synthesized into a single warning. The final result is a list of files containing warnings sorted by their rankings.

3.7 Annotations

Like other similar tools, RaceTrack sometimes produces false alarms. To allow a user to suppress warnings that are determined to be false alarms, we provided a simple annotation mechanism. Since RaceTrack tracks all managed code dynamically loaded and executed by the virtual machine, regardless of the source of the code, we adopted an unconventional approach to annotation that does not require any code modification. We use a separate *annotation file* to tell RaceTrack what it does not need to track. Currently, the annotation file is simply a list of fields, methods, and classes. The example annotation file

```
Field  System.Collections.Hashtable.count
Method System.Environment.GetStackTrace
Class  System.IO.File
```

instructs RaceTrack not to monitor any access to the `count` field of class `System.Collections.Hashtable`, any field access that appears in the `GetStackTrace` method of class `System.Environment`, and any access to any field of class `System.IO.File`. So far, we have found that such simple annotations have worked well in practice. Annotations for additional cases such as user-defined locks could easily be added by simple extensions to the vocabulary of the annotation file.

4. EXPERIENCE AND PERFORMANCE

Our experience with RaceTrack started with running the Common Language Runtime (CLR) basic regression test suite, which consists of 2122 tests designed to exercise the CLR and associated libraries. Passing the test suite helped assure us that our RaceTrack implementation did not interfere with the expected operation of the CLR. Although the basic CLR is written in native code, a vast collection of library code is written in C# or other languages that compile into the Common Intermediate Language (IL), for example, VB, Jscript, and IL itself. Since RaceTrack is implemented in the virtual machine, it monitors all of this code as it is executed. Running the CLR regression test suite, we estimated that RaceTrack monitored about half a million lines of code. Some of the regression tests happen to exercise library functions in a multithreading context. Our August 2004 version of RaceTrack reported about 1200 warnings for

[4]The memory address of a variable can not be used to uniquely identify the variable because the garbage collector can move the variable to different addresses in memory.

the entire suite. We spent over a week analyzing and categorizing all of these warnings by hand. We found non-bugs, near-bugs, and some serious bugs. Section 4.1 describes our manual categorization effort and three of the serious bugs we found.

RaceTrack provided an interesting perspective on the library code. Some programmers clearly assumed that locks would be expensive and so wrote parts of their code in a lock-free style. RaceTrack is likely to report warnings for such code. If the lock-free code happens to be correct, these warnings are false alarms. However, it is difficult even for experienced programmers to write correct lock-free code. By analyzing RaceTrack warnings, we found four real bugs in lock-free code, of which two were unknown and the other two had only recently been found. These bugs had existed for many years. RaceTrack served to focus special code-review attention on a few hundred lines of problematic lock-free code hiding within a vast expanse of innocuous code.

The manual categorization effort led to several improvements in RaceTrack. First, we noticed that about 95% of the warnings were duplicates, implicating the same source code as a prior warning on a different object or in a different test. Scanning these manually got tiresome very quickly. The solution was to add a post-process warning analyzer to organize and summarize raw warnings into a more efficient report. Second, we found that some of the warnings were too difficult to analyze because a single stack trace did not provide enough information. A data race, after all, requires two threads by definition. The solution was to report useful subsequent warnings for the same field and arrange for the warning analyzer to collate this extra information. Third, we observed that some warnings were false alarms because the accesses were serialized by fork/join-type operations. Adding threadset analysis eliminated these false alarms, but maintaining a vector clock for every field consumed so much memory that we could not monitor any decently-sized program. The solution was to employ *adaptive threadset*, automatically resorting to threadset analysis only where lockset analysis was insufficient. Fourth, we observed that individually monitoring each field consumed memory that was unnecessary in many cases, for example, if the object was never shared. The solution was to employ *adaptive granularity*, automatically resorting to field-level monitoring only where object-level monitoring was insufficient.

In August 2005 we measured the performance of our improved version of RaceTrack on several large, real-world programs. (We continued to test RaceTrack on the CLR regression suite, but since the suite so rarely uses multithreading it is not interesting as a benchmark.) The worst impact we saw was a slowdown of about 3x and a memory consumption of about 1.5x when using RaceTrack relative to the performance of the unmonitored program, which we consider acceptable. Section 4.2 reports the details. RaceTrack reported about a dozen distinct warnings in these programs that appear to be serious bugs based on a preliminary review. However, in most cases they have not been confirmed by the owners of the programs.

4.1 CLR Regression Test Suite

Our August 2004 version of RaceTrack reported about 1200 warnings for the basic CLR regression test suite. Eliminating duplicates left 70 distinct warnings, of which 22 implicated code in either higher-level libraries not part of the CLR or in the test drivers. After much head-scratching, we categorized the 48 distinct CLR warnings as shown in Table 1.

#	Category	
6	A.	false alarm - fork/join
2	B.	false alarm - user defined locking
5	C.	performance counter / inconsequential
4	D.	locked atomic mutate, unlocked read
4	E.	double-checked locking
2	F.	cache semantics
2	G.	other correct lock-free code
7	H.	unlocked lazy initialization
8	I.	too complicated to figure out
8	J.	potentially serious bug
48	total	

Table 1: Manual categorization of RaceTrack warnings for the CLR regression test suite (August 2004)

```
class UnlockedLazyInit {
  Object Compute() { ... }
  Object value = null;

  public Object Get() {
    if (value == null) value = Compute();
    return value;
  }
}
```

Figure 7: Unlocked lazy initialization

The first two categories comprise false alarms, in which a warning was reported but there is no race. **Category A:** Different threads modified a variable at different times, using no common lock, but the accesses were serialized by the use of fork/join-type thread operations. Seeing these false alarms motivated us to add threadset analysis to RaceTrack, which suppresses these warnings. **Category B:** The program defined its own locking protocol. One example involved buffers that were passed from pool to pool, with buffer accesses protected by the lock of the buffer's current pool. This obeys neither the lockset nor the fork/join discipline. These warnings could be suppressed by annotations.

The next five categories comprise races that are real, but benign. **Category C:** An inconsequential value such as a performance counter was read without any lock to protect against mutations. **Category D:** Write accesses to a variable were protected by a lock, read accesses used no lock, reads and writes were atomic, and the code was designed so that the variable contained the total representation of the state. **Category E:** The implicated code was performing double-checked locking [29]. **Category F:** The implicated code was managing a cache. Typically mutators excluded each other using a lock while readers ran right in with no lock. The code tended to be grossly perilous. Fortunately, a cache is permitted to forget answers it used to know, as long as it never gives the wrong answer. We spent two days analyzing one well-commented example, finally concluding that the cache would never give the wrong answer, but not for the reasons stated in the comments, which were all false. **Category G:** Two examples of correct lock-free code in the producer/consumer style.

Category H: The implicated code was performing unlocked lazy initialization (see Figure 7). This is similar to double-checked locking except there is no lock. It is benign

```
 1.  class StringBuilder {
 2.    thread_id owner = 0;
 3.    String buffer = "";
 4.
 5.    String GetBuffer() {
 6.      String s = buffer;
 7.      if (owner == me) return s;
 8.      return Copy(s);
 9.    }
10.    void ReplaceBuffer(String s) {
11.      owner = me;
12.      buffer = s;
13.    }
14.    public Mutate {
15.      String s = GetBuffer();
16.      Mutate s;
17.      ReplaceBuffer(s);
18.    }
19.    public String ToString() {
20.      String s = buffer;
21.      if (owner == me || owner == 0) {
22.        owner = 0;
23.        return s;
24.      }
25.      return Copy(s);
26.    }
27.  }
```

Figure 8: StringBuilder code sketch (has bug)

```
 1.  class Attributes {
 2.    int flags = 0;
 3.    bool CalcA () { ... }
 4.    bool CalcB () { ... }
 5.
 6.    // We are not protecting against a race.
 7.    // If there is a race while setting flags we
 8.    // will have to compute the result again, but
 9.    // we will always return the correct result.
10.    public bool IsA () {
11.      if ((flags & 0x01) == 0) {
12.        flags |= CalcA() ? 0x03 : 0x01;
13.      }
14.      return (flags & 0x02) != 0;
15.    }
16.    public bool IsB () {
17.      if ((flags & 0x10) == 0) {
18.        flags |= CalcB() ? 0x30 : 0x10;
19.      }
20.      return (flags & 0x20) != 0;
21.    }
22.  }
```

Figure 9: Attributes code sketch (has bug)

provided that the rest of the program does not depend on the specific object being returned. Of course, verifying this requires checking the entirety of the program, which may be impossible for library routines. We found one example where in fact it did depend on it, so that one got categorized as a bug. After discussions with developers, it was decided to treat all of them as bugs.

Category I: We could not figure out what was going on. These examples involved code that was being invoked in strange ways, such as via RPC callback or asynchronous notification callback, and the stack trace told us nothing. Seeing these cases motivated us to add subsequent-thread reporting to RaceTrack, so that more evidence could be compiled to figure out what was going on.

Category J: A real race that affected correctness. Some of these bugs were more serious than others, and some had already been found and fixed in the production system. But several were previously unknown. Below we describe three of the more interesting bugs in more detail. These three bugs had existed for years in the code base and in fact are present in the 2002 Shared Source Common Language Infrastructure (SSCLI) 1.0 Release [20].

4.1.1 StringBuilder Bug

In the CLR, character strings are represented as objects of the built-in `String` class. Motivated by the view that a character string is a constant, `String` objects are defined to be immutable. This fact can then be exploited by, for example, security code that examines a user's request to open a certain file. The security code can inspect the string file name directly without needing first to take a safe copy.

The CLR provides the `StringBuilder` class to make efficient the common task of building up a large string by editing together many smaller strings. For maximum efficiency, the `StringBuilder` records its intermediate state in the same representation as a `String`, so that when it comes time to produce the final result, a `String` object can be delivered immediately. Such a design is perilous in the presence of multi-threading. If one thread can deliver the `String` object while another is still modifying it, the appearance of immutability will be lost. Since it was considered too inefficient to add locking to preclude such a rare case, a lock-free design was adopted. Figure 8 gives a sketch of the code.

The basic idea of the lock-free design is to force each mutation to operate on a data structure private to its thread. The order of memory access to `owner` and `buffer` in lines 6-7, 11-12, and 20-21 is crucial. Unfortunately, there is a bug. It takes three threads to demonstrate. Let thread 1 pause before line 22. Then let thread 2 call *Mutate* twice, pausing the second time before line 16. Then let thread 3 call `ToString` and pause before line 21. If thread 1 now proceeds, it clears the owner, which allows thread 3 to return the buffer that thread 2 is about to mutate. The error was allowing `owner == 0` to be acceptable in line 21. Without this clause, the code would be correct. This very subtle bug escaped years of code review.

Source code of this bug may be seen in file `clr/src/bcl/system/text/stringbuilder.cs` of SSCLI 1.0 [20].

4.1.2 Attributes Bug

In the CLR, the protocol used to invoke a remote method varies depending on various attributes of the method, such as whether the method is "one way", whether the method is overloaded, and so on. These attributes can be computed by examining the method's signature. For efficiency, the attributes are not computed until they are actually needed and, once computed, they are saved so that they need not be computed again. Figure 9 shows a sketch of the code.

Two flags are used for each attribute: a "value" flag to hold the value of the attribute and a "valid" flag to indicate that the value is valid. No locks are used, so races are clearly an issue. The intriguing comment on lines 6-9 is quoted directly from the product code. Unfortunately, the comment is false. The problem is that all flags are stored in the same variable, the update operation (lines 12 and 18) is not atomic, and the valid and value flags are read separately (lines 11, 14 and 17, 20). By appropriately interleaving two

program	Boxwood				SATsolver				SpecJBB				Crawler				Vclient			
lines of code	8579				10,883				31,405				7246				165,192			
active threads	10				1				various				19				69			
	slowdown (sec)	ratio	memory (MB)	ratio	slowdown (sec)	ratio	memory (MB)	ratio	slowdown (ops/s)	ratio	memory (MB)	ratio	slowdown (pages)	ratio	memory (MB)	ratio	slowdown (%cpu)	ratio	memory (MB)	ratio
no RaceTrack	312	1.00	11.5	1.00	713	1.00	102	1.00	19174	1.00	373	1.00	2364	1.00	63.9	1.00	6.4	1.00	63.9	1.00
lockset	366	1.17	15.4	1.34	1974	2.77	170	1.67	6732	2.85	655	1.76	2189	1.08	84.8	1.33	12.5	1.95	74.4	1.17
+threadset	407	1.30	16.5	1.43	2123	2.98	222	2.18	6678	2.87	752	2.02	2214	1.07	108.0	1.69	12.8	2.00	75.6	1.19
+granularity	378	1.21	11.9	1.03	1822	2.56	155	1.52	6029	3.18	441	1.18	2212	1.07	65.0	1.02	12.8	2.00	74.7	1.18

Table 2: Performance impact of RaceTrack settings on real-world programs (August 2005)

threads, it is possible to return a value that is not valid. This subtle bug escaped years of code review.

Source code of this bug may be seen in class `RemotingMethodCachedData` in file `clr/src/bcl/system/runtime/remoting/remotingattributes.cs` of SSCLI 1.0 [20]. The SSCLI version omits the intriguing comment.

4.1.3 ClearCache Bug

In the CLR, many modules use caches to improve their performance on repeated calls. There are an enormous number of such caches: for example, every field and method potentially has an associated cache of reflection data. However, when the program is under memory pressure, it would be appropriate to clear these caches. To provide for this, the garbage collector implements a memory pressure callback registry. Each cache, when it is created, registers a callback routine with the garbage collector. When the garbage collector suffers from memory pressure, it removes the callback routines from the registry and invokes them each in turn. Caches that continue to exist have to re-register their callback routines.

Now, for managing a set of callback routines, the CLR has a good primitive known as a multicast delegate. A multicast delegate value is immutable, but a callback can be "added" to an existing value producing a new value, and such values can be stored in a multicast delegate variable. Invoking a multicast delegate causes each of the component callbacks to be invoked in turn. All of these operations are guaranteed "thread-safe." The garbage collector used this primitive to implement its memory pressure callback registry.

Unfortunately, although multicast delegate operations are "thread-safe," updating the multicast delegate variable is not atomic. So if separate threads try to add callbacks at the same time, one of the callbacks might get forgotten. Forgetting a callback means that the associated cache would never be cleared. The effect would look like a memory leak. Since there are an enormous number of caches, it's not uncommon for multiple threads to race on the registry. This bug had recently been detected by the product group and was fixed in the new release.

Source code of this bug may be seen in file `clr/src/bcl/system/gc.cs` of SSCLI 1.0 [20]. The code actually performs the multicast delegate operation inside a user-defined event handler. Default CLR event handlers are "synchronized," but user-defined event handlers must provide their own locking, which was overlooked.

4.2 Performance Impact on Large Programs

We measured the performance impact of RaceTrack on several large programs of different types. For comparison, we ran each program under each of the following four settings. Setting "no RaceTrack" is the original, unmodified CLR. Setting "lockset" implements lockset analysis. Four bytes of RaceTrack state per monitored field or array element are added to the end of each object or array. Setting "+threadset" adds adaptive threadset. Eight bytes of RaceTrack state per monitored field or array element are added to the end of each object or array. Setting "+granularity" adds adaptive granularity. Eight bytes of RaceTrack state are added onto the end of each object or array (but only if it contains a monitored field or array element). If the object is refined to field-level granularity, an additional eight bytes per monitored field or array element will be allocated.

The monitored programs are as follows. **Boxwood** is a simulation of a distributed BTree implementation [16] performing 100,000 random insertions and deletions. It uses multiple threads, fine-grained locking, and lots of inter-thread notification. **SATsolver** is an analysis of an unsatisfiable statement containing 20,000 variables in 180,000 clauses, using a C# version of a boolean satisfiability solver. It is single-threaded. Most of its execution consists of accessing tiny arrays of about six elements. **SpecJBB** is a C# version of a standard Java business benchmark [31] that runs twelve successive two-minute simulations and reports a benchmark throughput in operations per second. **Crawler** is a prototype multithreaded web crawler that runs for five minutes and reports the number of pages downloaded from the live Internet. **Vclient** is a video client displaying thirty minutes of MTV streaming from a server on the local network. It has real-time deadlines and performs extensive buffering and scheduling using many threads.

For each test program we measured slowdown and peak memory usage under each setting. For Boxwood and SATsolver slowdown was based on elapsed runtime, for SpecJBB on its reported throughput, for Crawler on the number of pages downloaded, and for Vclient on average CPU load. Tests were repeated several times to give an average measurement. For convenience, we also show each measurement as a ratio relative to the no-RaceTrack case. Table 2 gives the results. The programs were not modified in any way. Note that the number of lines of code does not count any of the code in the .NET library, which is quite vast and all of which is monitored by RaceTrack.

With respect to slowdown, generally the test programs suffered when going from no RaceTrack to lockset, then suffered further with the addition of threadset, and then stayed about the same with the addition of adaptive granularity. The additional suffering with the addition of threadset can be explained by the additional work required to manage the

	Boxwood	SATsolver	SpecJBB	Crawler
thread clock ticks	245,807	0	239	12
objects allocated (K)	30,383	3,054	23,093	3,635
objects refined	1,161	0	3,399	120
monitored accesses (M)	1,692	28,183	6,615	997
... at object-level (M)	1,557	28,183	5,997	994
... ... in ***Exclusive0*** (M)	984	28,183	4,550	917

Table 3: Behavior under adaptive granularity

threadsets. Adaptive granularity does not take significant additional work.

With respect to memory usage, generally the test programs suffered significantly when going from no RaceTrack to lockset-only RaceTrack, suffered significantly again with the addition of threadset, and then dramatically improved with the addition of adaptive granularity. In many cases adaptive granularity reduced the memory overhead to only a few percent. The memory overhead for SATsolver is explained by the fact that most of its objects are tiny arrays containing about six elements. Since SATsolver's execution is completely dominated by accesses to its innumerable tiny arrays, its slowdown ratios reflect the impact of inserting a RaceTrack helper call for each access.

It was difficult to get Vclient to behave consistently from run to run. Presumably depending on the characteristics of the video stream, Vclient would alter its CPU load from 1% to 80% over ten-second intervals. Presumably depending on its estimate of buffering requirements, Vclient would alter its memory usage by several megabytes. The main point of this test was to show that RaceTrack could handle a seriously large and long-running program. The developers were quite interested in the race reports.

We added performance counters to investigate the internal actions of several of the test programs under adaptive granularity, with the results shown in Table 3. (Including these counters degrades performance significantly, so they are not present in the data in Table 2.) Boxwood's use of notification is clearly reflected in the number of thread clock ticks that it performs. The enormous ratio between the number of objects allocated and the number of objects refined to field-level granularity shows why adaptive granularity does so well at saving memory. Looking at dynamic behavior, the vast majority of monitored accesses occur at object-level granularity and, of those, the majority are to an object in ***Exclusive0*** state, indicating that the object has never been shared. This shows the benefit of having a state optimized for this case.

Summarizing, RaceTrack's performance impact was a slowdown of about 3x on memory-intensive programs and typically much less than 2x on others, and a memory usage ratio typically less than 1.2x. Adaptive granularity proved to be highly effective at reducing memory overhead and also tended to improve the slowdown somewhat.

5. CONCLUSIONS

In this paper, we have described RaceTrack, a dynamic data race detector for programs executing in the Common Language Runtime environment. RaceTrack monitors the entire execution of a program transparently by modifying the virtual machine to instrument memory accesses and synchronization operations. To improve accuracy, RaceTrack uses a hybrid detection algorithm that supports both lock-based synchronization and fork-join parallelism, and monitors memory accesses at the granularity of individual object fields and array elements. To reduce overhead, RaceTrack uses a novel adaptive approach that dynamically adjusts both the detection granularity as well as the amount of access history maintained for each detection unit. A post-mortem analyzer provides much-needed help for programmers to better understand the causes of the race warnings.

We have implemented RaceTrack in the current production version of the CLR. Experimental results suggest that RaceTrack reports accurate warnings for a wide variety of applications; taking the CLR regression test suite for example, only a small number of the warnings were false positives, and RaceTrack identified several serious bugs that have escaped many rounds of serious code review. Additionally, we have shown that dynamic race detection can be done in a non-intrusive manner that imposes only a small overhead, despite the fact that RaceTrack performs detection at a fine granularity and monitors both user and library code. We have also demonstrated the scalability of RaceTrack by running it on several real-world applications. We believe that RaceTrack is ready for prime time.

Of course, implementing RaceTrack inside the CLR is not without its cost. Modern virtual machines are very complex and sensitive to the kind of modifications made by RaceTrack. Considerable effort was invested to gain a good understanding of the CLR implementation. We initially prototyped RaceTrack using the SSCLI, and experience learned from the prototype was highly valuable when we later ported RaceTrack to the production CLR. One alternative we considered was to use a profiling interface supported by the CLR. But we dismissed it because (a) we could not see how to make certain RaceTrack modifications such as those in the garbage collector and (b) many RaceTrack modifications could be best and naturally implemented by direct modifications to the CLR. Overall, we believe that our implementation strategy was the right choice.

There is plenty of future work to do. Some optimizations remain that we believe could further improve RaceTrack's performance. A useful extension would be to incorporate deadlock detection into the RaceTrack framework by monitoring the pattern of lock acquisitions. We believe that static analysis and annotation would play a key role in reasoning about for complex code patterns such as lock-free data structures. We plan to investigate such techniques and their combinations with RaceTrack to further reduce false positives.

Static race detection tools [10, 14, 32] have shown great promise. It is thus interesting to look into the possibility of combining them with dynamic tools like RaceTrack. A simple combination would be to use static tools to identify the set of objects and field accesses for RaceTrack to monitor. Warnings confirmed by RaceTrack would be considered to be more likely. Furthermore, compile-time analysis could be used to identify fields and accesses unnecessary for RaceTrack to track. For example, RaceTrack can safely ignore a field that can be determined statically to be properly protected by locks.

6. ACKNOWLEDGMENTS

We would like to thank Mike Burrows for valuable discussions. Roy Levin and Leslie Lamport supplied valuable comments on an earlier draft of this paper, and we would also like to thank the anonymous reviewers for their comments. Thanks to Dennis Fetterly for running his web crawler many times under different versions of RaceTrack. And special thanks to Jan Kotas, Michal Cierniak, Christopher Brumme, Anthony Moore, and the rest of the CLR team for taking the time to help us understand the Common Language Runtime and giving advice on how to make our changes.

7. REFERENCES

[1] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer. Detecting data races on weak memory systems. In *Proceedings of the 18th Annual International Symposium on Computer Architecture (ISCA)*, pages 234–243, May 1991.

[2] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (OOPSLA)*, Oct. 2001.

[3] A. Broder, S. Glassman, M. Manasse, and G. Zweig. Syntactic clustering of the web. In *Proceedings of the 6th International World Wide Web Conference*, 1997.

[4] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise data race detection for object oriented programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 285–297, 2002.

[5] J.-D. Choi, B. P. Miller, and R. H. B. Netzer. Techniques for debugging parallel programs with flowback analysis. *ACM Transactions on Programming Languages and Systems*, 13(4):491–530, October 1991.

[6] M. Christiaens and K. De Bosschere. TRaDe, a topological approach to on-the-fly race detection in Java programs. In *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM)*, Apr. 2001.

[7] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 1–10, 1990.

[8] A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. In *PADD '91: Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging*, pages 85–96, New York, NY, USA, 1991. ACM Press.

[9] Ecma International. Standard ECMA-335: Common language infrastructure (CLI), 2002. `http://www.ecma-international.org/publications/standards/Ecma-335.htm`.

[10] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 237–252, October 2003.

[11] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 219–232, 2000.

[12] J. Gough and K. Gough. *Compiling for the .NET Common Language Runtime*. Prentice Hall PTR, 2001.

[13] J. J. Harrow. Runtime checking of multithreaded applications with visual threads. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 331–342, London, UK, 2000. Springer-Verlag.

[14] T. Henzinger, R. Jhala, and R. Majumder. Race checking by context inference. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2004.

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

[16] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 105–120, Dec. 2004.

[17] F. Mattern. Virtual time and global states of distributed systems. In C. M. et al., editor, *Proc. Workshop on Parallel and Distributed Algorithms*, pages 215–226, North-Holland / Elsevier, 1989.

[18] J. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of Supercomputing*, November 1991.

[19] Microsoft Corporation. Basic class library communities. `http://msdn.microsoft.com/netframework/programming/classlibraries/`.

[20] Microsoft Corporation. Shared source common language infrastructure 1.0 release, Nov. 2002. `http://msdn.microsoft.com/net/sscli`.

[21] H. Nishiyama. Detecting data races using dynamic escape analysis based on read barrier. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium (VM)*, May 2004.

[22] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 167–178, 2003.

[23] D. Perković and P. J. Keleher. Online data-race detection via coherency guarantees. In *The Second Symposium on Operating Systems Design and Implementation (OSDI)*, pages 47–57, Oct. 1996.

[24] E. Pozniansky and A. Schuster. Efficient on-the-fly race detection in multithreaded C++ programs. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2003.

[25] C. Praun and T. Gross. Object race detection. In *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (OOPSLA)*, pages 70–82, 2001.

[26] M. O. Rabin. Fingerprinting by random polynomials. Report TR–15–81, Department of Computer Science, Harvard University, 1981.

[27] M. Ronsse and K. De Bosschere. RecPlay: A fully integrated practical record/replay system. *ACM Transactions on Computer Systems*, 17(2):133–152, May 1999.

[28] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[29] D. C. Schmidt and T. Harrison. Double-checked locking: An optimization pattern for efficiently initializing and accessing thread-safe objects. In M. Buschmann and D. Riehle, editors, *Pattern Languages of Program Design 3*. Addison-Wesley, Reading, MA, 1997.

[30] E. Schonberg. On-the-fly detection of access anomalies. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 285–297, 1989.

[31] Standard Performance Evaluation Corporation. SPEC JBB2000. `http://www.spec.org/jbb2000/`.

[32] N. Sterling. Warlock: A static data race analysis tool. In *Proceedings of USENIX Winter Technical Conference*, January 1993.

[33] Valgrind project. Helgrind: a data-race detector, 2005. `http://valgrind.org/docs/manual/hg-manual.html`.