# Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs

Stefan Savage*
Department of Computer Science and Engineering
University of Washington, Seattle

Michael Burrows   Greg Nelson   Patrick Sobalvarro
Digital Equipment Corporation
Systems Research Center

Thomas Anderson
Computer Science Division
University of California, Berkeley

## Abstract

Multi-threaded programming is difficult and error prone. It is easy to make a mistake in synchronization that produces a data race, yet it can be extremely hard to locate this mistake during debugging. This paper describes a new tool, called Eraser, for dynamically detecting data races in lock-based multi-threaded programs. Eraser uses binary rewriting techniques to monitor every shared memory reference and verify that consistent locking behavior is observed. We present several case studies, including undergraduate coursework and a multi-threaded Web search engine, that demonstrate the effectiveness of this approach.

## 1 Introduction

Multi-threading has become a common programming technique. Most commercial operating systems support threads, and popular applications like Microsoft Word and Netscape Navigator are multi-threaded.

Unfortunately, debugging a multi-threaded program can be difficult. Simple errors in synchronization can produce timing-dependent data races that can take weeks or months to track down. For this reason, many programmers have resisted using threads. The difficulties with using threads are well summarized by John Ousterhout in his 1996 USENIX presentation "Why Threads are a bad idea (for most purposes)"[Ousterhout 96].

*savage@cs.washington.edu

In this paper we describe a tool, called *Eraser*, that dynamically detects data races in multi-threaded programs. We have implemented Eraser for DIGITAL Unix and used it to detect data races in a number of programs, ranging from the AltaVista Web search engine to introductory programming exercises written by undergraduates.

Previous work in dynamic race detection is based on Lamport's *happens-before* relation[Lamport 78] and checks that conflicting memory accesses from different threads are separated by synchronization events. Happens-before algorithms handle many styles of synchronization, but this generality comes at a cost. We have aimed Eraser specifically at the lock-based synchronization used in modern multi-threaded programs. Eraser simply checks that all shared memory accesses follow a consistent *locking discipline*. A locking discipline is a programming policy that ensures the absence of data races. For example, a simple locking discipline is to require that every variable shared between threads is protected by a mutual exclusion lock. We will argue that for many programs Eraser's approach of enforcing a locking discipline is simpler, more efficient, and more thorough at catching races than the approach based on *happens-before*. As far as we know, Eraser is the first dynamic race detection tool to be applied to multi-threaded production servers.

The remainder of this paper is organized as follows: After reviewing what a data race is and describing previous work in race detection, we present the Lockset algorithm used by Eraser, first at a high level and then at a level low enough to reveal the main performance-critical implementation techniques. Finally, we describe the experience we have had using Eraser with a number of multi-threaded programs.

Eraser bears no relationship to the tool by the same name constructed by John Mellor-Crummey for detecting data races in shared-memory parallel Fortran programs as part of the ParaScope Programming Environment[Mellor-Crummey 93].

## 1.1 Definitions

A *lock* is a simple synchronization object used for mutual exclusion; it is either *available*, or *owned* by a thread. The operations on a lock `mu` are `lock(mu)` and `unlock(mu)`. Thus it is essentially a binary semaphore used for mutual exclusion, but differs from a semaphore in that only the owner of a lock is allowed to release it.

A *data race* occurs when two concurrent threads access a shared variable, and:

- at least one access is a write, and
- the threads use no explicit mechanism to prevent the accesses from being simultaneous.

If a program has a potential data race, then the effect of the conflicting accesses to the shared variable will depend on the interleaving of the thread executions. Although programmers occasionally deliberately allow a data race when the non-determinism seems harmless, usually a potential data race is a serious error caused by failure to synchronize properly.

## 1.2 Related work

An early attempt to avoid data races was the pioneering concept of a *monitor* introduced by C.A.R. Hoare [Hoare 74]. A monitor is a group of shared variables together with the procedures that are allowed to access them, all bundled together with a single anonymous lock that is automatically acquired and released at the entry and exit of the procedures. The shared variables in the monitor are out of scope (that is, invisible) outside the monitor, consequently they can be accessed only from within the monitor's procedures, where the lock is held. Thus monitors provide a static, compile-time guarantee that accesses to shared variables are serialized and therefore free from data races. Monitors are an effective way to avoid data races if all shared variables are static globals, but they don't protect against data races in programs with dynamically allocated shared variables, a limitation that early users found was significant[Lampson & Redell 80]. By substituting dynamic checking for static checking, our work aims to allow dynamically allocated shared data while retaining as much of the safety of monitors as possible.

Some attempts have been made to create purely static (that is, compile-time) race detection systems that work in the presence of dynamically allocated shared data: for example, Sun's `lock_lint` [SunSoft 94] and the Extended Static Checker for Modula-3 [Detlefs et al. 97, Nelson et al. 96]. But these approaches seem problematical since they require statically reasoning about the program's semantics.

Most of the previous work in dynamic race detection has been carried out by the scientific parallel programming community [Dinning & Schonberg 90, Netzer 91, Mellor-Crummey 91, Perkovic & Keleher 96] and is based on Lamport's *happens-before* relation, which we now describe.

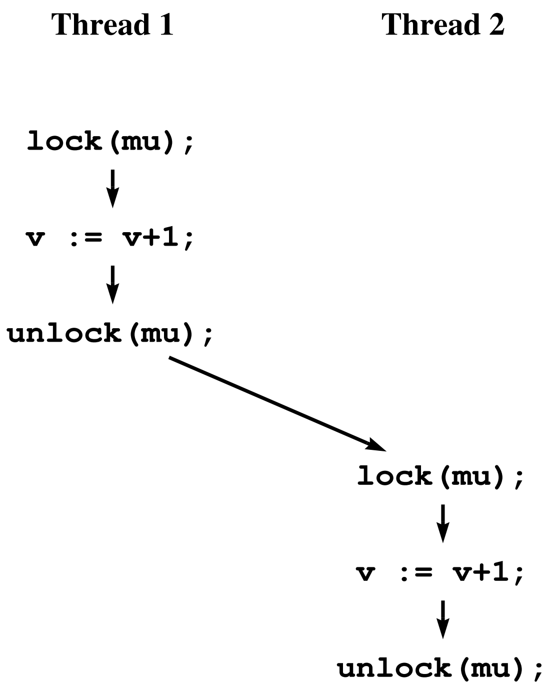


Figure 1: Lamport's *happens-before* orders events in the same thread in temporal order, and orders events in different threads if the threads synchronized with one another between the events.

The *happens-before* order is a partial order on all events of all threads in a concurrent execution. Within any single thread, events are ordered in the order in which they occurred. Between threads, events are ordered according to the properties of the synchronization objects they access. If one thread accesses a synchronization object and the next access to the object is by a different thread, then the first access is defined to *happen before* the second if the semantics of the synchronization object forbid a schedule in which these two interactions are exchanged in time. For example, Figure 1 shows one possible ordering of two threads executing the same code segment. The three program statements executed by Thread 1 are ordered by *happens-before* because they are executed sequentially in the same thread. The lock of `mu` by Thread 2 is ordered by *happens-before* with the unlock of `mu` by Thread 1 because a lock cannot be acquired before its previous owner has released it. Finally, the three statements executed by Thread 2 are ordered by *happens-before* because they are executed sequentially within that thread.

If two threads both access a shared variable and the accesses are not ordered by the *happens-before* relation, then in another execution of the program in which the slower thread ran faster and/or the faster thread ran slower, the two accesses could have happened simultaneously; that is, a data race could have occurred, whether or not it actually did occur. All previous dynamic race detection tools that we know of are based on this observation. These race detectors monitor every data reference and synchronization operation and check for conflicting accesses to shared variables that are unrelated by the *happens-before* relation for the particular execution they are monitoring.

Unfortunately, tools based on *happens-before* have two significant drawbacks. First, they are difficult to implement efficiently because they require per-thread information about concurrent accesses to each shared memory location. More importantly, the effectiveness of tools based on *happens-before* is highly dependent on the interleaving produced by the scheduler. Figure 2 shows a simple example where the *happens-before* approach can miss a data race. While there is a potential data race on the unprotected accesses to `y`, it will not be detected in the execution shown in the figure, because Thread 1 holds the lock before Thread 2, and so the accesses to `y` are ordered in this interleaving by *happens-before*. A tool based on *happens-before* would detect the error only if the scheduler produced an interleaving in which the fragment of code for Thread 2 occurred before the fragment of code for Thread 1. Thus, to be effective, a race detector based on *happens-before* needs a large number of test cases to test many possible interleavings. In contrast, the programming error in Figure 2 will be detected by Eraser with any test case that exercises the two code paths, because the paths violate the locking discipline for `y` regardless of the interleaving produced by the scheduler. While Eraser is a testing tool and therefore cannot guarantee that a program is free from races, it can detect more races than tools based on *happens-before*.

The *lock covers* technique of Dinning and Shonberg is an improvement to the happens-before approach for programs that make heavy use of locks[Dinning & Schonberg 91]. Indeed, one way to describe our approach would be that we extend Dinning and Shonberg's improvement and discard the underlying happens-before apparatus that they were improving.

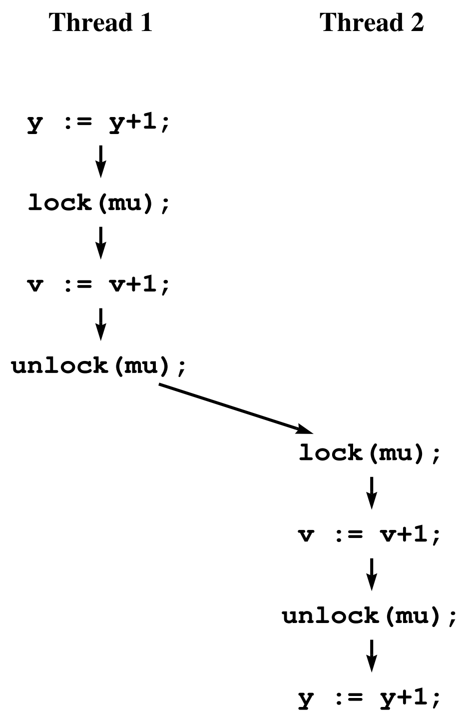


Figure 2: The program allows a data race on `y`, but the error is not detected by *happens-before* in this execution interleaving.

## 2 The Lockset algorithm

In this section we describe how the Lockset algorithm detects races. The discussion is at a fairly high level; the techniques used to implement the algorithm efficiently will be described in the following section.

The first and simplest version of the Lockset algorithm enforces the simple locking discipline that every shared variable is protected by some lock, in the sense that the lock is held by any thread whenever it accesses the variable. Eraser checks whether the program respects this discipline by monitoring all reads and writes as the program executes. Since Eraser has no way of knowing which locks are intended to protect which variables, it must infer the protection relation from the execution history.

For each shared variable $v$, Eraser maintains the set $C(v)$ of *candidate locks* for $v$. This set contains those locks that have protected $v$ for the computation so far. That is, a lock $l$ is in $C(v)$ if in the computation up to that point, every thread that has accessed $v$ was holding $l$ at the moment of the access. When a new variable $v$ is initialized, its candidate set $C(v)$ is considered to hold all possible locks. When the variable is accessed, Eraser updates $C(v)$ with the intersection of $C(v)$ and the set of locks held by the current thread. This process, called *lockset refinement*, ensures that any lock that consistently protects $v$ is contained in $C(v)$. If some lock $l$ consistently protects $v$, it will remain in $C(v)$ as $C(v)$ is refined. If $C(v)$ becomes empty this indicates that there is no lock that consistently protects $v$.

In summary, here is the first version of the Lockset algorithm:

Let $locks\_held(t)$ be the set of locks held by thread $t$.

For each $v$, initialize $C(v)$ to the set of all locks.

On each access to $v$ by thread $t$,
  set $C(v) := C(v) \cap locks\_held(t)$;
  if $C(v) = \{\}$, then issue a warning.

Figure 3 illustrates how a potential data race is discovered through lockset refinement. The left column contains program statements, executed in order from top to bottom. The right column reflects the set of candidate locks, $C(v)$, after each statement is executed. This example has two locks, so $C(v)$ starts containing both of them. After `v` is accessed while holding `mu1`, $C(v)$ is refined to contain that lock.

| *Program* | *locks_held* | *C(v)* |
|---|---|---|
| | `{}` | `{mu1,mu2}` |
| `lock(mu1);` | | |
| | `{mu1}` | |
| `v := v+1;` | | |
| | | `{mu1}` |
| `unlock(mu1);` | | |
| | `{}` | |
| `lock(mu2);` | | |
| | `{mu2}` | |
| `v := v+1;` | | |
| | | `{}` |
| `unlock(mu2);` | | |
| | `{}` | |

Figure 3: If a shared variable is sometimes protected by lock `mu1` and sometimes by lock `mu2`, then no lock protects it for the whole computation. The figure shows the progressive refinement of the set of candidate locks $C(v)$ for $v$. When $C(v)$ becomes empty, the Lockset algorithm has detected that no lock protects $v$.

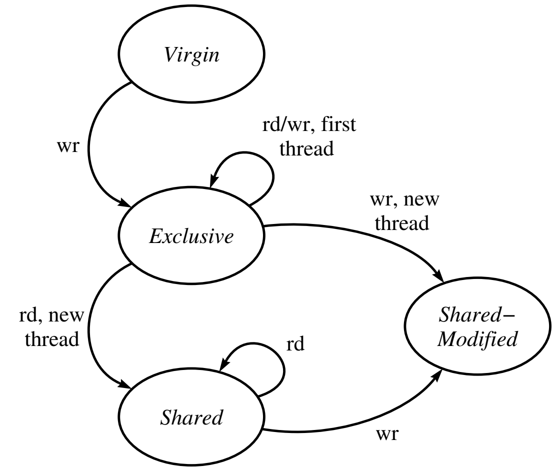


Figure 4: Eraser keeps track of the state of all locations in memory. Newly allocated locations begin in the *Virgin* state. As various threads read and write a location, its state changes according to the transitions in the figure. Race conditions are reported only for locations in the *Shared-Modified* state.

Later, $v$ is accessed again, with only `mu2` held. The intersection of the singleton sets {`mu1`} and {`mu2`} is the empty set, correctly indicating that no lock protects $v$.

## 2.1 Improving the locking discipline

The simple locking discipline we have used so far is too strict. There are three very common programming practices that violate the discipline yet are free from any data races:

- *Initialization.* Shared variables are frequently initialized without holding a lock.
- *Read-shared data.* Some shared variables are written during initialization only and are read-only thereafter. These can be safely accessed without locks.
- *Read-write locks.* Read-write locks allow multiple readers to access a shared variable, but allow only a single writer to do so.

We will extend the Lockset algorithm to accommodate initialization and read-shared data, and then extend it further to accommodate read-write locks.

## 2.2 Initialization and read-sharing

There is no need for a thread to lock out others if no other thread can possibly hold a reference to the data being accessed. Programmers often take advantage of this observation when initializing newly allocated data. To avoid false alarms caused by these unlocked initialization writes, we delay the refinement of a location's candidate set until after it has been initialized. Unfortunately, we have no easy way of knowing when initialization is complete. Eraser therefore considers a shared variable to be initialized when it is first accessed by a second thread. As long as a variable has been accessed by a single thread only, reads and writes have no effect on the candidate set.

Since simultaneous reads of a shared variable by multiple threads are not races, there is also no need to protect a variable if it is read-only. To support unlocked read-sharing for such data, we report races only after an initialized variable has become write-shared by more than one thread.

Figure 4 illustrates the state transitions that control when lockset refinement occurs and when races are reported. When a variable is first allocated, it is set to the *Virgin* state, indicating that the data is new and has not yet been referenced by any thread. Once the data is accessed, it enters the *Exclusive* state, signifying that it is has been accessed, but by one thread only. In this state, subsequent reads and writes by the same thread do not change the variable's state and do not update $C(v)$. This addresses the initialization issue, since the first thread can initialize the variable without causing $C(v)$ to be refined. When and if another thread accesses the variable, then the state changes. A read access changes the state to *Shared*. In the *Shared* state, $C(v)$ is updated, but data races are not reported, even if $C(v)$ becomes empty. This takes care of the read-shared data issue, since multiple threads can read a variable without causing a race to be reported. A write access from a new thread changes the state from *Exclusive* or *Shared* to the *Shared-Modified* state, in which $C(v)$ is updated and races are reported, just as described in the original, simple version of the algorithm.

Our support for initialization makes Eraser's checking more dependent on the scheduler than we would like. Suppose that a thread allocates and initializes a shared variable without a lock, and erroneously makes the variable accessible to a second thread before it has completed the initialization. Then Eraser will detect the error if any of the second thread's accesses occur before the first thread's final initialization actions, but otherwise Eraser will miss the error. We don't think this has been a problem, but we have no way of knowing for sure.

### 2.3 Read-write locks

Many programs use single-writer, multiple-reader locks as well as simple locks. To accommodate this style we introduce our last refinement of the locking discipline: we require that for each variable $v$, some lock $m$ *protects* $v$, meaning $m$ is held in write mode for every write of $v$, and $m$ is held in some mode (read or write) for every read of $v$.

We continue to use the state transitions of Figure 4, but when the variable enters the *Shared-Modified* state, the checking is slightly different:

Let $locks\_held(t)$ be the set of locks held in any mode by thread $t$.

Let $write\_locks\_held(t)$ be the set of locks held in write mode by thread $t$.

For each $v$, initialize $C(v)$ to the set of all locks.

On each read of $v$ by thread $t$,
set $C(v) := C(v) \cap locks\_held(t)$;
if $C(v) = \{\}$, then issue a warning.

On each write of $v$ by thread $t$,
set $C(v) := C(v) \cap write\_locks\_held(t)$;
if $C(v) = \{\}$, then issue a warning.

That is, locks held purely in read mode are removed from the candidate set when a write occurs, as such locks held by a writer do not protect against a data race between the writer and some other reader thread.

## 3 Implementing Eraser

Eraser is implemented for the DIGITAL Unix operating system on the Alpha processor, using the ATOM [Srivastava & Eustace 94] binary modification system. Eraser takes an unmodified program binary as input and adds instrumentation to produce a new binary that is functionally identical, but includes calls to the Eraser runtime to implement the Lockset algorithm.

To maintain $C(v)$, Eraser instruments each load and store in the program. To maintain $lock\_held(t)$ for each thread $t$, Eraser instruments each call to acquire or release a lock, as well as the stubs that manage thread initialization and finalization. To initialize $C(v)$ for dynamically allocated data, Eraser instruments each call to the storage allocator.

Eraser treats each 32-bit word in the heap or global data as a possible shared variable, since on our platform a 32-bit word is the smallest memory-coherent unit. Eraser does not instrument loads and stores whose address mode is indirect off the stack pointer, since these are assumed to be stack references, and shared variables are assumed to be in global locations or in the heap. Eraser will maintain candidate sets for stack locations that are accessed via registers other than the stack pointer, but this is an artifact of the implementation rather than a deliberate plan to support programs that share stack locations between threads.

When a race is reported, Eraser indicates the file and line number at which it was discovered and a backtrace listing of all active stack frames. The report also includes the thread ID, memory address, type of memory access, and important register values such as the program counter and stack pointer. We have found that this information is usually sufficient for locating the source of the race. If the cause of a race is still unclear, the user can direct Eraser to log all the accesses to a particular variable that result in a change to its candidate lock set.

### 3.1 Representing the candidate lock sets

A naïve implementation of lock sets would store a list of candidate locks for each memory location, potentially consuming many times the allocated memory of the program. We can avoid this expense by exploiting the fortunate fact that the number of distinct sets of locks observed in practice is quite small. In fact, we have never observed more than 10,000 distinct sets of locks occurring in any execution of the Lockset monitoring algorithm. Consequently, we represent each set of locks by a small integer, a *lockset index* into a table whose entries represent the set of locks as sorted vectors of lock addresses. Hashing is used to eliminate duplicates in the table and to find a lockset index from a given set of locks. The entries in the table are never deallocated or modified, so each lockset index remains valid for the lifetime of the program. Eraser also caches the result of each intersection, so that the fast case for set intersection is simply a table lookup. Each lock vector in the table is sorted, so that when the cache fails, the slow case of the intersection operation can be performed by a simple comparison of the two sorted vectors.

For every 32-bit word in the data segment and heap, there is a corresponding *shadow word* that is used to contain a 30-bit lockset index and a 2-bit state condition. In the *Exclusive* state, the 30 bits are not used to store a lockset index, but used instead to store the ID of the thread with exclusive access.

All the standard memory allocation routines are instrumented to allocate and initialize a shadow word for each word allocated by the program. When a thread accesses a memory location, Eraser finds the shadow word by adding a fixed displacement to the location's address.

## 3.2 Performance

Performance was not a major goal in our implementation of Eraser; consequently it has many opportunities for optimization. Applications typically slow down by a factor of 10 to 30 while using Eraser. We estimate that half of the slowdown is due to the overhead incurred by making a procedure call at every load and store instruction; which could be eliminated by using a version of ATOM that can inline monitoring code [Scales et al. 96]. Also, there are many opportunities for using static analysis to reduce the overhead of the monitoring code; but we have not explored them.

In spite of our limited performance tuning, we have found that Eraser is fast enough to debug most programs, and therefore meets the most essential performance criteria.

## 3.3 Program annotations

As expected, our experience with Eraser showed that it can produce false alarms. Part of our research was aimed at finding effective annotations to suppress false alarms without accidentally losing useful warnings. This is a key to making a tool like Eraser useful. If the false alarms are suppressed with accurate and specific annotations, then when a program is modified and the modified program is tested, only fresh and relevant warnings will be produced.

In our experience false alarms fell mainly into three broad categories:

- *Memory reuse*. False alarms were reported because memory is reused without resetting the shadow memory. Eraser instruments all of the standard C, C++, and Unix memory allocation routines. However, many programs implement free lists or private allocators, and Eraser has no way of knowing that a privately recycled piece of memory is protected by a new set of locks.

- *Private locks*. False alarms were reported because locks are taken without communicating this information to the Eraser runtime. This was usually caused by private implementations of multiple-reader/single-writer locks, which are not part of the standard `pthreads` interface that Eraser instruments.

- *Benign races*. True data races were found that did not affect the correctness of the program. Some of these were intentional and others were accidental.

For each of these categories, we developed a program annotation to allow users of Eraser to eliminate the false report. For benign races, we added

```
EraserIgnoreOn()
EraserIgnoreOff()
```

which inform the race detector that it shouldn't report any races in the bracketed code. To prevent memory reuse races from being reported, we added

```
EraserReuse(address, size)
```

which instructs Eraser to reset the shadow memory corresponding to the indicated memory range to the *Virgin* state. Finally, the existence of private lock implementations can be communicated by annotating them with

```
EraserReadLock(lock)
EraserReadUnlock(lock)
EraserWriteLock(lock)
EraserWriteUnlock(lock)
```

We found that a handful of these annotations usually suffices to eliminate all false alarms.

## 3.4 Race detection in an OS kernel

We have begun to modify Eraser to detect races in the *SPIN* operating system [Bershad et al. 95]. While we do not yet have results in terms of data races found, we have acquired some useful experience about implementing such a tool at the kernel level, which is different from the user level in several ways.

First, *SPIN* (like many operating systems) often raises the processor interrupt level to provide mutual exclusion to shared data structures accessed by device drivers and other interrupt level code. In most systems, raising the interrupt level to $n$ ensures that only interrupts of priority greater than $n$ will be serviced until the interrupt level is lowered. Raising and then restoring the interrupt level can be used instead of a lock, as follows:

```
level := SetInterruptLevel(n);
(* Manipulate data *)
RestoreInterruptLevel(level);
```

However, unlike locks, a particular interrupt level inclusively protects all data protected by lower interrupt levels. We have incorporated this difference into Eraser by assigning a lock to each individual interrupt level. When the kernel sets the interrupt level to $n$, Eraser treats this operation as if the first $n$ interrupt locks had all been acquired. We expect this technique to allow us to detect races between code using standard locks and code using interrupt levels.

Another difference is that operating systems make greater use of post/wait style synchronization. The most common example is the use of semaphores to synchronize execution between a thread and an I/O device driver. Upon receiving data, the device driver will perform some minimal processing and then use a `V` operation to signal a thread waiting on `P` operation; for example, to wake up a thread waiting for an I/O completion. This can cause problems for Eraser if data is shared between the device driver and the thread. Because semaphores are not "owned" it is difficult for Eraser to infer which data they are being used to protect, leading it to issue false alarms. Systems that integrate thread and interrupt processing [Kleiman & Eykholt 95] may have less trouble with this problem.

# 4 Experience

We calibrated Eraser on a number of simple programs that contained common synchronization errors (e.g. forgot to lock, used the wrong lock, etc.) and versions of those programs with the errors corrected. While programming these tests, we accidentally introduced a race, and encouragingly, Eraser detected it. These simple tests were extremely useful for finding bugs in Eraser. After convincing ourselves that the tool worked, we tackled some large multi-threaded servers written by experienced researchers at DEC SRC: the HTTP server and indexing engine from AltaVista, the Vesta cache server, and the Petal distributed disk system. We also applied Eraser to some homework problems written by undergraduate programmers.

Eraser found undesirable race conditions in three of the four server programs and in many of the undergraduate homework problems. It also produced false alarms, which we were able to suppress with annotations.

The programmers of the servers on which we tested Eraser did not begin with a plan to test Eraser or even to use Eraser's locking discipline. The fact that Eraser worked well on the servers is evidence that experienced programmers tend to obey the simple locking discipline even in an environment that offers many more elaborate synchronization primitives.

In the remainder of this section we report on the details of our experiences with each program.

## 4.1 AltaVista

We examined two components of the popular AltaVista [Digital Equipment 96a] Web indexing service, `mhttpd` and `Ni2`.

The `mhttpd` program is a lightweight HTTP server designed to support the extremely high server loads experienced by AltaVista. Each search request is handled by a separate thread and relies on locking to synchronize access by concurrent requests to shared data structures. In addition, `mhttpd` employs several additional threads to manage background tasks such as configuration and name cache management. The server consists of approximately 5000 lines of C source code. We tested `mhttpd` by invoking a series of test scripts from three separate Web browsers. The `mhttpd` test used approximately 100 distinct locks that formed approximately 250 different lock sets.

The `Ni2` indexing engine is used to look up information in response to index queries. Index data structures are shared among all of the server threads and explicitly use locks to guarantee that updates are serialized. The basic `Ni2` libraries contain approximately 20,000 lines of C source code. We tested `Ni2` separately using a utility called `ft` that submits a series of random requests using a specified number of threads (we used 10). The `ft` test used approximately 900 locks that formed approximately 3600 distinct lock sets.

We found a large number of reported races, most of which turned out to be false alarms. These were primarily caused by memory reuse, followed by private locks and benign races. The benign races found in `Ni2` are particularly interesting, because they exemplify the intentional use of races to reduce locking overhead. For example, consider the following code fragment:

```
/* unlocked hint */
if (p->ip_fp == (NI2_XFILE *) 0) {
   NI2_LOCKS_LOCK (&p->ip_lock);
   if (p->ip_fp == (NI2_XFILE *) 0) {
      p->ip_fp = ni2_xfopen (
                  p->ip_name, "rb");
   }
   NI2_LOCKS_UNLOCK (&p->ip_lock);
}
. . .
```

In this code fragment the `ip_fp` field is tested without a lock held, which creates a data race with other threads that modify the field with the `ip_lock` lock held. The race was deliberately programmed as an optimization to avoid locking overhead in the common case that `ip_fp` has already been set. The program is correct even with the race, since the `ip_fp` field never transitions from non-zero to zero while in the scope of multiple threads, and the program repeats the test inside the lock in case the field tested zero (thus avoiding the race in which two threads find the field zero and both then initialize it).

This kind of code is very tricky. For example, it might seem safe to access the `p->ip_fp` field in the rest of the procedure (the lines replaced by the ellipsis above). But in fact this would be a mistake, because the Alpha's memory consistency model permits processors to see memory operations out of order if there is no intervening synchronization. Although the `Ni2` code was correct, after using Eraser the programmer decided to reprogram this part of it so that its correctness argument was simpler.

We also found a benign race in the `Ni2` test harness program, where multiple threads race on reads and writes to a global variable called `kill_queries`. This variable is initialized to false and is set to true to indicate that all threads should exit. Each thread periodically polls the variable and exits when it is set to true. Other finalization code had similar benign races. To keep the race detector from reporting such races, we used the `EraserIgnoreOn/Off()` annotations. Similarly, `mhttpd` omits locks when periodically updating global configuration data and statistics. These are indeed synchronization errors, but their effect is relatively minor, which is perhaps why they were undetected for so long.

Inserting nine annotations in the `Ni2` library, five in the `ft` test harness, and ten in the `mhttpd` server reduced the number of reported races from more than a hundred to zero.

## 4.2 Vesta cache server

Vesta [Digital Equipment 96b] is an advanced software configuration management system. Configurations are written in a specialized functional language that describes the dependencies and rules used to derive the current state of the software. Partial results, such as "`.o`" files generated by the C compiler, are cached in the Vesta cache server and used by the Vesta builder to create a particular configuration. The cache server consists of approximately 30,000 lines of C++ code. We tested the cache server using the `TestCache` utility that issues a stream of concurrent random requests. The cache server used 10 threads, acquired 26 distinct locks and instantiated 70 different lock sets.

In testing the cache server, Eraser reported a number of races, mostly revolving around three data structures. The first set of races were detected in the code maintaining the fingerprints in cache entries. Because computing a fingerprint can be expensive, the cache server maintains a boolean field in the cache entry recording whether the fingerprint is valid. The fingerprint is computed only if its true value is needed and its current value is invalid. Unfortunately, the boolean was accessed without a protecting lock, in code like this:

```
Combine::XorFPTag::FPVal() {
   if (!this->validFP) {
      /* NamesFP changes this->fp */
      NamesFP(fps, bv, this->fp, imap);
      this->validFP = true;
   }
   return this->fp;
}
```

This is a serious data race, since in the absence of memory barriers the Alpha semantics don't guarantee that the contents of the `validFP` field is consistent with the `fp` field.

Another set of races revolved around free lists in the `CacheS` object. The `CacheS` object maintains a free list of various kinds of log entries. Our first response was to use `EraserReuse()` annotations where elements were allocated off this free list. However, this didn't make all the warnings disappear; calls to flush the log still caused races. Examination revealed that the head of each log was protected by a lock, but not the individual entries. The `Flush` routines lock the head of the log, store its value in a stack variable, set the head to 0, and release the lock. After this they access the individual entries without any locks held, ultimately putting them onto the free list. This is correct because other threads access the log entries with the log head lock held, and threads do not maintain pointers into the log. Consequently, `Flush` effectively makes the data private to the thread in which `Flush` was called. We eliminated the report of these races by moving the `EraserReuse()` annotations to the three `Flush` routines.

Finally, there were several false alarms related to the `TCP_sock` and `SRPC` objects that are used to implement server-side RPCs. The cache server uses a main server thread to wait for incoming RPC requests. Upon receiving a request, this thread passes the current socket and RPC data structures to a worker thread that is responsible for handling the rest of the RPC. Since the main thread and the worker thread will never access the data structures concurrently they don't need to use locks to serialize access. To Eraser this looks like violation of the locking discipline and is flagged as a race. With some effort it would be possible to modify Eraser to recognize this locking discipline, but we were able to achieve the same effect with two `EraserReuse()` annotations.

In total, ten annotations and one bug fix were enough to reduce the race reports from several hundred to zero.

## 4.3 Petal

Petal is a distributed storage system that presents its clients with a huge virtual disk implemented by a cluster of servers and physical disks [Lee & Thekkath 96]. Petal implements a distributed consensus algorithm as well as failure detection and recovery mechanisms. The Petal server is roughly 25,000 lines of C code and we used 64 concurrent worker threads in our tests. We tested Petal using a utility that issued random read and write requests.

We found a number of false alarms caused by a private reader-writer lock implementation. These were easily suppressed using annotations. We also detected a real race in the routine `GMapCh_CheckServerThread()`. This routine is run by a single thread and periodically checks to make sure that the neighboring servers are running. However, in so doing, it reads the `gmap->state` field without holding the `gmapState` lock (that all other threads hold before writing `gmap->state`).

We found two races where global variables containing statistics were modified without locking. These races were intentional, based on the premises that locking is expensive and that the server statistics need to be only approximately correct.

Finally, we found one false alarm that we were unable to annotate away. The function `GmapCh_Write2()` forks a number of threads and passes each a reference to a component of `GmapCh_Write2`'s stack frame. `GmapCh_Write2()` implements a join-like construct to keep the stack frame active until the threads return. But Eraser doesn't re-initialize the shadow memory for each new stack frame; consequently the reuse of the stack memory for different instances of the stack frame resulted in a false alarm.

## 4.4 Undergraduate coursework

As a counterpoint to our experience with mature multithreaded server programs, two of our colleagues used Eraser to examine the kinds of synchronization errors found in the homework assignments produced by their undergraduate operating systems class [Choi & Lewis 97]. We report their results here to demonstrate how Eraser functions with a less sophisticated code base.

The class was required to complete four standard multi-threading assignments. These assignments can be roughly categorized as low-level (build locks from test-and-set), thread-level (build a small threads package), synchronization-level (build semaphores and mutexes), and application-level (producer/consumer-style problems). Each assignment builds on the implementation of the previous assignment. Our colleagues used Eraser to examine each of these assignments for roughly 40 groups; a total of about 100 runnable assignments were turned in (not all groups completed all assignments, some didn't compile, and a few immediately deadlocked). Of these "working" assignments, 10 percent had data races found by Eraser. These were caused by forgetting to take locks, taking locks during writes but not for reads, using different locks to protect the same data structure at different times, and forgetting to re-acquire locks that were released in a loop.

Eraser also reported a false alarm that was triggered by a queue that implicitly protected elements by accessing the queue through locked head and tail fields (much like Vesta's `CacheS` object).

### 4.5 Effectiveness and Sensitivity

Since Eraser uses a testing methodology it cannot prove that a program is free from data races. But we believe that Eraser works well compared to manual testing and debugging, and that Eraser's testing is not very sensitive to the scheduler interleaving. To test these beliefs we performed two additional experiments.

We consulted the program history of `Ni2` and re-introduced two data races that had existed in previous versions. The first error was an unlocked access to a reference count used to garbage collect file data structures. The other race was caused by failing to take an additional and necessary lock needed to protect the data structures of a subroutine called in the middle of a large procedure. These races had existed in the `Ni2` source code for several months before they were manually found and fixed by the program author. Using Eraser, one of us was able to locate both races in several minutes without being given any information about where the races were or how they were caused. It took 30 minutes to correct both errors and verify the absence of further race reports.

We examined the issue of sensitivity by re-running the `Ni2` and `Vesta` experiments but using only two concurrent threads instead of ten. If Eraser was sensitive to differences in thread interleaving then we would expect to find a different set of race reports. In fact, we found the same race reports (albeit sometimes in different order) across multiple runs using using either two threads or ten.

## 5 Additional experience

In this section we briefly touch on two further topics, each of which concerns a form of dynamic checking for synchronization errors in multi-threaded programs that we experimented with and believe is important and promising, but which we did not implement in Eraser.

The first topic is protection by multiple locks. Some programs protect some shared variables by multiple locks instead of a single lock. In this case the rule is that every thread that writes the variable must hold all the protecting locks, and every thread that reads the variable must hold at least one protecting lock. This policy allows a pair of simultaneous accesses only if both accesses are reads, and therefore prevents data races.

Using multiple protecting locks is in some ways similar to using reader locks and writer locks, but it is not so much aimed at increasing concurrency as at avoiding deadlock in a program that contains upcalls.

Using an earlier version of Eraser that detected race conditions in multi-threaded Modula-3 programs, we found that the Lockset algorithm reported false alarms for Trestle programs[Manasse & Nelson 91] that protected shared locations with multiple locks, because each of two readers could access the location while holding two different locks. As an experiment, we dealt with the problem by modifying the Lockset algorithm to refine the candidate set only for writes, while checking it for both reads and writes, as follows:

On each read of $v$ by thread $t$,
  if $C(v) = \{\}$, then issue a warning.

On each write of $v$ by thread $t$,
  set $C(v) := C(v) \cap locks\_held(t)$;
  if $C(v) = \{\}$, then issue a warning.

This prevented the false alarms, but it is possible for this modification to cause false negatives. For example, if a thread $t_1$ reads $v$ while holding lock $m_1$, and a thread $t_2$ writes $v$ while holding lock $m_2$, the violation of the locking discipline will be reported only if the write precedes the read. In general, the modified version will do a good job only if the test case causes enough shared variable reads to follow the corresponding writes.

Theoretically it would be possible to handle multiple protecting locks without any risk of false negatives, but the data structures required (sets of sets of locks instead of just sets of locks) seem to have a cost in complexity that is out of proportion to the likely gain. Since we are uncomfortable with false negatives and the multiple protecting lock technique is not common, the current version of Eraser ignores the technique, producing false alarms for programs that use it.

The second topic is deadlock. If the data race is Scylla, the deadlock is Charybdis.

A simple discipline that avoids deadlock is to choose a partial order among all locks and program each thread so that whenever it holds more than one lock, it acquires them in ascending order. This discipline is similar to the locking discipline for avoiding data races: it is suitable for checking by dynamic monitoring, and it is easier to produce a test case that exposes a breach of the discipline than it is to produce a test case that actually causes a deadlock.

As a stand-alone experiment, we chose a large Trestle application that was known to have complicated synchronization (`formsedit`, a double-view user interface editor), logged all lock acquisitions and tested to see if an order existed on the locks that was respected by every thread. A few seconds into `formsedit` startup our experimental monitor detected a cycle of locks, showing that no partial order existed. Examining the cycle closely revealed a potential deadlock in `formsedit`. We consider this a promising result and conjecture that deadlock-checking along these lines would be a useful addition to Eraser. But more work is required to catalog the sound and useful variations on the partial order discipline, and to develop annotations to suppress false alarms.

# 6 Conclusion

Hardware designers have learned to design for testability. Programmers using threads must learn the same. It is not enough to write a correct program; the correctness must be demonstrable, ideally by static checking, realistically by a combination of partial static checking followed by disciplined dynamic testing.

This paper has described the advantages of enforcing a simple locking discipline instead of checking for races in general parallel programs that employ many different synchronization primitives, and has also demonstrated that with this technique it is practical to dynamically check production multi-threaded programs for data races.

Programmers in the area of operating systems seem to view dynamic race detection tools as esoteric and impractical. Our experience leads us to believe instead that they are a practical and effective way to avoid data races, and that dynamic race detection should be a standard procedure in any disciplined testing effort for a multithreaded program. As the use of multi-threading expands, so will the unreliability caused by data races, unless better methods are used to eliminate them. We believe that the Lockset method implemented in Eraser is promising.

# Acknowledgments

We would like the thank the following individuals for their contributions to this project. Sung-Eun Choi and E. Christoper Lewis were responsible for all of the undergraduate experiments. Alan Heydon, Dave Detlefs, Chandu Thekkath and Edward Lee, provided expert advice on Vesta and Petal. Puneet Kumar worked on a earlier version of Eraser. Cynthia Hibbard, Brian Bershad, Michael Ernst, Wilson Hsieh and Terri Watson provided useful feedback on earlier drafts of this paper. The SOSP reviewers and our shepherd, Paulo Guedes, also provided many useful comments.

# References

[Bershad et al. 95] Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M., Becker, D., Eggers, S., and Chambers, C. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, CO, December 1995.

[Choi & Lewis 97] Choi, S.-E. and Lewis, E. C. University of Washington CSE 552 Project. Personal Communication, March 1997. `http://www.cs.washington.edu/homes/sungeun/notes/cse552/index.html`.

[Detlefs et al. 97] Detlefs, D. L., Leino, R. M., Nelson, G., and Saxe, J. B. Extended Static Checking. Technical Report 149, Digital Equipment Corporation, Systems Research Center, to appear 1997.

[Digital Equipment 96a] Digital Equipment. AltaVista Search. `http://altavista.digital.com/`, 1996.

[Digital Equipment 96b] Digital Equipment. Vesta Home Page. `http://www.research.digital.com/SRC/vesta/`, 1996.

[Dinning & Schonberg 90] Dinning, A. and Schonberg, E. An Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–10, Seattle, WA, March 1990.

[Dinning & Schonberg 91] Dinning, A. and Schonberg, E. Detected Access Anomalies in Programs with Critical Sectons. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 26(12):85–96, December 1991.

[Hoare 74] Hoare, C. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10), October 1974.

[Kleiman & Eykholt 95] Kleiman, S. and Eykholt, J. Interrupts as Threads. *ACM Operating Systems Review*, 29(2):21–26, April 1995.

[Lamport 78] Lamport, L. Time, clock, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), July 1978.

[Lampson & Redell 80] Lampson, B. and Redell, D. Experiences with Processes and Monitors in Mesa. *Communications of the ACM*, 23(2):104–117, February 1980.

[Lee & Thekkath 96] Lee, E. K. and Thekkath, C. A. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, 1996.

[Manasse & Nelson 91] Manasse, M. S. and Nelson, G. Trestle Reference Manual. Technical Report Research Report 68, Digital Equipment Corporation's Systems Research Center, Palo Alto, California, December 1991.

[Mellor-Crummey 91] Mellor-Crummey, J. On-the-fly Detection of Data Races for Programs with Nested Fork-Join Parallelism. In *Proceedings of the 1991 Supercomputer Debugging Workshop*, pages 1–16, November 1991.

[Mellor-Crummey 93] Mellor-Crummey, J. Compile-time Support for Efficient Data Race Detection in Shared-Memory Parallel Programs. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 129–139, May 1993.

[Nelson et al. 96] Nelson, G., Leino, K. R. M., Saxe, J., and Stata, R. Extended Static Checking Home Page. `http://www.research.digital.com/SRC/esc/Esc.html`, 1996.

[Netzer 91] Netzer, R. H. B. *Race Condition Detection for Debugging Shared-Memory Parallel Programs*. PhD dissertation, University of Wisconsin-Madison, 1991.

[Ousterhout 96] Ousterhout, J. K. Why Threads Are A Bad Idea (for most purposes). Invited talk at the 1996 USENIX Conference, January 1996. `http://www.smli.com/people/john.ousterhout/threads.ps`.

[Perkovic & Keleher 96] Perkovic, D. and Keleher, P. Online Data-Race Detection via Coherency Guarantees. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 47–58, Seattle, WA, October 1996.

[Scales et al. 96] Scales, D. J., Gharachorloo, K., and Thekkath, C. A. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, 1996.

[Srivastava & Eustace 94] Srivastava, A. and Eustace, A. ATOM: A System for Building Customized Program Analysis Tools. In *Proceedings of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 196–205, 1994.

[SunSoft 94] SunSoft. `lock_lint` User's Guide. SunSoft Manual, August 1994.