Effective Data-Race Detection for the Kernel

John Erickson, Madanlal Musuvathi,
Sebastian Burckhardt, Kirk Olynyk
Microsoft Research
{jerick, madanm, sburckha, kirko}@microsoft.com

Abstract

Data races are an important class of concurrency errors where two threads erroneously access a shared memory location without appropriate synchronization. This paper presents *DataCollider,* a lightweight and effective technique for dynamically detecting data races in kernel modules. Unlike existing data-race detection techniques, DataCollider is oblivious to the synchronization protocols (such as locking disciplines) the program uses to protect shared memory accesses. This is particularly important for low-level kernel code that uses a myriad of complex architecture/device specific synchronization mechanisms. To reduce the runtime overhead, DataCollider randomly samples a small percentage of memory accesses as candidates for data-race detection. The key novelty of DataCollider is that it uses breakpoint facilities already supported by many hardware architectures to achieve negligible runtime overheads. We have implemented DataCollider for the Windows 7 kernel and have found 25 confirmed erroneous data races of which 12 have already been fixed.

1. Introduction

Concurrent systems are hard to design, arguably because of the difficulties of finding and fixing concurrency errors. Data races are an important class of concurrency errors, where the program fails to use proper synchronization when accessing shared data. The effects of an erroneous data race can range from immediate program crashes to silent lost updates and data corruptions that are hard to reproduce and debug.

Two memory accesses in a program are said to *conflict* if they access the same memory location and at least one of them is a write. A program contains a data race if two conflicting accesses can occur concurrently. Figure 1 shows a variation of a data race we found in the Windows kernel. The threads appear to be accessing different fields. However, these bit-fields are mapped to the same word by the compiler and the concurrent accesses result in a data race. In this case, an update to the statistics field possibly hides an update to the status field.

This paper presents *DataCollider,* a tool for dynamically detecting data races in kernel modules. DataCollider is lightweight. It samples a small number of memory accesses for data-race detection and uses code-breakpoint and data-breakpoint[1] facilities available in modern hardware architectures to efficiently perform this sampling. As a result, DataCollider has *no* runtime overhead for non-sampled memory accesses allowing the tool to run with negligible overheads for low sampling rates.

We have implemented DataCollider for the 32-bit Windows kernel running on the x86 architecture, and used it to detect data races in the core kernel and several modules such as the filesystem, the networking stack, the storage drivers, and a network file system. We have found a total of 25 erroneous data races of which 12 have already been fixed at the time of writing. In our experiments, the tool is able to find erroneous data races for sampling rates that incur runtime overheads of less than 5%.

Researchers have proposed multitude of dynamic data-race detectors [1,2,3,4,5,6,7] for user-mode programs. In essence, these tools work by dynamically monitoring the memory accesses and synchronizations performed during a concurrent execution. As data races manifest rarely at runtime, these tools attempt to infer conflicting accesses that *could* have executed concurrently. The tools differ in how they perform this inference, either

[1] Data breakpoints are also called hardware watchpoints.

```
struct{
   int status:4;
   int pktRcvd:28;
} st;
```

Thread 1	Thread 2
`st.status = 1;`	`st.pktRcvd ++;`

Figure 1: An example of data race. Even though the threads appear to be modifying different variables in the source code, the variables are bit fields mapping to the same integer

using the *happens-before* [8] ordering induced by the synchronization operations [4,5,6] or a *lock-set* based reasoning [1] or a combination of the two [2,3,7]

There are several challenges in engineering a data-race detection tool for the kernel based on previous approaches. First, the kernel-mode code operates at a lower concurrency abstraction than user-mode code, which can rely on clean abstractions of threads and synchronizations provided by the kernel. In the kernel, the same thread context can execute code from a user-mode process, a device interrupt service routine, or a deferred procedure call (DPC). In addition, it is an onerous task to understand the semantics of complex synchronization primitives in order to infer the happens-before relation or lock-sets. For instance, Windows supports more than a dozen locks with different semantics on how the lock holder synchronizes with hardware interrupts, the scheduler, and the DPCs. It is also common for kernel modules to roll-out custom implementations of synchronization primitives.

Second, hardware-facing kernel modules need to synchronize with hardware devices that concurrently modify device state and memory. It is important to design a data-race detection tool that can find these otherwise hard-to-find data races between the hardware and the kernel.

Finally, existing dynamic data-race detectors add prohibitive run-time overheads. It is not uncommon for such tools to incur up to 200x slowdowns [9]. The overhead is primarily due to the need to monitor and process all memory and synchronization operations at run time. Significant engineering effort in building data-race detectors goes in reducing the runtime overhead and the associated memory and log management [9,3]. Replicating these efforts within the constraints of kernel programming is an arduous, if not impossible, task.

```
AtPeriodicIntervals() {
  // determine k based on desired
  // memory access sampling rate
  repeat k times {
    pc = RandomlyChosenMemoryAccess();
    SetCodeBreakpoint( pc );
  }
}

OnCodeBreakpoint( pc ) {
  // disassemble the instruction at pc
  (loc, size, isWrite) = disasm( pc );

  DetectConflicts(loc, size, isWrite);

  // set another code break point
  pc = RandomlyChosenMemoryAccess();
  SetCodeBreakpoint( pc );
}

DetectConflicts( loc, size, isWrite) {
  temp = read( loc, size );

  if ( isWrite )
    SetDataBreakpointRW( loc, size );
  else
    SetDataBreakpointW( loc, size );

  delay();

  ClearDataBreakpoint( loc, size );

  temp' = read( loc, size );

  if( temp != temp' ||
      data breakpoint fired )
    ReportDataRace( );
}
```

Figure 2: The basics of the DataCollider algorithm. Right before a read or write access to shared memory location, chosen at random, DataCollider monitors for any concurrent accesses that conflict with the current access.

Moreover, these tools rely on invasive instrumentation techniques that are difficult to get right on low-level kernel code.

DataCollider uses a different approach to overcome these challenges. The crux of the algorithm is shown in Figure 2. DataCollider samples a small number of memory accesses at runtime by inserting code breakpoints at randomly chosen memory access instructions. When a code breakpoint fires, DataCollider detects data races involving the sampled memory access for a small time window. It simultaneously employs two strategies

to do so. First, DataCollider sets a data breakpoint to trap conflicting accesses by other threads. To detect conflicting writes performed by hardware devices and by processors accessing the memory location through a different virtual address, DataCollider use a *repeated-read* strategy. It reads the value once before and once after the delay. A change in value is an indication of a conflicting write, and hence a data race.

The DataCollider algorithm has two features that make it suitable for kernel data-race detection. First and foremost, it is easy to implement. Barring some implementation details (Section 3), the entire algorithm is shown in Figure 2. In addition, it is entirely oblivious to the synchronization protocols used by the kernel and the hardware, a welcome design point as DataCollider does not have to understand the complex semantics of kernel synchronization primitives.

When the DataCollider finds a data race through the data-breakpoint strategy, it catches both threads "red-handed," as they are about to execute conflicting accesses. This greatly simplifies the debugging of data race reports from DataCollider as the tool can collect useful debugging information, such as the stack trace of the racing threads along with their context information, without incurring this overhead on non-sampled or non-racy accesses.

Not all data races are erroneous. Such *benign* races include races that do not affect the program outcome, such as updates to logging/debugging variables, and races that affect the program outcome in a manner acceptable to the programmer, such as conflicting updates to a low-fidelity counter. DataCollider uses a post-processing phase that prunes and prioritizes the data-race reports before showing them to the user. In our experience with DataCollider, we have observed that only around 10% percentage of data-race reports correspond to real errors, making the post-processing step absolutely crucial for the usability of the tool.

2. Background and Motivation

Shared memory multiprocessors are specifically built to allow concurrent access to shared data. So why do data races represent a problem at all?

The key motivation for data race detection is the empiric fact that programmers most often use synchronization to restrict accesses to shared memory. Data races can thus be an indication of incorrect or insufficient synchronization in the program. In addition, data races can also reveal programming mistakes not directly related to concurrency, such as buffer overruns or use-after-free, which indirectly result in inadvertent sharing of memory.

Another important reason for avoiding data races is to protect the program from the weak memory models of the compiler and the hardware. Both the compiler and hardware can reorder instructions and change the behavior of racy programs in complex and confusing ways [10,11]. Even if a racy program works correctly for the current compiler and hardware configuration, it might fail on future configurations that implement more aggressive memory-model relaxations.

While bugs caused by data races may of course be found using more conventional testing approaches such as stress testing, the latter often fails to provide actionable information to the programmer. Clearly, a data race report including stack traces or data values (or even better, including a core dump that is demonstrating the actual data race) is easier to understand and fix than a silent data corruption that leads to an obscure failure at some later point during program execution.

2.1. Definition of Data Race

There is no "gold standard" for defining data races; several researchers have used the term to mean different things. For our definition, we consulted two respected standards (Posix threads [12] and the drafts of the C++ and C memory model standards [11,10]) and generalized their definitions to account for the particularities of kernel code. Our definition of data race is:

- Two operations that access main memory are called *conflicting* if
  - the physical memory they access is not disjoint,
  - at least one of them is a write, and
  - they are not both synchronization accesses.
- A program *has a data race* if it can be executed on a multiprocessor in such a way that two conflicting memory accesses are performed simultaneously (by processors or any other device).

This definition is a simplification of [11,10] insofar we replaced the tricky notion of "not ordered before" with the unambiguous "performed simultaneously" (which refers to real time).

An important part of our definition is the distinction between synchronization and data accesses. Clearly, some memory accesses participate in perfectly desirable races: for example, a mutex implementation may perform a "release" by storing the value 0 in a shared loca-

tion, while another thread is performing an acquire and reads the same memory location. However, this is not a *data* race because we categorize both of these accesses as synchronization accesses. Synchronization accesses either involve hardware synchronization primitives such as interlocked instructions or use volatile or atomic annotations supported by the compiler.

Note that our definition is general enough to apply to code running in the kernel, which poses some unique problems not found in user-mode code. For example, in some cases data races can be avoided by turning off interrupts; also, processes can exhibit a data race when accessing different virtual addresses that map to the same physical address. We talk more about these topics in Section 2.3.4.

2.2. Precision of Detection

Clearly, we would like data race detection tools to report as many data races as possible without inundating the user with false error reports. We use the following terminology to discuss the precision and completeness of data race detectors. A *missed race* is a data race that the tool does not warn about. A *benign* data race is a data race that does not adversely affect the behavior of the program. Common examples of benign data races include threads racing on updates to logging or statistics variables and threads concurrently updating a shared counter where the occasional incorrect update of the counter does not affect the outcome of the program. On the other hand, a *false* data race is an error report that does not correspond to a data race in the program. Static data-race detection techniques commonly produce false data races due to their inherent inability to precisely reason about program paths, aliased heap objects, and function pointers. Dynamic data-race detectors can report false data races if they do not identify or do not understand the semantics of *all* the synchronizations used by the program.

2.3. Related Work

Researchers have proposed and built a plethora of race detection tools. We now discuss the major approaches and implementation techniques appearing in related work. We describe both happens-before-based and lock-set-based tracking in some detail (Sections 2.3.2 and 2.3.3), before explaining why neither one is very practical for data race detection in the kernel (Section 2.3.4).

2.3.1. Static vs. Dynamic

Data race detection can be broadly categorized into *static* race detection [13,14,15,16,17], which typically analyzes source or byte code without directly executing the program, and *dynamic* race detection [1,2,3,4,5,6,7], which instruments the program and monitors its execution online or offline.

Static race detectors have been successfully applied to large code bases [13,14]. However, as they rely on approximate information, such as pointer aliasing, they are prone to excessive false warnings. Some tools, especially those targeting large code bases, approach this issue by filtering the reported warnings using heuristics [13]. Such heuristics can successfully reduce the false warnings to a tolerable level, but may unfortunately also eliminate correct warnings and lead to missed races. Other tools, targeted towards highly motivated users that wish to interactively prove absence of data races, report all potential races to the user and rely on user-supplied annotations that indicate synchronization disciplines [16,17].

Dynamic data race detectors are less prone to false warnings than static techniques because they monitor an actual execution of the program. However, they may miss races because successful detection might require an error-inducing input and/or an appropriate thread schedule. Also, many dynamic detectors employ several heuristics and approximations that can lead to false alarms.

Dynamic data race detectors can be classified into categories based on whether they model a happens-before relation [6,5,7] (see Section 2.3.2), lock sets [1] (see Section 2.3.3), or both [2,18].

2.3.2. Happens-Before Tracking

Dynamic data race detectors do not just detect data races that actually took place (in the sense that the conflicting accesses were truly simultaneous during the execution), but look for evidence that such a schedule would have been possible for a slightly different timing. Tracking a *happens-before* relation on program events [8] is one way to infer the existence of a racy schedule. This transitive relation is constructed by recording both the ordering of events within a thread and the ordering effects of synchronization operations across threads.

Once we can properly track the happens-before relation, race detection is straightforward: For any two conflicting accesses A and B, we simply check whether A happens-before B, or B happens-before A, or neither. If

neither, we know there exists a schedule where A and B are simultaneous. If properly tracked, happens-before does not lead to any false alarms. However, precise tracking can be difficult to achieve in practice, as discussed in Section 2.3.4.

2.3.3. Lock Sets

When detecting races in programs that follow a strict and consistent locking discipline, using a lock-set approach can provide some benefits. The basic idea is to examine the lock set of each data access (that is, the set of locks held during the access) and then to take for each memory location the intersection of the lock sets of all accesses to it. If that intersection is empty, the variable is not consistently protected by any one lock and a warning is issued.

The main limitation of the lock set approach is that it does not check for true data races but for violations of a specific locking discipline. Unfortunately, many applications (and in particular kernel code) use locking disciplines that are complex and use synchronization other than locks.

Whenever a program departs from a simple locking scheme in any of the above ways, lock-set-based race detectors will be forced to either issue false warnings, or to use heuristics to suppress these warnings. The latter approach is common, especially in the form of state machines that track the "sharing status" of a variable [1,3]. Such heuristics are necessarily imperfect compromises, however (they always fail to suppress some false warnings and always suppress some correct warnings), and it is not clear how to tune them to be useful for a wide range of applications.

2.3.4. Problems with Tracking Synchronizations

Both lock-set and happens-before tracking require a thorough understanding of the synchronization semantics, lest they produce false alarms or miss races. There are two fundamental difficulties we encountered when trying to apply these techniques in the kernel:

- Abstractions that we take for granted in user mode (such as threads) are no longer clearly defined in kernel mode.
- The synchronization vocabulary of kernel code is much richer and may include complicated sequences and ordering mechanisms provided by the hardware.

For example, interrupts and interrupt handlers break the thread abstraction, as the handler code may execute in a thread context without being part of that thread in a logical sense. Similar problems arise when a thread calls into the kernel scheduler. The code executing in the scheduler is not logically part of that same thread.

Another example illustrating the difficulty of modeling synchronization inside the kernel are DMA accesses. Such accesses are not executing inside a thread (in fact, they are not even executing on a processor). Clearly, traditional monitoring techniques have a problem because they cannot "instrument" the DMA access.

Similar case holds for interrupt processing. For example, code may first write some data and then raise an interrupt, and then the same data is read by an interrupt handler. Lock sets would report a false alarm because the data is not locked. But even happens-before techniques are problematic, because they would need to precisely track the causality between the instruction that set the interrupt and the interrupt handler.

For these reasons, we decided to employ a design that entirely avoids modeling the happens-before ordering or lock-sets. As our results show, somewhat surprisingly, neither one is required to build an effective data race detector.

2.3.5. Sampling to Reduce Overhead

To detect races, dynamic data race detectors need to monitor the synchronizations and memory accesses performed at runtime. This is typically done by instrumenting the code and inserting extra monitoring code for each data access. As the monitoring code executes at every memory access, the overhead can be quite substantial.

One way to ameliorate this issue is to exclude some data accesses from processing. Prior work has identified several promising strategies: adaptive sampling that backs off hot locations [5] (the idea is that for such locations the monitoring can be less frequent and still detect races), or perform the full monitoring only for a fixed fraction of the time [4] (the idea is that the probability of catching a race is roughly proportional to this fraction multiplied by the number of times the race repeats). But these techniques still suffer from the cost of sampling, performed at every memory access. DataCollider avoids this problem by using hardware breakpoint mechanisms.

3. DataCollider Implementation

This section describes the implementation of the DataCollider algorithm for the Windows kernel on the x86 architecture. The implementation heavily uses the code and data breakpoint mechanisms available on x86. The techniques described in this paper can be extended to other architectures and to user-mode code. But we have not pursued this direction in this paper.

Figure 2 describes the basics of the DataCollider algorithm. DataCollider uses the sampling algorithm, described in Section 3.1, to process a small percentage of memory accesses for data-race detection. For each of the sampled memory accesses, DataCollider uses a conflict detection mechanism, described in Section 3.2, to find data races involving the sampled access. After detecting data races, DataCollider uses several heuristics, described in Section 3.3, to prune benign data races.

3.1. The Sampling Algorithm

There are several challenges in designing a good sampling algorithm for data-race detection. First, data races involve two memory accesses both of which need to be sampled to detect the race. If memory accesses are sampled independently, then the probability of finding the data race is a product of the individual sampling probabilities. DataCollider avoids this multiplicative effect by sampling the first access and using a data breakpoint to trap the second access. This allows DataCollider to be effective at low sampling rates.

Second, data races are rare events – most executed instructions do not result in a data race. The sampling algorithm should weed out the small percentage of racing accesses from the majority of non-racing accesses. The key intuition behind the sampling algorithm is that if a program location is buggy and fails to use the right synchronization when accessing shared data, then every dynamic execution of that buggy code is likely to participate in a data race. Accordingly, DataCollider performs *static* sampling of program locations rather than *dynamic* sampling of executed instructions. A static sampler provides equal preference to rarely execution instructions (which are likely to have bugs hidden in them) and frequently executed instructions.

3.1.1. Static Sampling Using Code Breakpoints

The static sampling algorithm works as follows. Given a program binary, DataCollider disassembles the binary to generate a *sampling set* consisting of all program locations that access memory. The tool currently requires the debugging symbols of the program binary to perform this disassembly. This requirement can be relaxed by using sophisticated disassemblers [19] in the future.

DataCollider performs a simple static analysis to identify instructions that are guaranteed to only touch thread-local stack locations and removes them from the sampling set. Similarly, DataCollider removes synchronizing instructions from the sampling set by removing instructions that accesses memory locations tagged as "volatile" or those that use hardware synchronization primitives, such as interlocked. This prevents DataCollider from reporting races on synchronization variables. However, DataCollider can still detect a data race between a synchronization access and a regular data access, if the latter is in the sampling set.

DataCollider samples program locations from the sampling set by inserting code breakpoints. The initial breakpoints are set at a small number of program locations chosen uniformly randomly from the sampling set. If and when a code breakpoint fires, DataCollider performs conflict detection for the memory access at that breakpoint. Then, DataCollider choses another program location uniformly randomly from the sampling set and sets a breakpoint at that location.

This algorithm uniformly samples all program locations in the sampling set irrespective of the frequency with which the program executes these locations. This is because the choice of inserting a code breakpoint is performed uniformly at random for all locations in the sampling set. Over a period of time, the breakpoints will tend to reside at rarely executed program locations, increasing the likelihood that those locations are sampled the next time they execute.

If DataCollider has information on which program locations are likely to participate in a race, either through user annotations or through prior analysis [20] then the tool can prioritize those locations by biasing their selection from the sampling set.

3.1.2. Controlling the Sampling Rate

While the program cannot affect the sampling distribution over program locations, the sampling *rate* is intimately tied to how frequently the program executes locations with a code breakpoint. In the worst case, if all of the breakpoints are set on dead code, DataCollider will stop performing data-race detection altogether. To avoid this and to better control the sampling rate, DataCollider periodically checks the number of breakpoints fired every second, and adjusts the number of

breakpoints set in the program based on whether the experienced sampling rate is higher or lower than the target rate.

3.2. Conflict-Detection

As described in the previous section, DataCollider picks a small percentage of memory accesses as likely candidates for data-race detection. For these sampled accesses, DataCollider pauses the current thread waiting to see if another thread makes a conflicting access to the same memory location. It uses two strategies: data breakpoints and repeated-reads. DataCollider uses these two strategies simultaneously as each complements the weaknesses of the other.

3.2.1. Detecting Conflicts with Data Breakpoints

Modern hardware architectures provide a facility to trap when a processor reads or writes a particular memory location. This is crucial for efficient support for data breakpoints in debuggers. The x86 hardware supports four data breakpoint registers. DataCollider uses them to effectively monitor possible conflicting accesses to the currently sampled access.

When the current access is a write, DataCollider instructs the processor to trap on a read or write to the memory location. If the current access is a read, DataCollider instructs the processor to trap only on a write, as concurrent reads to the same location do not conflict. If no conflicting accesses are detected, DataCollider resumes the execution of the current thread after clearing the data breakpoint registers.

Each processor has a separate data breakpoint register. DataCollider uses an inter-processor interrupt to update the break points on all processors atomically. This also synchronizes multiple threads attempting to sample different memory locations concurrently.

An x86 instruction can access variable sized memory. For 8, 16, or 32-bit accesses, DataCollider sets a breakpoint of the appropriate size. The x86 processor traps if another instruction accesses a memory location that overlaps with a given breakpoint. Luckily, this is precisely the semantics required for data-race detection. For accesses that span more than 32 bits, DataCollider uses more than one breakpoint up to the maximum available of four. If DataCollider runs out of breakpoint registers, it simply resorts to the repeated-read strategy discussed below.

When a data breakpoint fires, DataCollider has successfully detected a race. More importantly, it has caught the racing threads “red handed” – the two threads are at the point of executing conflicting accesses to the same memory location.

One particular shortcoming of data breakpoint support in x86 that we had to work around was the fact that, when paging is enabled, x86 performs the breakpoint comparisons based on the virtual address and has no mechanism to modify this behavior. Two concurrent accesses to the same virtual addresses but different physical addresses do not race. In Windows, most of the kernel resides in the same address space with two exceptions.

Kernel threads accessing the user address space cannot conflict if the threads are executing in the context of different processes. If a sampled access lies in the user address space, DataCollider does not use breakpoints and defaults to the repeated-read strategy.

Similarly, a range of kernel-address space, called *session memory*, is mapped to different address spaces based on the session the process belongs to. When a sampled access lies in the session memory space, DataCollider sets a data breakpoint but checks if the conflicting accesses belong to the same session before reporting the conflict to the user.

Finally, a data breakpoint will miss conflicts if a processor uses a different virtual address mapped to the same physical address as the sampled access. Similarly, data breakpoints cannot detect conflicts arising from hardware devices directly accessing memory. The repeated-read strategy discussed below covers all these cases.

3.2.2. Detecting Conflicts with Repeated Reads

The repeated-read strategy relies on a simple insight: if a conflicting write changes the value of a memory location, DataCollider can detect this by repeatedly reading the memory location checking for value changes. An obvious disadvantage of this approach is that it cannot detect conflicting reads. Similarly, it cannot detect multiple conflicting writes the last of which writes the same value as the initial value. Despite these shortcomings, we have found this strategy to be very useful in practice. This is the first strategy we implemented (as it is easier to implement than using data breakpoints) and we were able to find several kernel bugs with this approach.

However, repeated-reads strategy catches only one of the two threads "red-handed." This makes it harder to debug data races, as one does not know which thread or device was responsible for the conflicting write. This was our prime motivation for using data breakpoints.

3.2.3. Inserting Delays

For a sampled memory access, DataCollider attempts to detect a conflicting access to the same memory location by delaying the thread for a short amount of time. For DataCollider to be successful, this delay has to be long enough for the conflicting access to occur. On the other hand, delaying the thread for too long can be dangerous especially if the thread holds some resource crucial for the proper functioning of the entire system. In general, it is impossible to predict how long to insert the delay. After experimenting with many values, we chose the following delay algorithm.

Depending on the IRQL (Interrupt Request Level) of the executing thread, DataCollider delays the thread for a preset maximum amount of time. At IRQLs higher than the DISPATCH level (the level at which the kernel scheduler operates), DataCollider does not insert any delay. We considered inserting a small window of delay at this level to identify possible data races between interrupt service routines. But we did not expect that DataCollider would be effective at short delays.

Threads running at the DISPATCH level cannot yield the processor to another thread. As such, the delay is simply a busy loop. We currently delay threads at this level for a random amount of time less than 1 ms. For lower IRQLs, DataCollider delays the thread for a maximum of 15 ms by spinning in a loop that yields the current time quantum. During this loop, the thread repeatedly checks to see if other threads are making progress by inspecting the rate at which breakpoints fire. If progress is not detected, the waiting thread prematurely stops its wait.

3.3. Dealing with Benign Data Races

Research on data-race detection has amply noted the fact that not all data races are erroneous. A practical data-race detection tool should effectively prune or deprioritize these benign data races when reporting to the user. However, inferring whether or not a data race is benign can be tricky and might require deep understanding of the program. For instance, a data race between two concurrent non-atomic counter updates might be benign if the counter is a statistic variable whose fidelity is not important to the behavior of the program. However, if the counter is used to maintain the number of references to a shared object, then the data race could lead to a memory leak or a premature free of the object.

During the initial runs of the tool, we found that around 90% of the data-race reports are benign. Inspecting these we identified the following patterns that can be identified through simple static and/or dynamic analysis and incorporated them in a post-process pruning phase.

Statistics Counters: Around half of the benign data races involved conflicting updates to counters that maintain various statistics about the program behavior [21]. These counters are not necessarily write-only and could affect the control flow of the program. A common scenario is to use these counter value to perform periodic computation such as flushing a log buffer. If DataCollider reports several data races involving an increment instruction and the value of the memory location consistently increases across these reports, then the pruning phase tags these data races as statistics-counter races. Checking for an increase in memory values helps the pruning phase in distinguishing these statistics counters from reference counters that are usually both incremented and decremented.

Safe Flag Updates: The next prominent class of benign races involves a thread reading a flag bit in a memory location while another thread updates a different bit in the same memory location. By analyzing few memory instructions before and after the memory access, the pruning phase identifies read-write conflicts that involve different bits. On the other hand, write-write conflicts can result in lost updates (as shown in Figure 1) and are not tagged as benign.

Special Variables: Some of the data races reported by DataCollider involve special variables in the kernel where races are expected. For instance, Windows maintains the current time in a variable, which is read by many threads while being updated by the timer interrupt. The pruning phase has a database of such variables and prunes races involving these variables.

While it is possible to design other patterns that identify benign data races, one has to tradeoff the benefit of the pruning achieved with the risk of missing real data races. For instance, we initially designed a pattern to classify two writes that write the same value as benign. However, very few data-race reports matched this property. On the other hand, Figure 4 shows an example of a harmful data-race that we found involving two such writes.

Also, we have made an explicit decision to make the benign data races available to the user, but deprioritized

Data Races Reported	Count
Fixed	12
Confirmed and Being Fixed	13
Under Investigation	8
Harmless	5
Total	38

Figure 3: Bugs reported to the developers after excluding benign data-race reports.

against races that are less likely to be benign. Some of our users are interested in browsing through the pruned benign races to identify potential portability problems and memory-model issues in their code. We also found an instance where a benign race, despite being harmless, indicated unintended sharing in the code and resulted in a design change.

4. Evaluation

There are two metrics for measuring the success of a data-race detection tool. First, is it able to find data races that programmers deem important enough to fix? Second, is it able to scale to a large system, which in our case is the Windows operating system, with reasonable runtime overheads? This section presents a case for an affirmative claim on these two metrics.

4.1. Experimental Setup

For the discussion in this section, we applied DataCollider on several modules in the Windows operating system. DataCollider has been has been used on class drivers, various PnP drivers, local and remote file system drivers, storage drivers, and the core kernel executive itself. We are successfully able to boot the operating system with DataCollider and run existing kernel stress tests.

4.2. Bugs Found

Figure 3 presents the data race reports produced by the different versions of DataCollider during its entire development. We reported a total 38 data-race reports to the developers. This figure does not reflect the number of benign data races pruned heuristically and manually. We defer the discussion of benign data races to Section 4.4.

Of these 38 reports, 25 have been confirmed as bugs and 12 of which have already been fixed. The developers indicated that 5 of these are indeed harmless. For instance, one of the benign data races results in a driver issuing an idempotent request to the device. While this could result in a performance loss, the expected frequency of the data race did not justify the cost of adding synchronization in the common case. Identifying such benign races requires intimate knowledge of the code and would not be possible without the programmers help.

As DataCollider naturally delays the racing access that temporally occurs first, it is likely to explore both outcomes of the race. Despite this, only one of the 38 data races crashed the kernel in our experiments. This indicates that the effects of an erroneous data race are not immediately apparent for the particular input or the hardware configuration of the current run.

We discuss two interesting error reports below

4.2.1. A Boot Hang Caused by a Data Race

A hardware vendor was consistently seeing a kernel hang at boot-up time. This was not reproducible in any of the in-house machine configurations, till the vendor actually shipped the hardware to the developers. After inspecting the hang, a developer noticed a memory corruption in a driver that could be a result of a race condition. When analyzing the driver in question, DataCollider found the data race in an hour of testing on a regular in-house machine (in which the kernel did not hang). Once the source of the corruption was found (performing a status update non-atomically), the bug was immediately fixed.

```
void AddToCache() {
   // ...
   A: x &= ~(FLAG_NOT_DELETED);
   B: x |= FLAG_CACHED;

   MemoryBarrier();
   // ...
}

AddToCache();
assert( x & FLAG_CACHED );
```

Figure 4: An erroneous data race when the `AddToCache` function is called concurrently. Though the data race appears benign, as the conflicting accesses "write the same values," the `assert` can fail on some thread schedules.

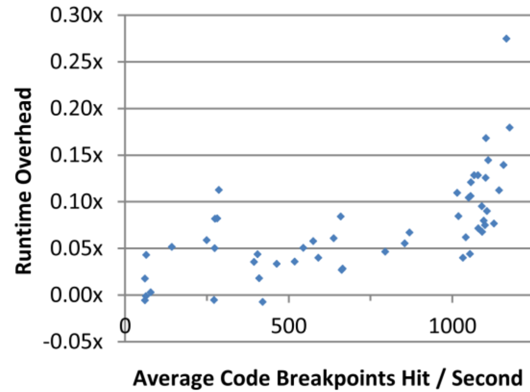

Figure 5: Runtime overhead of DataCollider with increasing sampling rate, measured in terms of the number of code breakpoints firing per second. The overhead tends to zero as the sampling rate is reduced, indicating that the tool has negligible base overhead.

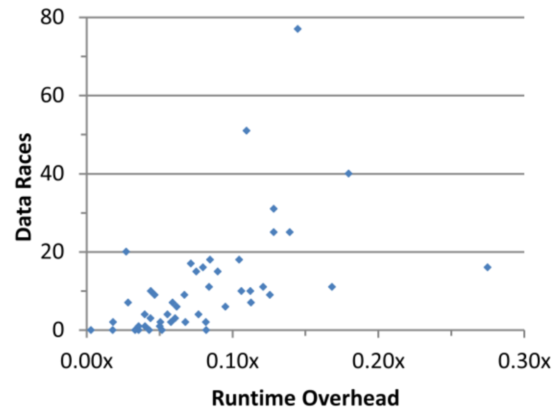

Figure 6: The number of data races, uniquely identified by the pair of racing program locations, with the runtime overhead. DataCollider is able to report data race even under overheads under 5%

4.2.2. A Not-So-Benign Data Race

Figure 4 shows an erroneous data race. The function `AddToCache` performs two non-atomic updates to the flag variable. DataCollider produced an error report with two threads simultaneously updating the flag at location B. Usually, two instructions writing the same values is a good hint that the data race is benign. However, the presence of the memory barrier indicated that this report required further attention – the developer was well aware of consequences of concurrency and the rest of the code relied on crucial invariants on the flag updates. When we reported this data race to the developer he initially tagged it as benign. On further discussion, we discovered that the code relied on the invariant that the `CACHED` bit is set after a call to `AddToCache`. The data race can break this invariant when a concurrent thread overwrites `CACHED` bit when performing the update at A, but gets preempted before setting the bit at B.

4.2.3. How Fixed

While data races can be hard to find and result in mysterious crashes, our experience is that most are relatively easy to fix. Of the 12 bugs, 3 were the result of missing locks. The developer could easily identify the locking discipline that was meant to be followed, and could decide which lock to add without the fear of a deadlock. 6 data races were the fixed by using an atomic instructions, such as interlocked increment, to make a read-modify-write to a shared variable. 2 bugs were a result of unintended sharing and were fixed by making the particular variable thread local. Finally, one bug indicated a broken design due to a recent refactoring and resulted in a design change.

4.3. Runtime Overhead

Users have an inherent aversion to dynamic analysis tools that add prohibitive runtime overheads. The obvious reason is the associated wastage of test resources – a slowdown of ten means that only one-tenth the amount of testing can be done with a given amount of resources. More importantly, runtime overheads introduced by a tool can affect the real-time execution of the

Data Race Category		Count
Benign – Heuristically Pruned	Statistic Counter	52
	Safe Flag Update	29
	Special Variable	5
	Subtotal	86
Benign – Manually Pruned	Double-check locking	8
	Volatile	8
	Write Same Value	1
	Other	1
	Subtotal	18
Real	Confirmed	5
	Investigating	4
	Subtotal	9
Total		113

Figure 7: Categorization of data races found by DataCollider during kernel stress.

program. The operating system could start a recovery action if a device interrupt takes too long to finish. Or a test harness can incorrectly tag a kernel-build faulty if it takes too long to boot.

To measure the runtime overhead of DataCollider, we repeatedly measured the time taken for the boot-shutdown sequence for different sampling rates and compared against a baseline Windows kernel running without DataCollider. These experiments where done on the x86 version of Windows 7 running on a virtual machine with 2 processors and 512 MB memory. The host machine is an Intel Core2-Quad 2.4 GHz machine with 4 GB memory running Windows Server 2008. The guest machine was limited to 50% of the processing resources of the host. This was done to prevent any background activity on the host from perturbing the performance of the guest.

Figure 5 shows the runtime overhead of DataCollider for different sampling rates, measured by the average number of code breakpoints fired per second during the run. As expected, the overhead increases roughly linearly with the sampling rate. More interestingly, as the sampling rate tends to zero, DataCollider's overhead reaches zero. This indicates that DataCollider can be "always on" in various testing and deployment scenarios, allowing the user to tune the overhead to any acceptable limit.

Figure 6 shows the number of data races detected for different runtime costs. DataCollider is able to detect data races even for overheads less than 5% indicating the utility of the tool at low overheads.

4.4. Benign Data Races

Finally, we performed an experiment to measure the efficacy of our pruning algorithm for benign data races. The results are shown in Figure 7. We enabled DataCollider while running kernel stress tests for 2 hours sampling at approximately 1000 code breakpoints per second. DataCollider found a total of 113 unique data races. The patterns described in Section 3.3 can identify 86 (76%) of these as benign errors. We manually (and painfully) triaged these reports to ensure that these races were truly benign. Of the remaining races, we manually identified 18 as not erroneous. 8 of them involved the double-checked locking idiom, where a thread performs a racy read of a flag without holding a lock, but reconfirms the value after acquiring the lock. 8 were accesses to volatile variables that DataCollider's analysis was unable to infer the type of. These reports can be avoided with a more sophisticated analysis for determining the program types. This table demonstrates that a significant percentage of benign data races can be heuristically pruned without risks of missing real data races. During this process, we found 9 potentially harmful data races of which 5 have already been confirmed as bugs.

5. Conclusion

This paper describes DataCollider, a lightweight and effective data-race detector specifically designed for low-level systems code. Using our implementation of DataCollider for the Windows operating system, we have found to date 25 erroneous data races of which 12 are already fixed.

We would like to thank our shepherd Junfeng Yang and all our anonymous reviewers for valuable feedback on the paper.

References

[1] Stefan Savage, Michael Burrows, Greg Nelson, and Patrick Sobalvarro, "Eraser: A Dynamic Data Race Detector for Multithreaded Programs," *ACM Transactions on Computer Systems*, vol. 15, no. 4, pp. 391-411, 1997.

[2] Robert O'Callahan and Jong-Deok Choi, "Hybrid Dynamic Data Race Detection," *SIGPLAN Not.*, vol. 38, no. 10, pp. 167-178, 2003.

[3] Yuan Yu, Tom Rodeheffer, and Wei Chen, "RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking," in *Symposium on Operating System Principles (SOSP)*, 2005, pp. 221-234.

[4] Michael D Bond, Katherine E Coons, and Kathryn S McKinley, "PACER: Proportional Detection of Data Races," in *Programming Languages Design and Implementation (PLDI)*, 2010.

[5] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasami, "LiteRace: Effective Sampling for Lightweight Data-Race Detection," in *Programming Language Design and Implementation*, 2009, pp. 134-143.

[6] Cormac Flanagan and Stephen N Freund, "FastTrack: Efficient and Precise Dynamic Race Detection," in *Programming Language Design and Implementation*, 2009, pp. 121-133.

[7] E Pozniansky and A Schuster, "MultiRace: Efficient on-the-fly data race detection in multithreaded C++ programs," *Concurrency and Computation: ractice and Experience*, vol. 19, no. 3, pp. 327-340, 2007.

[8] Leslie Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558-565, 1978.

[9] Paul Sack, Brian E Bliss, Zhiqiang Ma, Paul Petersen, and Josep Torrellas, "Accurate and Efficient Filtering for the Intel Thread Checker Race Detector," in *Workshop on Architectural and System Support for Improving Software Dependability*, 2006, pp. 34-41.

[10] Hans Boehm and Sarita Adve, "Foundations of the C++ Concurrency Memory Model," HP Labs, Technical Report HPL-2008-56 , 2008.

[11] Hans Boehm. (2009, Sep.) N1411: Memory Model Rationale. [Online]. http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1411.htm

[12] IEEE, POSIX.1c, Threads extensions, 1995, IEEE Std 1003.1c.

[13] Dawson Engler and Ken Ashcraft, "RacerX: Effective, Static Detection of Race Conditions and Deadlocks," in *Symposium on Operating Systems Principles (SOSP)*, 2003, pp. 237-252.

[14] Mayur Naik, Alex Aiken, and John Whaley, "Effective Static Race Detection for Java," in *Programming Language Design and Implementation (PLDI)*, 2006, pp. 308-319.

[15] Cormac Flanagan and Stephen Freund, "Type-Based Race Detection for Java," in *Programming Language Design and Implementation (PLDI)*, Vancouver, 2000, pp. 219-232.

[16] Zachary Anderson, David Gay, and Mayur Naik, "Lightweight Annotations for Controlling Sharing in Concurrent Data Structures," in *Programming Language Design and Implementation (PLDI)*, Dublin, 2009.

[17] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard, "Ownership Types for Safe Programming: Preventing Data Races and Deadlocks," in *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2002, pp. 211-230.

[18] A Dinning and E Schonberg, "Detecting access anomalies in programs with critical sections," in *Workshop on Parallel and Distributed Debugging*, 1991, pp. 85-96.

[19] The IDA Pro Disassembler and Debugger. [Online]. http://www.hex-rays.com/idapro/

[20] Koushik Sen, "Race Directed Random Testing of Concurrent Programs," in *Programming Language Design and Implementation (PLDI'08)*, 2008, pp. 11-21.

[21] Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder, "Automatically Classifying Benign and Harmful Data Races Using Replay Analysis," in *Programming Language Design and Implementation (PLDI '07)*, 2007, pp. 22-31.

[22] Donald E. Knuth, *The Art of Computer Programming, Volume 2.*: Addison-Wesley Longman, 1997.

[23] Steven C. Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder P. Singh, and Anoop Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *ISCA '95: International Symposium on Computer architecture*, 1995, pp. 24-26.

[24] Amitabh Srivastava and Alan Eustace, "ATOM: A System for Building Customized Program Analysis Tools," in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, 1994, pp. 196-205.