Aikido: Accelerating Shared Data Dynamic Analyses

Marek Olszewski Qin Zhao David Koh Jason Ansel Saman Amarasinghe

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
{mareko, jansel, qin_zhao, dkoh, saman}@csail.mit.edu

Abstract

Despite a burgeoning demand for parallel programs, the tools available to developers working on shared-memory multicore processors have lagged behind. One reason for this is the lack of hardware support for inspecting the complex behavior of these parallel programs. Inter-thread communication, which must be instrumented for many types of analyses, may occur with any memory operation. To detect such thread communication in software, many existing tools require the instrumentation of all memory operations, which leads to significant performance overheads. To reduce this overhead, some existing tools resort to random sampling of memory operations, which introduces false negatives. Unfortunately, neither of these approaches provide the speed and accuracy programmers have traditionally expected from their tools.

In this work, we present Aikido, a new system and framework that enables the development of efficient and transparent analyses that operate on shared data. Aikido uses a hybrid of existing hardware features and dynamic binary rewriting to detect thread communication with low overhead. Aikido runs a custom hypervisor below the operating system, which exposes per-thread hardware protection mechanisms not available in any widely used operating system. This hybrid approach allows us to benefit from the low cost of detecting memory accesses with hardware, while maintaining the word-level accuracy of a software-only approach. To evaluate our framework, we have implemented an Aikido-enabled vector clock race detector. Our results show that the Aikido enabled race-detector outperforms existing techniques that provide similar accuracy by up to 6.0x, and 76% on average, on the PARSEC benchmark suite.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming – Parallel Programming; D.2.5 [*Software Engineering*]: Testing and Debugging – Debugging Aids; D.4.0 [*Operating Systems*]: General

General Terms Design, Reliability, Performance

Keywords Data Race Detection, Debugging, Multicore

1. Introduction

Despite recent significant efforts to simplify parallel programming, developing parallel applications remains a daunting task. To facilitate parallel software development, a number of tools, such as data-race detectors [28, 31, 30, 19, 27, 7, 16] and atomicity checkers [18, 32, 26, 20] have been proposed. While these tools can provide valuable insights into thread interaction and ultimately help developers discover both correctness and performance bugs, they have not seen widespread use. One reason behind this lack of adoption stems from the fact that for many use cases, they incur either large overheads or do not provide enough accuracy.

Performance issues are endemic to parallel debugging tools that rely on dynamic analyses to monitor access to shared data. These types of analyses, which we call *shared data analyses*, often require instrumentation of all memory operations that read or write to thread private data. Unfortunately, for most programming languages, it is impossible to statically determine which operations access shared memory or what data is shared. As a consequence, many shared data analyses will conservatively instrument all memory accesses and incur large overheads. For example, FastTrack [19], a vector clock based race detector for Java applications, incurs slowdowns of around 8.5x. Tools that operate directly on binaries, such as the Intel Thread Check race detector, incur even larger overheads that are on the order of 200x [30].

To deal with these performance problems, recent work has explored methods for improving race detector performance by trading false negatives for performance using either filtering or sampling techniques [30, 27, 7, 16]. These methods reduce the number of instructions that need to be instrumented in a manner that enables them to detect a subset of errors, typically without introducing false positives. While useful in discovering new bugs during testing, tools built around such techniques offer few benefits to developers that need assistance with debugging a specific bug they are already aware of.

In addition, because they introduction false negatives, such filtering methods are generally not applicable for accelerating existing tools used to verify correctness properties of a parallel program. For example, the Nondeterminator race detector [17] can guarantee that a lock-free Cilk [21] program will execute race free (on all runs for a particular input) provided that it has no false negatives. Likewise, detecting the absence of data races is a useful method of ensuring that an application executing with deterministically ordered synchronization operations (e.g. under a Weak/SyncOrder Deterministic system [29, 13, 25]) will execute deterministically for a

ASPLOS'12, March 3–7, 2012, London, England, UK.

given input. Such a guarantee can only be offered with a race detector that has no false negatives.

1.1 Aikido

This paper presents and evaluates Aikido, a new system and framework for developing efficient and transparent shared data analyses. Rather than instrumenting all memory accesses or a statistical subset, Aikido uses a hybrid of hardware and dynamic binary rewriting techniques to detect shared data at a coarse granularity and the instructions that operate on it. By limiting analysis to just this data, shared data analyses can be accelerated with almost no loss in accuracy. Our prototype system uses the following techniques to enable efficient and transparent shared data analyses:

- **Transparent Per-Thread Page Protection:** To avoid instrumenting all memory accesses, Aikido relies on a novel algorithm for efficiently detecting shared pages using per-thread page protection. Gaining access to per-thread page protection is difficult due to the lack of operating system support (which use a single page table per process). To overcome this challenge, researchers have turned to either modifying the Linux kernel [3], or creating custom compilers that construct multithreaded applications out of multiple processes [4, 24]. Unfortunately, these solutions require either changes to developer tool chains or to the underlying operating system. In contrast, Aikido employs a novel approach that provides per-thread page protection through a hypervisor (AikidoVM) without any modifications. The hypervisor exposes a hypercall-based API that enables user applications to perform per-thread protection requests. Using this approach, Aikido supports tools written to operate on native applications under native operating systems, allowing Aikido to be used today with almost no changes to the development infrastructure.
- **Shared Page Access Detection:** Aikido uses per-thread page protection to dynamically determine which pages each thread is accessing, as well as which instructions are accessing those pages. Pages accessed by more than one thread are deemed shared, and any instructions that access these pages are presented to the shared data analysis for instrumentation. However, because Aikido must detect all instructions that access a shared page, Aikido cannot unprotect a page once it has been found to be shared. Consequently, Aikido uses a novel dynamic rewriting technique to dynamically modify instructions that access shared pages to perform the accesses via a specially mapped *mirror* page. Mirror pages point to the same physical memory as the pages they are mirroring, but are not protected.

We have implemented Aikido as an extension to the Umbra project [34], an efficient framework for writing *shadow value tools* (tools that require per-address metadata) in the DynamoRIO [9] binary instrumentation system.

1.2 Experimental Results

We evaluate Aikido on applications from the PARSEC benchmark suite [6], using a dynamic race detector analysis that implements the FastTrack algorithm [19]. Our results show that Aikido improves the performance of a shared data analysis by 76% on average, and by up to 6.0x when running on applications that exhibit little data sharing.

1.3 Contributions

This paper makes the following contributions:

- **Accelerating Shared Data Analyses:** It presents a new system for accelerating shared data dynamic analyses and evaluates the performance of such a system.
- **Transparent Per-thread Page Protection:** It presents a new mechanism for providing per-thread page protection support to unmodified applications running under unmodified operating systems using a hypervisor.
- **AikidoVM:** It presents AikidoVM, a hypervisor that enables user applications to transparently perform per-thread protection requests.
- **Efficient Shared Page Detection:** It presents a new method of using per-thread page protection to dynamically and efficiently detect shared pages of memory.
- **Experimental Results:** It presents experimental results demonstrating Aikido's ability to enable efficient and transparent shared data analysis tools.

2. Background

In this section, we provide background on DynamoRIO and the Umbra shadow memory framework. Aikido uses these systems to perform dynamic instrumentation and as a basis for providing mirror-pages, which will be described in Section 3.

2.1 DynamoRIO

DynamoRIO [9, 10] is a dynamic instrumentation and optimization framework that supports both 32 and 64 Windows and Linux platforms. DynamoRIO allows programs to be dynamically instrumented both transparently and efficiently. Rather than executing a program directly, DynamoRIO runs application instructions through a *code cache* that is dynamically filled one basic block at a time. As basic blocks are copied into the code cache they are modified both by the DynamoRIO and by *tools* in the DynamoRIO system that are responsible for performing instrumentation and optimization. Blocks in the code cache are linked together via direct jumps or fast lookup tables so as to reduce the number of context switches to the DynamoRIO runtime system. In addition, DynamoRIO stitches sequences of hot code together to create single-entry multiple-exit traces that are stored in a separate trace cache for further optimization. DynamoRIO allows users to build DynamoRIO tools using the APIs provided. These tools can manipulate the application code by supplying callback functions which are called by DynamoRIO before the code is placed in either code caches.

2.2 Umbra

Umbra [33, 34] is an extension to the DynamoRIO binary instrumentation system that allows building efficient *shadow value tools* [8]. Shadow memory tools store information about every piece of application data, both dynamically and statically allocated. This information is called *shadow metadata*. Umbra supports arbitrary mappings from configurable number of bytes of application data to a configurable number of bytes of shadow metadata. This shadow metadata can be updated by DynamoRIO tools at the granularity of individual instructions as the application executes. Shadow value tools have been created for a wide variety of purposes, including finding memory usage errors, tracking tainted data, detecting race conditions, and many others. Like many software-based shadow value tools, Umbra dynamically inserts (using DynamoRIO) additional instructions into the target application to be executed along with the target code. This inserted instrumentation allows a shadow value tool to update shadow metadata during program execution.

The inserted instrumentation code performs three tasks: mapping an application data location to the corresponding shadow metadata location, updating metadata, and performing checks. These tasks represent the major source of runtime overhead in Umbra tools.

Umbra uses a translation scheme that incurs modest metadata mapping overhead. Umbra leverages the fact that application memory is typically sparsely populated, with only a small number of regions in memory that contain densely laid out memory allocations (such as the stack, heap, data and code segments). This observation allows Umbra to use a simple yet efficient shadow memory mapping scheme that uses such densely populated regions as a mapping unit: for each densely populated region of memory, the Shadow Metadata Manger allocates a shadow memory region and associates it with the application memory region. Umbra associates each region with one shadow memory region and maintains a table of offsets to convert an application address in a region to an address in the corresponding shadow memory region. By focusing on densely populated regions of memory rather than on the entire address space, Umbra is able to scale to large 64-bit address spaces without requiring multiple translation steps or extremely large tables.

Much of the performance of Umbra comes from many layers of caching for offsets from application data to shadow memory regions. In the common case, a lookup of shadow data will occur in an inlined memoization cache that is inserted directly into the application code. In the less common case, lookups will occur in one of three levels of thread-local caches that are checked in a *lean procedure* that executes without a stack and requires only a partial context switch away from user code. In rare cases, lookups need to be performed in a more expensive function that requires a full context switch to invoke.

3. Design and Implementation

This section describes the design and implementation of the complete Aikido system, which can be used to accelerate DynamoRIO tools that perform shared data analysis.

3.1 Overview

Figure 1 displays an overview of the Aikido system. The system is comprised of a hypervisor, a modified version of the DynamoRIO system, a sharing detector, and a user specified instrumentation tool that performs a shared data analysis.

At the base of the system is the *AikidoVM* hypervisor which transparently provides the per-thread page protection mechanism to any userspace application running under the guest operating system. AikidoVM exposes this new feature through the userspace *AikidoLib*, which communicates with the hypervisor using *hypercalls* that bypass the guest operating system. This library can be used by any userspace application wishing to use per-thread page protections. In Aikido, this library is linked with a version of DynamoRIO that is modified to support tools that wish to set per-thread page protection on any portion of a target application's address space. Using this mechanism, the Aikido sharing detector, or *AikidoSD*, efficiently constructs a dynamic mapping of instructions to pages, which can be used to determine which pages of memory are shared within the target application, and which instructions access those pages. Once a page is discovered to be shared, AikidoSD presents all instructions that access it to the DynamoRIO tool for instrumentation. In this way, a DynamoRIO tool running under the Aikido system will only be presented with instructions that are guaranteed to access shared pages of memory.

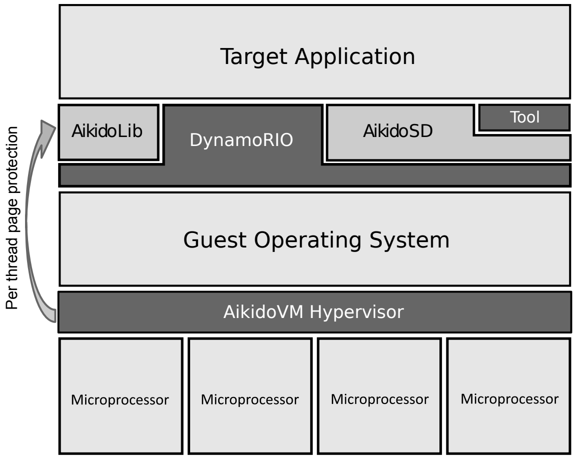

Figure 1. An overview of the Aikido system

3.2 AikidoVM

In this section, we discuss the design and implementation of the AikidoVM hypervisor. We first begin with an overview of the system. Next, we provide some background on memory virtualization and conclude with a description of our implementation.

3.2.1 Overview

AikidoVM is based on the Linux KVM virtual machine, a robust and lightweight virtual machine manager that supports a number of operating systems. AikidoVM extends KVM to transparently add support for per-thread page protection for user space applications executing in the guest operating system. AikidoVM targets Intel x86-64 processors and uses Intel's VMX virtualization extensions to virtualize privileged operations.

3.2.2 Traditional Memory Virtualization

Most operating systems today use page tables to translate virtual addresses to physical addresses. Each process is assigned its own page table, giving it a virtual address space that is isolated from other processes. To construct a shared address space for multiple threads executing in a single process, modern operating systems share the process's page table with all threads within the process. In addition to mapping virtual to physical addresses, page tables also store a protection level for each virtual page that is enforced by the machine hardware. Protection levels are specified by three bits that control whether a page is present (i.e. readable), writable, and userspace accessible. Because all threads within a process share a single page table, all threads running in the same address space must observe the same protection on all pages mapped in the address space.

In a typical hypervisor, a guest virtual machine is given the illusion of executing in a private physical machine that is isolated from all other virtual machines on the system. To guarantee memory isolation, hypervisors present each virtual machine with a *guest physical* address space by adding an extra level of indirection to the address translation. This translation maps *guest physical addresses*, which provide a software abstraction for the zero-based physical address space that the guest operating system expects, to *machine addresses*, which refer to actual memory in the physical machine, Subsequently, the guest operating system maintains a mapping of *guest virtual addresses* to *guest physical addresses* for each process, to complete the translation.

To perform this translation, today's hypervisors will either maintain *shadow page tables* in a software MMU which directly maps guest virtual addresses to machine addresses, or on newer hardware with support for *nested paging* (e.g. using Intel's VMX EPT and AMD-V extensions), maintain a page table that maps guest physical to machine addresses, and uses the hardware MMU to perform the complete address translation by walking both the guest OS and the hypervisor's page tables. In this work, we refer only to the former shadow paging strategy, though we believe that our techniques are generally applicable to hardware MMU virtualization systems based on nested paging as well.

In a software MMU, shadow page tables provide an efficient way of mapping guest virtual addresses directly to machine addresses. The processor uses these shadow page tables during execution instead of the guest page tables, caching the translations in the TLB allowing most memory accesses to avoid virtualization overhead. The shadow page tables must be kept in sync with the guest operating system's page tables by monitoring the guest's changes to these page tables. This is typically done by write-protecting the pages that comprise the guest page tables, allowing the hypervisor to intercept any changes made to them. Once a write is detected, the hypervisor will emulate the write on behalf of the guest and update the shadow page table accordingly. The hypervisor also intercepts all context switches performed by the guest OS by trapping on updates to the control register (CR3 on x86) that points to the root of the page table. On x86, this can be done using Intel VMX virtualization extensions to request an exit from the virtual machine on any write to the CR3 register. This allows the hypervisor to switch shadow page tables to correspond with the guest page tables on context switches.

3.2.3 Memory Virtualization in AikidoVM

Rather than maintaining a single shadow page for every page in the guest operating system, AikidoVM maintains multiple shadow pages, one for each thread in an Aikido enabled guest process. Each copy of a shadow page table performs the same mapping from virtual to physical memory, so each thread executing in the guest process will share the exact same view of memory. As with a traditional hypervisor, AikidoVM write protects the guest operating system's page table pages to detect updates to them. If a change is detected, AikidoVM updates each of the shadow page tables for each of the threads.

Additionally, AikidoVM must intercept all context switches between threads within the same address space so that it is able to switch shadow page tables. Unfortunately, modern operating systems do not write to the CR3 register on such context switches because such switches do not require a change in address space. Therefore, it is not possible to use traditional mechanisms for requesting an exit from the virtual machine during such context switches. As a result, we currently insert a hypercall into the context switch procedure of the guest operating system to inform AikidoVM of the context switch. However, to support truly unmodified operating systems, we are in the process of updating our system to request virtual machine exits on writes to the GS and FS segment registers. These registers are used in modern operating systems to specify thread local storage for each thread on x86 systems, and are updated on all context switches, including context switches between threads in the same process. Alternatively, a hypercall can be inserted into an unmodified guest operating system at runtime by inserting a trampoline-based probe [11] into the context switch function.

3.2.4 Per-Thread Page Protection

By maintaining multiple shadow page tables, AikidoVM is able to set different page protections for each thread in the Aikido-enabled guest userspace process. AikidoVM maintains a per-thread protection table that is queried and used to determine the protection bits on any new page table entry in a thread's shadow page table (see Figure 2). When a page fault occurs, AikidoVM first checks the current thread's protection table before checking the guest operating system's page table to determine whether the page fault was caused by an Aikdio initiated protection, or by regular behavior of the guest application and operating system. Being able to differentiate between the two types of page faults has proven to be extremely useful for our system, as it has greatly simplified the number of changes required to support Aikido in DynamoRIO.

When a guest operating system makes a change to its page tables, AikidoVM will trap and apply the changes to each of the thread's shadow page tables, updating the page protection bits according to the contents of the per-thread protection table. In order to do so, it must maintain two reverse mapping tables that map guest physical addresses to the per-thread shadow pages and to Aikido's per thread protection table.

3.2.5 Delivering Page Faults

For non-Aikido caused page faults, AikidoVM delivers the page fault to the guest operating system, allowing the guest operating system to handle it as it would normally. For Aikido related page faults, AikidoVM reuses the guest's signal delivery mechanisms by injecting a fake page fault at a special address predetermined with the Aikido library. To prevent the guest operating system from ignoring this page fault, Aikido requires that this predetermined address is mapped in the guest userspace application's address space and has the same page protection as the faulting location. This is achieved in the Aikido library by allocating a page with no write access and one with no read access and reporting both page addresses to the AikidoVM on initialization. To report the true faulting address, AikidoVM records the address at a memory location that is registered with the Aikido library. Subsequently, the signal handler that is handling the fault can both check whether a fault is Aikido related, and obtain the faulting address using the `aikido_is_aikido_pagefault()` function in the Aikido library.

3.2.6 Handling Guest Operating System Faults

Because the guest operating system is unaware of any additional page protection set by the guest userspace application through AikidoVM, it may cause a page fault when accessing a page in the application's address space, which it will not be able to handle. As a

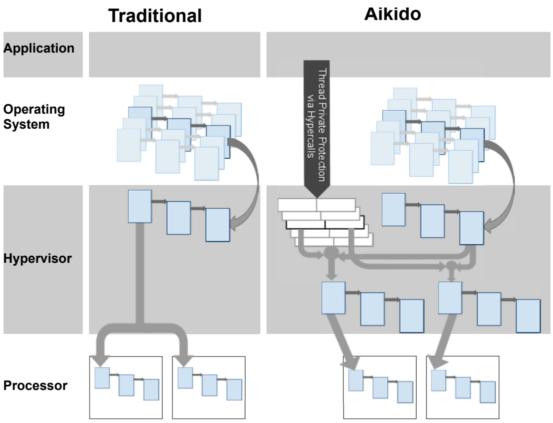

Figure 2. Compares the shadow page tables maintained by a traditional hypervisor to that of AikidoVM. A traditional hypervisor maintains one shadow page table for every page table in the guest operating system, while the AikidoVM maintains multiple shadow pages (one for each thread sharing the guest page table) for each guest page table, and multiple per-thread page protection tables that store the desired page protection.

result, AikidoVM detects such page faults and emulates the instruction in the guest operating system that caused the page fault. To prevent subsequent page faults (and further emulation) to the same page, AikidoVM will also temporarily unprotect the page. Unfortunately, because Intel's VMX extensions cannot be used to request VM exits when the guest operating system returns execution to the guest userspace application, AikidoVM has no way of reprotecting these pages before the guest userspace application resumes. Therefore, when temporarily unprotecting a page, AikidoVM ensures that it is not userspace accessible (i.e.: USER bit is not set). Thus, a page fault will be triggered when the userspace application attempts to access a temporarily unprotected page, at which point AikidoVM will restore the original protections to all pages accessed by the guest operating system.

3.3 AikidoSD

In this section, we describe the Aikido sharing detector (AikidoSD), the component that is responsible for instrumenting and tracking the target application's accesses to memory to determine which pages are shared. Once AikidoSD finds a shared page, it presents all instructions that access the page to a user supplied DynamoRIO tool for instrumentation.

3.3.1 Overview

AikidoSD is built on top of a version of the Umbra shadow memory framework that has been modified to map application addresses to *two* shadow addresses (instead of just one). The first of these two shadow memories is used to store metadata for the shared data analysis tool and to track the shared state of each page by the Aikido sharing detector. The second shadow memory is used to construct *mirror* pages. Mirror pages duplicate the content of the target application's address space without any additional protections added by AikidoSD. This provides AikidoSD a mechanism for accessing application data while the original location is protected.

3.3.2 Detecting Sharing

The Aikido sharing detector is designed with the key goal of ensuring that instructions that access only private data incur close to no overheads. With this goal in mind, the sharing detector relies on thread-private page protection to page protect each module of memory mapped by the target application. When the target application begins execution, AikidoSD will page protect all mapped pages in the target application's address space. When a thread accesses a page for the first time (first scenario in Figure 3), AikidoSD catches the segmentation fault delivered by AikidoVM, unprotects the page, sets the page's status to private and finally resumes execution. All subsequent accesses to the page by the thread will continue without additional page faults or overhead (second scenario in Figure 3). In this way, the Aikido sharing detector requires just one page fault per thread for each page that will remain private to a single thread throughout the execution of the target application.

If another thread accesses a page that has been previously marked private (third scenario in Figure 3), AikidoSD will catch the resulting segmentation fault and atomically set the state of the page to shared and globally protect the page so that it is not accessible by any thread. Once the page is globally inaccessible (last scenario in Figure 3), AikidoSD will receive a segmentation fault every time a new instruction attempts to access a shared page. At this point, any instruction that causes a page fault is known to be accessing shared data and can be instrumented by the DynamoRIO tool. This is accomplished by deleting all cached basic blocks that contain the faulting instruction and re-JITing them to include any desired instrumentation by the tool. Additionally, since the page being accessed by the faulting instruction is no longer accessible, AikidoSD must update the code in the basic block such that the instruction accesses all memory via mirror pages. This is achieved by either modifying the immediate effective address in direct memory instructions, or for indirect memory instructions, inserting a sequence of instruction that translate the base address of the instruction. AikidoSD uses the same efficient sequence of instructions used by Umbra to perform the translation. Finally, for indirect memory instructions only, which may continue to access shared or private data, AikidoSD will also emit a branch that checks whether the instruction is accessing a shared or private page on subsequent invocations. This check allows AikidoSD to jump over the instrumentation and memory address redirection code when the instruction accesses a private page. Figure 4 outlines the code sequence emitted by AikidoSD when instrumenting an instruction.

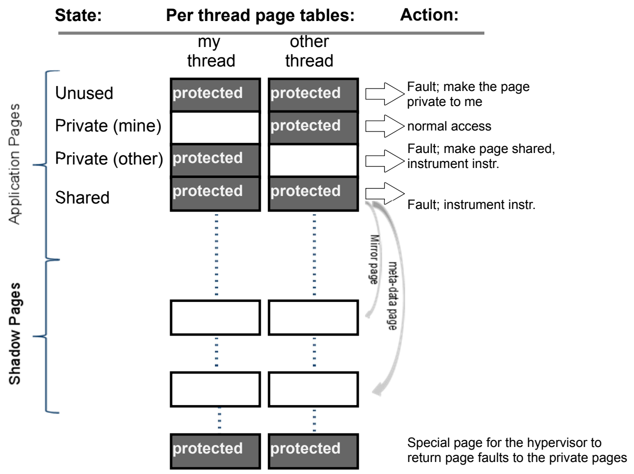

Figure 3. An example page protection state and the corresponding actions that will occur on a memory access to each page.

```
// Translate address from application to shadow
shd_addr = app_to_shd(app_addr);
if (page_status(shd_addr) == SHARED) {
  // Execute instrumentation emitted by the tool
  // (e.g. race detection instrumentation)
  tool_instrumentation_code(shd_addr);
  // Translate address from app to mirror
  mirror_addr = app_to_mirror(app_addr);
  // Redirect memory access to mirror page
  redirect(APP_INSTR, mirror_addr);
} else {
  // Execute the original application instr
  APP_INSTR;
}
```

Figure 4. Pseudocode of the instrumentation inserted at an instruction that has accessed a shared page

3.3.3 Mirror Pages

AikidoSD starts by mirroring all allocated pages within the target application's address space at program start. This is done by creating a new backing file for every segment of memory, copying the contents of memory to that file, and finally mmapping the file both over the originally allocated segment, and into the mirrored memory region for that segment.

After the application begins to execute, AikidoSD will intercept all `mmap` and `brk` system calls to ensure that all new memory allocations are correctly mirrored. For non-anonymous shared mmaps, AikidoSD simply performs the same mmap into the mirrored pages. For private (copy-on-write) mappings (including anonymous mappings), AikidoSD creates a new backing file, copies the contents of the original file and mmaps the new backing file twice using shared mappings. The new backing file is required so that its contents can be shared without affecting any other mappings of the original file. To handle extensions to the heap, AikidoSD intercepts all `brk` system calls and emulates their behaviour using mmaped files, which are again mapped twice in the address space.

3.4 Changes to DynamoRIO

DynamoRIO often needs access to the entire contents of the target application's address space. Furthermore, DynamoRIO will write-protect certain pages in the application's address space (to detect self-modifying code) and can get confused when it discovers unexpected page protections. As a result, we made a number of changes to DynamoRIO to have it support tools that use Aikido's per-thread page protections.

Most significantly, we updated the master signal handler to differentiate between Aikdio and non-Aikido page faults. When an Aikido page fault is discovered, DynamoRIO will check the location of the faulting instruction. If the page fault was triggered by the target application, DynamoRIO will forward the signal to the Aikido sharing detector for further examination. If it occurred within DynamoRIO or within a DynamoRIO tool, DynamoRIO will unprotect the page for that thread, and will add the page address to a list of unprotected pages. When returning control back to the target application, DynamoRIO will iterate through this list and reprotect all the pages that needed uprotecting.

3.5 Portability

While our implementation has been in the context of a Linux guest operating system, we believe that the approach we have taken can be easily applied to other operating systems, such as Microsoft Windows or Mac OS X. Indeed, great care was taken to ensure that the implementation is as portable as possible. Because of this, we expect that our system will operate almost out of the box when running on the Windows version of DynamoRIO.

4. Aikido Race Detector

To demonstrate the efficacy of Aikido in accelerating shared data analyses, we implemented a race detector that takes advantage of the sharing detection in Aikido to only check for races that occur in shared data. Our race detector implements the FastTrack [19] algorithm, an efficient *happens-before* race detector. We first provide background on the FastTrack algorithm and then present details of how we incorporated it into Aikido.

4.1 FastTrack Algorithm Background

FastTrack operates by computing a *happens-before* relation on the memory and synchronization operations in a program execution. A happens-before relation is a partial ordering on the instructions in the program trace that can be used to identify races. A race occurs when two instructions are not ordered by this happens-before relation, they read or write to the same variable, and one instruction is a write. FastTrack computes the happens-before relation by keeping track of a vector clock for each thread. Each thread is given a logical clock that is incremented each time a synchronization event occurs. Each thread's vector clock records the clock of every thread in the system. Entries for other threads in a vector clock are updated as synchronization occurs.

To update the thread vector clocks, each lock also maintains a vector clock, which serves as a checkpoint of how far along each thread was when it last accessed the lock. When a thread performs a lock acquire, the lock's vector clock is used to update the thread's vector clock, thus passing on information about happens-before invariants originating from past holders of the lock. Upon lock release, the lock's vector clock is updated using the thread's vector clock to record the event. Finally, after the lock is released, the thread increments its own clock.

Each variable in the program has two vector clocks, one for reads and one for writes. To detect races during program execution, FastTrack compares the clocks of each variable to the clock of the accessing thread. This check ensures that a read must occur after the last write from any other thread to that variable, and a write must occur after any read or write from any other thread. Upon a read or write, the entry corresponding to the thread performing the access in the variable's read vector clock or write vector clock is updated to contain the thread's clock.

FastTrack uses a novel optimization of this algorithm revolving around the concept of *epochs*, which reduces the metadata for variables by only recording information about the last access when a totally ordered sequence of accesses occurs. Like other *happens-before* race-detectors, FastTrack does not report false positives and can only find races that could manifest for a particular tested execution schedule for a given input.

4.2 Aikido Race Detector

There were a number of challenges in implementing FastTrack in DynamoRIO and Aikido. The largest issue resulted from operating on unmodified x86 binaries. While FastTrack operates on variables in Java, x86 binaries contain instructions that operate directly on memory and may contain overlapping accesses to data. To simplify the design, we divided the address space into fixed size (currently 8-byte) blocks that we considered "variables" for the FastTrack algorithm. This has the downside of possibly introducing false positives for tightly packed data, but simplifies the allocation of shadow memory to keep track of the metadata required by FastTrack. To store metadata we use thread-local storage for thread metadata, a hash-table for per-lock metadata, and shadow memory for storing metadata about each block of memory. The mechanism for providing shadow memory is summarized in Section 2.2. We use DynamoRIO to insert the instrumentation needed to perform metadata updates and checks for races.

Additionally, to simplify clock inheritance and metadata updates on thread creation, we serialize all thread creation using locking. We also added special handling to avoid reporting spurious races from the C/C++ libraries, the pthread library, and the loader.

When running under AikidoVM, our race detector only instruments instructions that access shared data and only maintains the epoch metadata for shared data. Initially, instructions are not instrumented and metadata is not maintained for memory. When Aikido finds shared data, the instructions accessing that data are instrumented and the metadata for the shared data is initialized. This achieves our performance improvements, but may prevents us from instrumenting instructions before they perform their *first* access to shared data (i.e. before we detected that the data was shared). This can potentially introduce a false negative if a race occurs between two instructions that are each the first instruction in their respective threads to access the same page.

5. Evaluation

In this section we evaluate the performance of Aikido on the race detector described in Section 4.

5.1 Experimental Setup

We tested Aikido on a quad-socket Xeon X7550 system running Debian 5.0 using 8 threads. We ran our race detector on 10 benchmarks from the PARSEC 2.1 benchmark suite [5, 6] using the simsmall input set.

5.2 Results and Analysis

Figure 5 compares the performance of our Aikido-FastTrack race detector versus the regular FastTrack tool. Both numbers are normalized to the native execution time, resulting in a slowdown number where lower is better. Aikido is able to improve the performance of 6 of the 10 benchmarks. In the remaining 4 benchmarks it exhibits little change for 3 benchmarks, and a 3% increase in overhead for fluidanimate. On average, Aikido is able to speed up the FastTrack race detector by 76%, and up to 6.0x for the raytrace benchmark.

Figure 6 shows the percentage of accesses in each benchmark that target shared pages. We see that the three top performing benchmarks: raytrace, swaptions, and blackscholes, all exhibit low levels of sharing between threads, which permits the up to 6.0x improvement in performance when run with Aikido. Likewise, Aikido's poor performance on fluidanimate can be explained in part by the large amount of sharing in the benchmark.

Table 1 shows how the overheads for our two worst performing benchmarks (fluidanimate and vips) change at different thread counts. For both benchmarks, the overheads are significantly lower at two threads than at 8 threads. Additionally, at thread counts of both two and four, Aikido-FastTrack always outperforms FastTrack for both of the two benchmarks. At two threads, Aikido-FastTrack is up to 45% faster than the FastTrack algorithm for vips.

Table 2 outlines a number of other statistics for each of the PARSEC benchmarks recorded while running the Aikido-FastTrack tool. The first column lists the number of memory accessing instructions executed by each of the benchmarks. This represents the number of instructions that would typically need to be instrumented by a race detector to conservatively instrument all accesses to shared memory. The second column displays the dynamic execution count of all instructions that were found to access shared pages, while the third column lists the number of times these instructions accessed a shared page, which indicates the number of times the Aikido-FastTrack instrumentation was executed. Lastly, the fourth column displays the number of segmentation faults delivered by the AikidoVM hypervisor. This number also represents

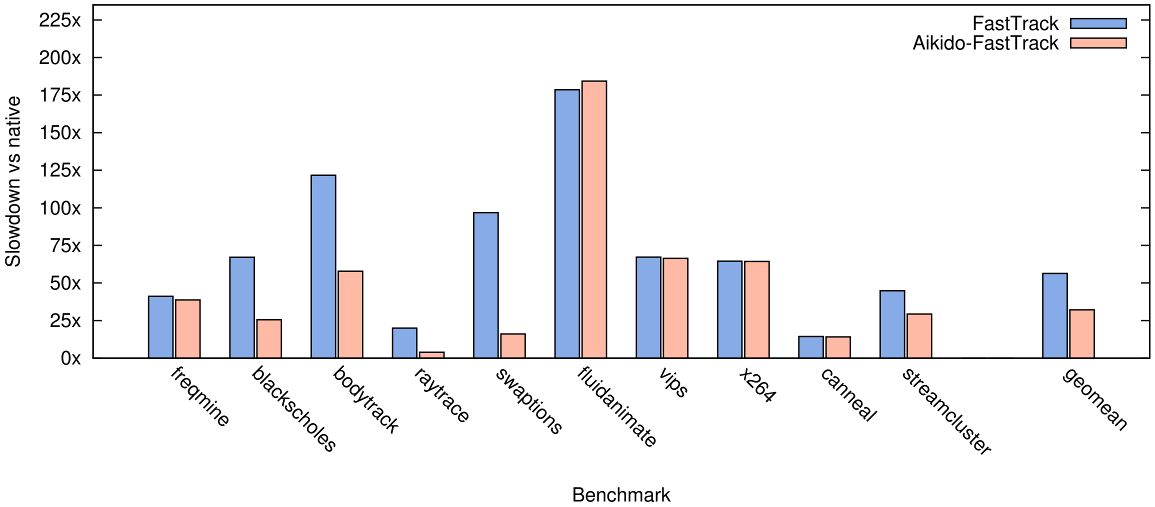

Figure 5. Performance of the FastTrack race detector algorithm when running both with and without the Aikido. Both numbers are normalized to the native execution time running without race detection (lower is better).

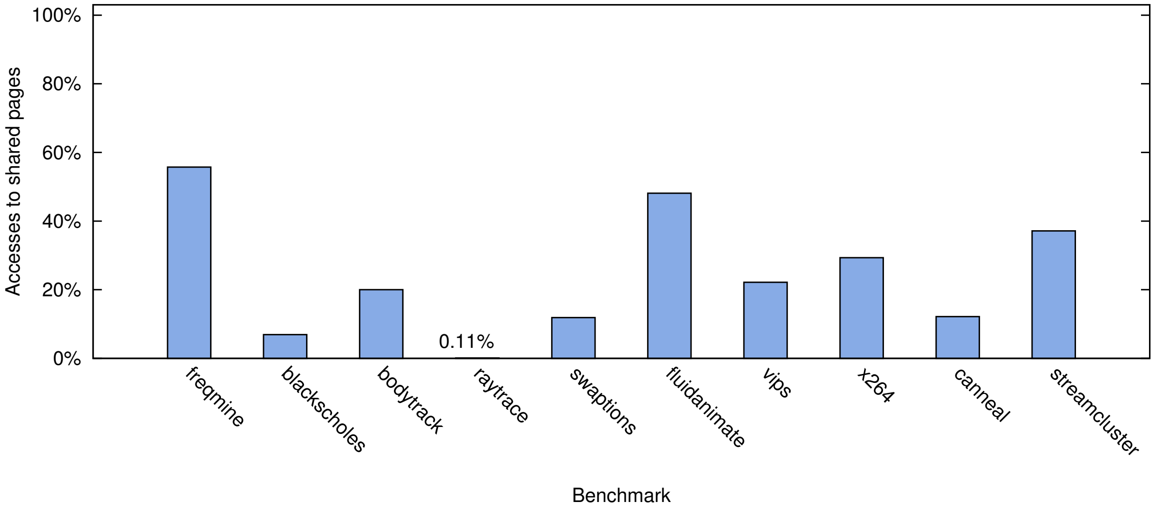

Figure 6. Percentage of accesses that target shared page for each benchmark.

the number of times DynamoRIO needed to rebuild a basic block (and potentially a trace).

By comparing the first two columns, it is possible to determine an upper bound, for each benchmark, on the amount of instrumentation that can be avoided using our approach when running with 8 threads. Across all of the benchmarks, our technique yields a geometric mean reduction of 6.75x in the number of memory accessing instructions that need to be instrumented.

5.3 Detected Races

We compared the outputs between both the FastTrack and Aikido-FastTrack tools to check that both tools were detecting the same races. In both tools, we found a number races in the PARSEC benchmark suite. We believe most of the races found were considered benign and the result of custom synchronization in libraries or "racy reads" where outdated results are acceptable to the programmer.

An example race we found was in the random number generator (based on Mersenne Twister) in the canneal benchmark. This might be considered a benign race, since there is no correct result of a random number generator. However, it is not clear that the known statistical properties of the Mersenne Twister function hold in a runtime with data races.

6. Discussion

AikidoSD determines whether a page is shared by waiting until two threads have accessed the same page, triggering two segmentation faults. Because of this, any tool built on top of Aikido may not instrument the first two instructions that access a particular piece of shared data. This fact introduces a small number of false negatives to such tools. While any degree of false negatives can be unde-

	2 Threads	4 Threads	8 Threads
fluidanimate (FastTrack)	55.79x	127.62x	178.60x
fluidanimate (Aikido-FastTrack)	48.11x	110.65x	184.33x
vips (FastTrack)	45.52x	53.34x	67.24x
vips (Aikido-FastTrack)	31.5x	35.96x	66.37x

Table 1. Performance overheads over native execution for the FastTrack and Aikido-FastTrack race detectors on the fluidanimate and vips benchmarks at different thread counts.

Benchmark	Instrs. Referencing Memory	Instrumented Instrs.	Shared Page Accesses	Segmentation Faults
freqmine	1,167,712,401	742,195,956	651,009,521	24,880
blackscholes	105,944,404	7,395,315	7,340,038	889
bodytrack	384,925,938	83,514,877	77,116,382	8,993
raytrace	13,186,394,771	16,920,360	14,419,167	23,350
swaptions	350,009,582	58,348,333	4,160,2078	1,778
fluidanimate	556,317,760	356,317,897	267,758,255	11,054
vips	1,044,161,383	253,794,130	231,533,572	10,227
x264	241,456,020	82,561,137	70,813,420	32,616
canneal	560,635,087	69,108,663	68,153,896	23,049
streamcluster	1,067,233,548	403,953,097	396,265,668	5,918

Table 2. Instrumentation statistics recorded while running the Aikido-FastTrack tool.

sirable, we were careful to design Aikido such that it introduces false negatives in a well-defined and targeted manner. We believe that our approach represents a point in the performance/soundness trade-off space is desirable. By limiting our false negatives to well defined application behaviors, false negatives can be handled separately in cases where they present a problem.

For example, in the context of Weak/SyncOrder Deterministic systems [29, 13, 25]), which will execute an application deterministically for a given input only if the program is guaranteed to be race free for that input, Aikido-FastTrack cannot provide a guarantee on the required race freedom. However, it can still be used to provide a guarantee on determinism if the underlying deterministic multithreading system also ensures that the first two memory accesses to any memory location are ordered deterministically. We believe that ordering such accesses can be achieved cheaply using off-the-shelf process-wide (i.e. non per-thread) page protection available in today's operating systems. We hope that similar workarounds can be applied for other scenarios where sound shared data analyses are currently required.

7. Related Work

We divide the discussion of related work into three areas. First, we cover related systems to AikidoVM's technique for providing per-thread page protection. Next, we discuss systems that also use mirror-pages, and finally we cover systems related to the race detector we implemented in Aikido.

7.1 Private Page Protections

Overshadow [12] is a hypervisor that isolates application memory from the operating system by encrypting any application memory that the operating system must be able to access. This provides a layer of security in case the operating system becomes compromised. Overshadow does not provide per-thread page protection, but uses a technique similar to AikidoVM's to present its encrypted and plain text views of memory to the system. Overshadow maintains multiple shadow page tables and encrypts (or decrypts) pages depending on which part of the system is attempting to read or write to a page.

The Grace [4] and DTHREADS [24] projects use per-thread page tables for the purpose of implementing language features and for deterministic multithreading [29, 14], in contrast to the shared data analyses targeted by Aikido. These systems achieve per-thread page tables by converting all threads into separate processes and taking steps to create the illusion of a single process and address space. While it does not require a custom hypervisor, this technique is operating system specific and may find it difficult to maintain the illusion of a single process for more complex applications. For example, without special care, file descriptors created in one process after all processes are forked, will not be visible in the other processes.

The dOS [3] project uses per-thread page tables to track ownership of pages in order to enforce deterministic ordering of memory operations. dOS implements per-thread page tables through extensive modifications to the 2.6.24 Linux kernel.

Finally, SMP-ReVirt [15] uses per-processor private page mappings within a modified Xen Hypervisor for efficient full-system record/replay that runs on commodity hardware. SMP-ReVirt implements the CREW [23] protocol at a page-level granularity to track and later replay page-ownership transitions. Like Aikido, SMP-ReVirt uses shadow page tables within the hypervisor to achieve the private page mappings. However, unlike Aikido, these mappings are per processor rather than per (guest) thread. While Aikido must determine whether each thread has accessed a particular page, the thread abstraction is irrelevant for STMP-ReVirt to replay the execution of an entire operating system, allowing it to track page ownership changes on (guest) physical pages.

7.2 Address Space Mirroring

Abadi et. al present a Software Transactional Memory (STM) system [1] that uses off-the-shelf page-level memory protection to lower the cost of providing strong atomicity guarantees for C# applications. Under this system, the C# application heap is mapped twice in virtual memory. One mapping is used by code executing within a transaction while the other mapping is used during non-transactional execution. Pages in the latter mapping are then dynamically page protected before every access to the heap within a transaction to ensure that all potentially conflicting non-transactional accesses trigger a segmentation fault. This allows the STM system to detect conflicting accesses to memory that occur between concurrently executing transactional and non-transactional code. Because such conflicts tend to be rare, the strategy achieves low overheads. Furthermore, in cases where a large amount of conflicts do occur, the system can patch instructions that frequently cause segmentation faults to jump to code that performs the same operation but within a transaction.

Aikido differs from this work in the following ways. First, Aikido maintains different page protections for each thread for the main (non-mirrored) pages of memory. This addition is very important to be able to efficiently determine which pages of memory are shared, as it enables accesses to private data to execute without redirection. Additionally, because Aikido must redirect all memory accesses that access shared memory (rather than just accesses that frequently conflict with transactions), it must dynamically rewrite all accesses to protected pages instead of just a small number of frequently occurring ones. Doing so efficiently is a significant challenge. Finally, Aikido's use of a hypervisor and binary rewriting transparently bring some of the benefits of mirror pages to a large number of applications (e.g. binary applications) and potential shared data analyses.

7.3 Race Detectors

Foremost in relevance to Aikido's race detector is the FastTrack [19] algorithm on which our race detector is based. We differ from FastTrack in that we operate directly on x86 binaries, as opposed to on Java variables, and that we only instrument accesses to shared data as opposed to all variables.

FastTrack (and our system) computes its happens-before [22] relation using a vector-clocks [28] based approach. This strategy has the advantage that it is precise, and does not introduce false positives. Alternate approaches, such as Eraser's LockSet algorithm [31], that try to associate locks with data protected by them, can report false positives. Other techniques in this area use a hybrid of these two approaches or explore the trade-offs between performance and precision by using sampling or heuristics to decide what to instrument [30, 27, 7, 16].

In each of these systems, the race detector's ability to detect races is often tied to the particular execution schedule seen by the application during execution, which may not be deterministic. This is in contrast to special purpose race detectors such as Nondeterminator and CilkScreen [17, 2] which are schedule independent and can have no false negatives for a subset of legal Cilk and Cilk++ code [21].

8. Conclusions

We have presented Aikido, a novel framework that enables the development of efficient and transparent shared data analysis tools. Aikido achieves transparency through the use of a custom hypervisor that exposes per-thread hardware protection mechanisms. This enables Aikido to use a hybrid hardware and dynamic binary instrumentation approach to dynamically detect shared data in parallel programs. Our results demonstrate that Aikido can be used to speed up shared data analyses. In particular, we have shown that it is possible to speed up existing race detectors by focusing attention to data races caused by instructions that access shared pages. The resulting race detector offers significant performance improvements over existing techniques that provide similar accuracy.

Enabling faster and more transparent shared data analysis can enable new, practical, analysis techniques. The end goal is to enable a new set of tools that helps developers write, understand, debug and optimize parallel programs.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable and insightful comments. We also thank Stelios Sidiroglou for providing invaluable feedback on the manuscript and for numerous helpful discussions. This work was supported in part by NSF grant CCF-0832997, DOE SC0005288, and DARPA HR0011-10-9-0009.

References

[1] Martín Abadi, Tim Harris, and Mojtaba Mehrara. Transactional Memory with Strong Atomicity Using Off-the-Shelf Memory Protection Hardware. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 185–196, New York, NY, USA, 2009. ACM.

[2] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Charles E. Leiserson. On-the-Fly Maintenance of Series-Parallel Relationships in Fork-Join Multithreaded Programs. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '04, pages 133–144, New York, NY, USA, 2004. ACM.

[3] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D. Gribble. Deterministic Process Groups in dOS. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–16, 2010.

[4] Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. Grace: Safe Multithreaded Programming for C/C++. In *ACM SIGPLAN Conference On Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 81–96, New York, NY, USA, 2009. ACM.

[5] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[6] Christian Bienia and Kai Li. PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, June 2009.

[7] Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. PACER: Proportional Detection of Data Races. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 255–268, New York, NY, USA, 2010. ACM.

[8] D. Bruening and Qin Zhao. Practical Memory Checking with Dr. Memory. In *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*, pages 213 –223, april 2011.

[9] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An Infrastructure for Adaptive Dynamic Optimization. In *International Symposium on Code Generation and Optimization*, San Francisco, Mar 2003.

[10] Derek Bruening, Vladimir Kiriansky, Timothy Garnett, and Sanjeev Banerji. Thread-Shared Software Code Caches. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 28–38, Washington, DC, USA, 2006. IEEE Computer Society.

[11] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic Instrumentation of Production Systems. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '04, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.

[12] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. In *International conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, pages 2–13, New York, NY, USA, 2008. ACM.

[13] Heming Cui, Jingyue Wu, Chia-Che Tsai, and Junfeng Yang. Stable Deterministic Multithreading Through Schedule Memoization. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–13, 2010.

[14] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. DMP: Deterministic Shared Memory Multiprocessing. In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 85–96, 2009.

[15] George W. Dunlap, Dominic G. Lucchetti, Michael A. Fetterman, and Peter M. Chen. Execution Replay of Multiprocessor Virtual Machines. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 121–130, New York, NY, USA, 2008. ACM.

[16] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective Data-Race Detection for the Kernel. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[17] Mingdong Feng and Charles E. Leiserson. Efficient Detection of Determinacy Races in Cilk Programs. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 1–11, Newport, Rhode Island, June22–25 1997.

[18] Cormac Flanagan and Stephen N Freund. Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '04, pages 256–267, New York, NY, USA, 2004. ACM.

[19] Cormac Flanagan and Stephen N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 121–133, New York, NY, USA, 2009. ACM.

[20] Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. Velodrome: A Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 293–303, New York, NY, USA, 2008. ACM.

[21] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 212–223, Montreal, Quebec, Canada, June 1998.

[22] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[23] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Trans. Comput.*, 36(4):471–482, 1987.

[24] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Dthreads: Efficient Deterministic Multithreading. In *Proceedings of the ACM SIGOPS 23rd symposium on Operating systems principles*, SOSP '11, New York, NY, USA, 2011. ACM.

[25] Li Lu and Michael L. Scott. Toward a Formal Semantic Framework for Deterministic Parallel Programming. In *Proceedings of the 25th International Symposium on Distributed Computing*.

[26] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 37–48, New York, NY, USA, 2006. ACM.

[27] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. LiteRace: Effective Sampling for Lightweight Data-Race Detection. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 134–143, New York, NY, USA, 2009. ACM.

[28] Friedemann Mattern. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1989.

[29] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: Efficient Deterministic Multithreading in Software. In *The International Conference on Architectural Support for Programming Languages and Operating Systems*, Washington, DC, Mar 2009.

[30] Paul Sack, Brian E. Bliss, Zhiqiang Ma, Paul Petersen, and Josep Torrellas. Accurate and Efficient Filtering for the Intel Thread Checker Race Detector. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, ASID '06, pages 34–41, New York, NY, USA, 2006. ACM.

[31] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Trans. Comput. Syst.*, 15:391–411, November 1997.

[32] Min Xu, Rastislav Bodík, and Mark D. Hill. A Serializability Violation Detector for Shared-Memory Server Programs. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 1–14, New York, NY, USA, 2005. ACM.

[33] Qin Zhao, Derek Bruening, and Saman Amarasinghe. Efficient Memory Shadowing for 64-bit Architectures. In *Proceedings of the 2010 international symposium on Memory management*, ISMM '10, pages 93–102, New York, NY, USA, 2010. ACM.

[34] Qin Zhao, Derek Bruening, and Saman Amarasinghe. Umbra: Efficient and Scalable Memory Shadowing. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, pages 22–31, New York, NY, USA, 2010. ACM.