Data Provenance Tracking for Concurrent Programs

Brandon Lucia
Carnegie Mellon University, Department of
Electrical and Computer Engineering
blucia@ece.cmu.edu

Luis Ceze
University of Washington, Department of
Computer Science and Engineering
luisceze@cs.washington.edu

Abstract

We propose *Last Writer Slicing* (LWS), a mechanism for tracking *data provenance information* in multithreaded code in a production setting. Last writer slices dynamically track provenance of values by recording the thread and operation that last wrote each variable. We show that this information complements core dumps and greatly improves debugability. We also propose *communication traps* (CTraps), an extensible framework that uses LWS to interpose on operations that lead to inter-thread communication. We show how to use CTraps to implement multiple dynamic analysis tools for mulithreaded programs. Our evaluation on server programs and PARSEC benchmarks shows that LWS has low run time overheads (0–15%) for many applications,including memcached, LevelDB, MySQL, and Apache. Our debugging case studies of real, buggy programs shows that LWS are indeed useful for debugging.

1. Introduction

Multi-threaded programming is challenging. In contrast to simple sequential reasoning, multi-threaded programs require complex reasoning about many threads and their interactions. Threads primarily interact by reading and writing shared memory locations and different threads' reads and writes interleave arbitrarily and nondeterministically. Different interleavings can lead to different behaviors, some of which may be undesirable, like crashes or data corruption. Such undesirable interleavings often evade testing and cause production failures because complete concurrency testing is infeasible.

The problem that this work addresses is that debugging support for understanding thread interactions is inadequate. Today, programmers use a debugger to examine memory state during a debugging execution or by loading a core dump after a failure. Examining memory reveals *what* the state of the program is (*e.g.*, "is the pointer null?"). Unfortunately, the real question programmers must answer is *why* the state of the program is what it is (*e.g.*, "what thread and instruction made the pointer null?"). Understanding why a program entered an undesirable state (*e.g.*, crashed) is the key to fixing buggy code to prevent that state. Aiding developers in this understanding for failed production executions is especially important because hard-to-find bugs may manifest in production only.

Our approach is to provide the programmer with data *provenance* information that can be examined alongside the program's memory state (*i.e.*, core dump). Provenance information describes why a memory location contains its value. In this work we develop a new provenance tracking mechanism called *Last Writer Slicing* (LWS). A memory location's last writer slice is a record containing the program point and the identifier of the thread that last wrote data to that memory location. As the program executes in production, we collect last writer slices for each potentially shared memory location, recording the provenance of the data in each location. Last writer slices are saved with a core dump and are available in the debugger.

We have three goals for LWS: (1) To collect general provenance information, not limited to certain values or locations only. (2) To provide provenance information focused on how threads interact to aid in understanding concurrency bugs. (3) To engineer our system to have overheads low enough for use in production, aiming for a 10 – 50% maximum overhead.

Last writer slices are primarily useful for concurrency debugging, but provenance information is the foundation of many other concurrency analyses. Several tools from prior work have used *ad hoc* last-writer information as part of more heavy-weight analyses in domains like bug detection (*e.g.*, [11, 16]), automatic debugging (*e.g.*, [13, 19, 32]), profiling (*e.g.*, [7, 26]), and program understanding [29].

Motivated by these *ad hoc* efforts, we develop a dynamic analysis framework called *Communication Traps* or *CTraps*. CTraps uses the provenance information provided by last writer slices to identify memory accesses that lead to *communication* between the threads. Communication occurs when a thread accesses memory last written by a different thread and CTraps delivers a software *trap* to a thread when it communicates with another thread, giving it information about how communication occurred. Programmer-defined *CTraps Applications* can implement arbitrary analysis using that information. By default, CTraps has low overheads, appropriate for production in many cases. As our evaluation

2015 IEEE/ACM International Symposium on Code Generation and Optimization
978-1-4799-8161-8/15/$31.00 ©2015 IEEE

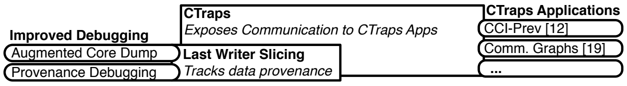


Figure 1: System Overview. Last writer slicing tracks provenance, which augments the core dump and enables provenance debugging. CTraps uses provenance to convey communication to CTraps applications, which implement concurrency analyses, like communication graphs [19] or CBI [13].

shows, the overhead of CTraps applications scales with the complexity of their analysis. Figure 1 shows the relationship between last writer slices, CTraps, and CTraps applications.

Contributions of This Work To summarize, this work makes several contributions:

Last writer slicing (*LWS*): a provenance tracking mechanism useful for concurrency debugging. We present an LWS implementation efficient enough for production use.

Communication traps (CTraps): an extensible framework for implementing concurrency analyses, built on LWS.

Efficacy evaluation: We show that LWS is useful for debugging with a study of several real concurrency bugs. We show CTraps is useful for building concurrency analyses by implementing two analyses from prior work [13, 19, 23].

Performance evaluation: We show that in many cases our designs have overheads low enough for use in deployment (0–15%) with a study of a diverse set of real programs (*e.g.*, MySQL, memcached and the PARSEC benchmarks [3]).

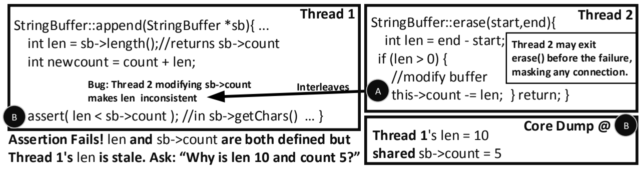
2. Background

This section motivates the use of LWS in deployed systems by showing how provenance is useful for debugging an example program. We then provide context for CTraps by discussing other concurrency analyses that use provenance.

2.1 Debugging and Data Provenance

Prior work shows that data provenance information is directly useful during program debugging. The WhyLine Debugger [14, 15] showed that when programmers debug, they benefit from the ability to ask provenance ("why") questions of their debugger. Bad Value Origin Tracking [4] is a limited form of provenance analysis that tracks intructions that stored certain unusable values. The authors of Bad Value Origin Tracking showed that even such limited provenance information helps understand some bug root causes. Like these prior efforts, we focus on using provenance analysis for *debugging* the root cause of a failure, rather than detecting unknown bugs.

Figure 2 shows how provenance helps debug code from the JDK-1.4 stringbuffer library, adapted to C++ in prior work [31]. The stringbuffer object pointed to by `sb` in Thread 1's code contains a buffer and a variable called `count` that stores its length. The `append()` code should atomically read `sb->count` and update the string's buffer (the full update code is not pictured). The failure occurs when the `append()` code in Thread 1 is interleaved with the `erase()` code in Thread 2 as indicated in the figure. The `erase()` code updates the buffer's `count`, making the version that Thread 1 read into `len` stale. Thread 1's assertion fails at **B** using the stale `len` value, causing a crash.


Thread 1
Thread 2
Bug: Thread 2 modifying sb->count makes len inconsistent
Interleaves
Thread 2 may exit erase() before the failure, masking any connection.
A
B
Core Dump @ B

```
StringBuffer::append(StringBuffer *sb){ ...
   int len = sb->length();//returns sb->count
   int newcount = count + len;
 assert( len < sb->count ); //in sb->getChars()  ... }
```

```
StringBuffer::erase(start,end){
   int len = end - start;
  if (len > 0) {
    //modify buffer
    this->count -= len;  } return; }
```

Assertion Fails! len **and** sb->count **are both defined but Thread 1's** len **is stale. Ask: "Why is len 10 and count 5?"**

Thread 1's len = 10
shared sb->count = 5

Figure 2: Atomicity violation in string buffer code. The core dump does not reveal the connection between `append()` and `erase()`, which is crucial to understand the bug.

When the failure occurs, Thread 1 is at the assertion in `append()`. However, Thread 2 may have completed its erase and moved on to some unrelated code. Thread 2's call stack may lack evidence of its involvement in the failure, despite the fact that its interleaving with the `append()` code is the root cause. The core dump shows that length and buffer are inconsistent, but does not show *why* they are inconsistent.

Understanding the connection between `append()` and `erase()` is the key to fixing this bug. The provenance of the value stored in `sb->count` at **B** was the write at **A** in `erase()`. Provenance information shows the programmer that key connection. Prior provenance work on origin tracking for unusable values [4] does not help here. The involved values are inconsistent, but they are all *usable* so their provenance would not be tracked.

Another challenge posed by this bug is that it manifests as a failure infrequently and under only real-world conditions. Prior slicing and provenance debugging techniques [15, 24, 27], have overheads too high to use in production. Used offline, these techniques may not observe the failure in a reasonable amount of time and triggering the bug may require unavailable real-world inputs making offline analysis impossible. A better strategy is to continuously collect that information in deployed systems and package it with core dumps sent to programmers with bug reports. In this work, we propose a LWS design that espouses this *in situ* data collection strategy and does so with overheads appropriate for production systems.

2.2 Communication Tracking in Concurrency Analyses

Many concurrency analyses monitor inter-thread communication. The domains of these analyses include debugging [9, 17, 19, 22, 23, 25, 32], software engineering [29, 30], anomaly detection [8, 11, 13, 16], bug and failure avoidance [12, 18] and concurrent performance profiling [2, 7].

Despite the potential of a general communication tracking mechanism, prior work has largely relied on *ad hoc* communication analyses. For example, DefUse [23] tracks communication via read-after-write (RAW) dependences, but not write-after-write (WAW) dependences. Recon [19] tracks some inter-thread WAW and RAW dependences. These analyses could both be built with a general communication tracking mechanism. Unfortunately, their single-purpose implementations track slightly different information, making them incompatible and difficult to compose.

These analyses vary in their purpose and environments, but a cross-cutting theme is the advantage reaped from their use in a deployment environment. Debugging tools, profilers, and anomaly detectors benefit from seeing diverse, real-world behavior. Dynamic failure avoidance techniques must work in deployed systems to be effective. Deployment environments demand low overheads and all of these analyses benefit from a general, high-performance communication tracking framework. Unfortunately, many of these techniques are built using heavy-weight infrastructure, like binary instrumentation [20], with overheads too high for production.

There is a need for a general communication tracking framework that has performance overheads that are low enough to use in production systems. As we describe in Section 4, we aim to satisfy that need in this work.

3. Last Writer Slices

A last writer slice is a dynamic property of a memory location at a point in an execution. A memory location's last writer slice records the program point and thread that last wrote data to that memory location. We gather last writer slices at runtime by maintaining a *last writer table* (LWT) that maps from a memory location's address to its last writer slice. Just before a thread writes to a memory location, the thread first finds the memory location's entry in the LWT and updates that entry with its thread identifier and its current point in the program. After updating the LWT, the thread performs the original memory access.

Figure 3 shows the basic operation of the LWT. The right side of the figure shows a program execution with three threads. On the left is the LWT at the point of each memory access. There are three key things to notice. First, write operations update the LWT entry of the memory location they access. Second, there is no action necessary for read operations. Making read operations cheap is key to making the overhead of LWS low enough for production use. Third, if the program stops at a point like the `Rd Y` illustrated in Thread 1, the LWT holds provenance information for the involved values (*i.e.*, Thread 3 last wrote `Y` at the code point shown). That information helps understand *why* `Y` holds its value.

3.1 Debugging with Last Writer Slices

At each point in an execution, a memory location's LWT entry represents the provenance of the data stored in that memory location. Last writer slicing performs *no action on a read operation*. Instead, a programmer can examine an LWT entry at any arbitrary point in an execution and see a value's provenance (*e.g.*, at a breakpoint or after a failure). Including the LWT with core dumps means provenance is available when post-mortem debugging production failures.

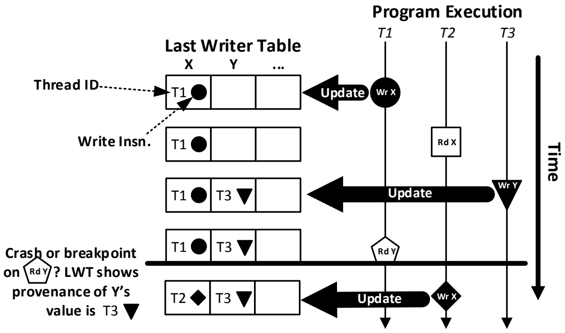
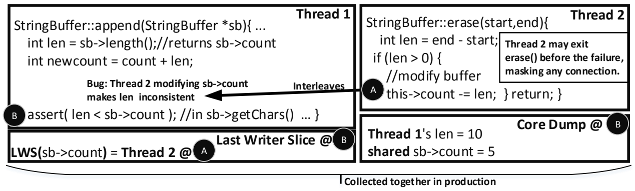


Figure 3: Basic operation of the last writer table. Different shapes signify different distinct memory operations.

Figure 4 illustrates how the information in the LWT helps debug the JDK stringbuffer bug introduced in Figure 2. Recall that the failure occurs because Thread 2's execution of `erase()` violates the atomicity of Thread 1's execution of `append()`. The LWT entry for `sb->count` at **B** shows the programmer that its last writer was Thread 2 at **A**. Thread 2 may have run ahead to an arbitrary point in its execution. Nothing in the core dump implicates `erase()` in causing this bug. The last writer slice reveals the interference between `append()` and `erase()` via `sb->count`, helping the programmer understand the bug's cause.



```
StringBuffer::append(StringBuffer *sb){ ...
   int len = sb->length();//returns sb->count
   int newcount = count + len;
assert( len < sb->count ); //in sb->getChars()  ... }
```

```
StringBuffer::erase(start,end){
   int len = end - start;
  if (len > 0) {
     //modify buffer
     this->count -= len;  } return; }
```

Thread 1
Thread 2
Bug: Thread 2 modifying sb->count makes len inconsistent
Interleaves
Thread 2 may exit erase() before the failure, masking any connection.
A
B
Last Writer Slice @ B
LWS(sb->count) = Thread 2 @ A
Core Dump @ B
Thread 1's len = 10
shared sb->count = 5
Collected together in production

Figure 4: Using last writer slices to debug an ordering violation in the JDK 1.4 StringBuffer library. The last writer slice reveals the buggy interaction between `erase()` and `append()`.

Another important trait of last writer slices illustrated by Figure 4 is that when the assertion fails at **B**, stopping the execution, the LWT is saved and can be sent to the programmer with the core dump. Capturing good debugging information is especially important for concurrency bugs that manifest very rarely or in the presence of only peculiar real-world inputs. A primary goal of this work is to show that we can collect last writer slices with very low overheads so they can be included pervasively in bug reports.

3.2 Runtime & Compiler Support

Figure 5 shows an overview of our system support for last writer slicing. The programmer writes their program as usual. Our compiler instruments write operations in the program with calls to our runtime that update the LWT. The LWT is maintained in the runtime and saved with the core dump.

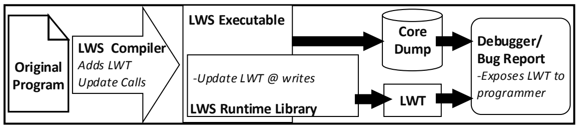


Figure 5: Overview of system support for LWS. Our compiler and runtime library ensure the LWT remains updated so it can be used during debugging.

The LWT implemented in the runtime library is a *shadow memory* that holds one entry for each potentially shared memory address accessed by the program. Each LWT entry records the thread and program point that last wrote to the entry's corresponding memory location. When a thread performs a write to a memory location, our runtime library's LWT update function updates the LWT by recording the thread's identifier and the program point of the write operation in the memory location's LWT entry. A program point may be an instruction address or a call stack. We use instructions by default, but we also experimented with 2-address call stacks.

Collecting last writer slices requires instrumentation on each write operation. We built an instrumenting compiler pass that inserts a call to the LWT update function in our runtime library just before each write that may access data last written by another thread . LWT update calls are passed three pieces of information: the address of the memory word involved in the access, the thread identifier of the accessing thread, and the accessing program point. We use escape analysis to identify accesses to unshared data and we do not instrument them.

3.3 LWT Formalism and Correctness

We developed a formalism of our LWT-based last writer slices support, but we omit it due to space constraints. We used our formalism to show that the LWT correctly records last writer slices. There are two important caveats in our formal discussion. The first caveat is that the LWT implementation introduces no synchronization. We proved that the LWT is always correct in data-race-free programs because LWT accesses are synchronized by program synchronization. However, if an execution has a data-race, program memory operations and LWT accesses are no longer atomic. If such a data-race leads to an unserializable interleaving of LWT accesses, the memory location's LWT entry may reflect the wrong last writer. The second caveat is that the LWT assumes that a memory location accessed at one granularity (*e.g.*, byte-aligned) is never accessed at a different granularity (*e.g.*, as part of a word-aligned access). Assuming accesses do not overlap ensures there is always only one LWT entry for each memory location.

4. Communication Traps

CTraps is a framework for analyzing and interposing on inter-thread communication. CTraps uses LWS (§ 3) to track inter-thread communication. CTraps exposes communication events as *traps* during an execution that *trap handlers* can handle to perform analysis and interposition.

4.1 CTraps Design

We now discuss CTraps in more detail by describing the design of our CTraps system support. Figure 6 shows an overview of our system support. The programmer writes their program as usual and compiles it using the CTraps compiler. The CTraps compiler includes the compiler support for collecting last writer slices. The CTraps compiler also inserts calls to the CTraps runtime at points where communication may occur. The resulting compiled executable links to the CTraps runtime that collects last writer slices and monitors communication. The runtime loads and manages CTraps handlers when the execution starts via a plugin interface. CTraps handlers, written independently of the original program implement CTraps applications. Traps are delivered to handlers by the runtime when operations communicate. Together, the CTraps executable and the CTraps handlers are deployed.

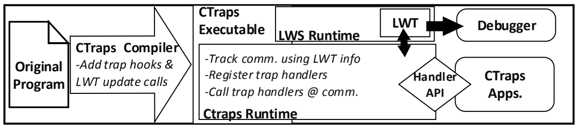


Figure 6: CTraps system support.

4.1.1 Compiler and Runtime Support

CTraps delivers software traps before each read and write operation executed by one thread that accesses memory that was last written by another thread, *i.e.*, communicating memory operations. Traps are simply function calls made before such operations execution. CTraps implements traps using a combination of compiler and runtime library support.

Compiler Support Our compiler inserts a call to the CTraps *trap hook* function immediately before all read and write operations to potentially shared variables. Trap hooks are passed the variable's address, the accessing thread's ID, the type of access, and the program point of the access.

Runtime Support The CTraps runtime implements the trap hook function, which has two purposes. The trap hook function detects communication and delivers traps to CTraps applications when communication occurs. The trap hook function detects communication by looking up the LWT entry for the memory location being accessed. If the thread identifier in the LWT entry is different from the accessing thread, then the trap hook function delivers a trap to the accessing thread, just before the memory access is allowed to proceed.

To deliver a trap, the trap hook function calls *trap handler routines* that are defined in CTraps applications. Trap handler routines are like signal handlers. The CTraps runtime maintains a list of trap handlers that are registered when the program starts via a configuration stored in the environment. Trap handlers may contain arbitrary application-specific code. CTraps defines a trap handling API. Trap handlers are passed the four pieces of information passed to the trap hook, as well as the LWT's record of the program point and thread identifier of the last write to the involved location.

4.2 CTraps Applications

CTraps applications are implemented as shared library plugins that are loaded by our runtime. They must implement the CTraps trap handler API, which requires a trap handler function and permits a constructor, destructor, thread constructor, and thread destructor, which allow initialization and disposal of global and thread-local state. Many useful CTraps applications are possible, several of which were outlined in Section 2.2. To demonstrate that CTraps can implement real analyses with little work, we implemented two applications. The first application is a variant of an analysis from CCI [13]. The second application is a communication graph collector, like DefUse [23], Recon [19], and DMTracker [11].

CCI-Prev Implementation CCI-Prev is a technique from CCI [13] that records a set of code points that access a memory location when the previous access to the same location was by a different thread. We implemented a variant of CCI-Prev using CTraps that records such a set of code points. In our implementation, communcating read operations update the LWT like writes normally do. Under this policy, any operation that accesses a location that was last accessed by a different thread is recorded. The set of recorded accesses is the output of the analysis. Our CTraps implementation of CCI-Prev took about 10 lines of code (on top of our base system).

Communication Graph Implementation Communication graphs are the basis of several prior debugging techniques [17, 19, 23]. We implement communication graph collection using CTraps. When a trap is delivered, our implementation records communication graph edges composed of the code point of the last writer to the location being accessed and the code point of the trapping access. The set of recorded edges is the tool's output. Our implementation took about 50 lines of code.

5. System Implementation

We implemented LWS and CTraps. We built our compiler support as a plugin for GCC (`gcc-4.7`). We used GCC's points-to and escape analysis support to prune instrumentation on accesses to non-escaping memory locations. Our compiler instruments calls to `free` and `delete` as writes to the pointer being deallocated. Our analysis handles all code compilable using this version of GCC except for a few cases: we do not handle accesses that compile to GCC `BITFIELD_REFERENCE` IR types because these are undocumented, we do not handle inlined assembly instructions, and we do not handle C++ exceptions. Note that these limitations of our research prototype are not fundamental and the limited cases are uncommon.

We built our runtime system from scratch. The LWT is a fixed size 2GB array of 64-bit words. Each entry packs a program point and a thread identifier into the 64-bit word. By default, program points are single instruction addresses, but we also support limited (2-address) context-sensitivity by packing the current instruction's address and the nearest return address into a 64-bit LWT entry. When an access to a memory location occurs, we index into the LWT with the lower bits of the location's address. Our prototype implementation uses a lossy resolution policy for hash collisions. We use this policy in our prototype because it is fast and bounds memory overheads at 2GB. Collisions are rare (see Section 6.5), so this simplification is unlikely to be a problem.

Release We released our implementation, free and open-sourced, on the web [1] and published it to the GCC plugin repository. Since its release, we have seen contributions to our codebase from members of the GCC community and downloads from interested developers.

6. Evaluation

We evaluated last writer slices and CTraps along several axes. We show that last writer slices provide information that is useful during debugging in Section 6.1, looking at a set of buggy programs and comparing to prior work. We evaluate our performance overheads and show that last writer slices and CTraps are feasible for use in deployed systems for a majority of workloads we studied. We then characterize our instrumentation and discuss sources of run time overhead. We show that CTraps is useful by implementing two existing applications and analyzing their performance.

6.1 Debugging with Last Writer Slices

We illustrate the debugging benefits of last writer slices by using our system to debug several real-world programs studied previously in the debugging literature. Table 1 shows the bugs we studied, describing the program, bug type, and report information. We followed the procedure described by the author of [31] for installing and triggering each bug . We study various bug types, including both single- and multi-variable bugs and both atomicity and ordering violations.

The table also summarizes our evaluation of LWS's debugging benefits. We mirror the evaluation strategy of bad value origin tracking [4] and evaluate our system using their three evaluation criteria - (1) **effectiveness**, (2) **triviality**, and (3) **usefulness**. Our technique is **effective** if it tracks a

App	Ver. #	Bug #	Type	Eff.?	Triv.?	Usef.?	O.T.?
jdk1.4	1.70	N/A	Atom.	Yes	No	Yes	No
pbzip2	0.9.4	N/A	Order	Yes	No	Yes	Yes
httrack	3.43.9	N/A	Order	Yes	No	Yes	No
transmission	1.42	N/A	Order	Yes	No	Yes	No
mysql	4.0.12	791	Atom.	Yes	No	Yes	No
apache1	2.0.48	21285	Atom.	Yes	No	Yes	Some
apache2	2.2.9	45605	Atom.	Yes	No	Some	No

Table 1: Concurrency debugging with last writer slices. We describe each bug and summarize the debugging benefit of last writer slices using the evaluation criteria from [4].

correct last writer slice for the value or values involved in causing the failure. A failure is **trivial** to debug if its cause is made obvious by examining the core dump alone, *i.e.*, a last writer slice is unnecessary. A last writer slice is **useful** if it is effective and the thread and code point in the slice help understand why the failure occurred or how to fix it. We directly compare to bad value origin tracking [4]. The **O.T.** column in Table 1 shows whether their analysis could help debug each bug, based on the description of their technique.

The results in Table 1 show that for both atomicity and ordering violation bugs, last writer slices are effective. In all cases, we found that the last writer slice revealed the provenance of at least some data involved in the failure, so LWS was effective. We found that none of the failures we looked at were trivially debuggable. This finding is consistent with the difficulty of debugging concurrency bugs – the core dump does not reveal the interaction between threads that led to a failure, only the result of the interaction.

Last writer slices were useful for helping understand how to fix the bug in all but one case and useful at least for understanding the bug in the remaining case, apache2. The main reason last writer slices are helpful is that they reveal the connection between code running in different threads that interacts to cause a failure. Without last writer slices, programmers have no easy way of connecting these otherwise disparate parts of a program. The case studies in Section 6.1.1 show in more detail how LWS is useful for debugging and illustrates its limitations.

Our evaluation shows that bad value origin tracking [4] is helpful in one case where LWS is useful and provides limited help in one other case. For a use-after-delete bug in pbzip2, bad value origin tracking helps. For this bug, when one thread deletes a variable, it makes that variable unusable and bad value origin tracking would track that value. When the failure occurs, the programmer would then know why the value was deleted.

For apache1, bad value origin tracking may help. The bug is an atomicity violation that leads to a double free. Two threads manipulate an object's reference count and deallocate the object when the count hits zero. Zero-checks and deallocation should be atomic, but the code does not ensure they are. If threads interleave their code, multiple threads might see zero reference counts, which is incorrect. When that happens, both threads free the object and the second free causes a crash.

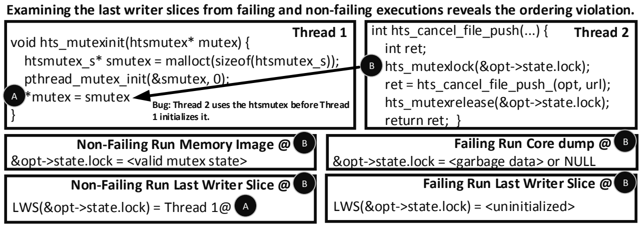
Bad value origin tracking may help in this case. At the first free, the freed pointer becomes unusable and its origin is tracked. At the crash, the bad value's reported origin is the code point of the free. That information may help debug the program by suggesting a double free. However, in apache, many threads perform different tasks and all call the same free code. Bad value origin tracking does not record the thread information. The programmer is left unsure whether the problem is with single-threaded logic or synchronization. In contrast, last writer slices reports the code point and thread, making it clear the bug is a concurrency bug and identifying the involved threads.

6.1.1 Debugging Case Studies

We now provide detailed cases studies showing the debugging role of last writer slices in several of the bugs we examined.

jdk-1.4 The running example in Figures 2 and 4, and the text in Section 3.1 describes the benefit provided by last writer slices in debugging an ordering violation in the JDK-1.4.

httrack Figure 7 shows how last writer slices help understand the root cause of a concurrency bug from HTTrack-3.43.9. The program crashes at point **B** when it calls hts_mutex_lock() with an invalid mutex. The root cause of this bug is that Thread 1 fails to initialize the mutex before Thread 2 uses it. The core dump is of little debugging use, only showing that, indeed, the lock is uninitialized. Even if the programmer compared the core dump from the failing run to the memory image from a correct execution – a helpful debugging strategy – they would find only that in a correct execution, the lock contains valid mutex state.


Examining the last writer slices from failing and non-failing executions reveals the ordering violation.
Thread 1
Thread 2
A
B
Bug: Thread 2 uses the htsmutex before Thread 1 initializes it.
Non-Failing Run Memory Image @ B
Failing Run Core dump @ B
Non-Failing Run Last Writer Slice @ B
Failing Run Last Writer Slice @ B

```
void hts_mutexinit(htsmutex* mutex) {
   htsmutex_s* smutex = malloct(sizeof(htsmutex_s));
   pthread_mutex_init(&smutex, 0);
   *mutex = smutex
}
```

```
int hts_cancel_file_push(...) {
   int ret;
   hts_mutexlock(&opt->state.lock);
   ret = hts_cancel_file_push_(opt, url);
   hts_mutexrelease(&opt->state.lock);
   return ret;  }
```

&opt->state.lock = <valid mutex state>

&opt->state.lock = <garbage data> or NULL

LWS(&opt->state.lock) = Thread 1@ A

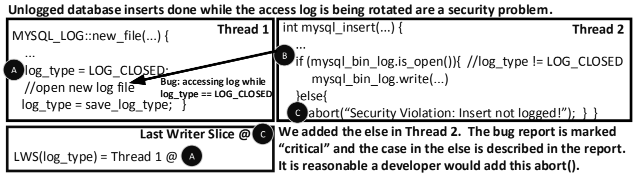
LWS(&opt->state.lock) = <uninitialized>

Figure 7: Using last writer slices to debug an ordering violation in HTTrack. The provenance information in the mutex's last writer slice helps understand the bug's root cause.

Looking at last writer slices make the root cause clear: in the failing execution, the last writer slice indicates the mutex is uninitialized; in the non-failing execution, the last writer slice indicates the mutex was initialized by Thread 1 at point **A**. While the core dump provides no information about the connection between point **A** and point **B**, the last writer slice reveals that the programmer should ensure **B** follows **A** to prevent the failure. Note that even augmenting the core dump with bad value origin tracking [4] does not help with this bug. In the failing execution, origin tracking shows that the unusable lock is uninitialized, revealing no more than the core dump. In the non-failing execution, origin tracking does not monitor the *usable* lock value, adding nothing. By contrast, comparing the last writer slices for the failing and non-failing executions reveals how to fix this bug.

transmission We used last writer slices to debug a use-before-initialization bug in transmission-1.42. We omit a full case study due to space constraints, but we mention a few notable properties of this case. First, like `httrack`, we debug `transmission` by comparing the last writer slice from failing and non-failing executions. Second, last writer slices reveal all the code and data involved in this bug, making the bug and its fix clear. Third, bad value origin tracking does not work in this case because in failing executions, the bad value has no origin and in non-failing executions, the involved variable holds only usable values.

mysql We used last writer slices to debug an atomicity violation in MySQL-4.0.12 that is a critical security vulnerability. Figure 8 shows the buggy code. Thread 1 rotates the database's log file. Log rotation should be atomic, but the lack of synchronization allows Thread 2 to run its insert during log rotation. In that case, the insert query is not logged, manifesting the failure. We added an `abort` statement to the non-logging case because this behavior is a security vulnerability that the bug report describes as "critical". Adding the `abort` is reasonable, as the bug report describes the `else` block as an error case. Running in a debugger, a programmer might equivalently use a breakpoint, rather than adding an `abort`.



Unlogged database inserts done while the access log is being rotated are a security problem.

```
MYSQL_LOG::new_file(...) {
   ...
A log_type = LOG_CLOSED;
   //open new log file
  log_type = save_log_type;  }
```

```
int mysql_insert(...) {
   ...
B  if (mysql_bin_log.is_open()){  //log_type != LOG_CLOSED
       mysql_bin_log.write(...)
   }else{
C  abort("Security Violation: Insert not logged!");  }  }
```

```
LWS(log_type) = Thread 1 @ A
```

We added the else in Thread 2. The bug report is marked "critical" and the case in the else is described in the report. It is reasonable a developer would add this abort().

Figure 8: Using last writer slices to debug an atomicity violation in MySQL 4.0.12. The last writer slice reveals the code that closes the log before a database insertion that goes unlogged, which is a security problem.

We triggered the bug by running an insert query during a log rotation and execution stopped at **C**. The core dump showed the log was marked closed, which is the failure state (*e.g.*, `log_type` was `LOG_CLOSED`). However, the core dump did not reveal *why* the log was marked closed. The last writer slice at **C** for `log_type` shows that it was written last by Thread 1 at **A**, in the log rotation code, which should be atomic. Thus, the last writer slice shows the programmer the need for log rotation to be atomic with respect to the insert code to prevent this failure. Bad value origin tracking does not help debug this bug because `log_type` only ever contains usable values so it would not be tracked.

apache2 We used last writer slices to debug an atomicity violation in Apache-2.2.9 and we found that last writer slices help understand the bug, but not fully how to fix it. The bug involves a group of worker threads that handle web requests and a network listener thread that passes requests to the workers. The server tracks the number of idle workers. When a worker becomes idle, it increments the number of idlers and when the listener passes work to a worker, it decrements that number. The counter's increments and decrements are synchronized using condition variables. If there are no idlers, the listener waits for a signal. When a worker becomes idle when no others are, it delivers a signal to the listener.

The problem is that the listener does not check the condition governed by the condition variable after being signaled on that condition variable. As a result, the listener can incorrectly decrement the idler count twice, underflowing the unsigned value, which then causes an assertion to fail. When the failure occurs, the core dump shows the idler count underflowed, revealing a primary symptom of the bug, but not providing much help. The last writer slice for the idler count reports that the listener thread last wrote the idler count at its decrement operation. That information shows the programmer *why* the underflow occurred, which is helpful for understanding the failure. However, we conclude that last writer slices are only somewhat useful in this case. Solving this complex bug requires understanding not just the connection between the underflow and the listener's decrement, but also the synchronization protocol that coordinates that decrement and other accesses to the variable.

6.2 Performance Evaluation: Setup and Benchmarks

We conducted our evaluation on a machine running Linux 2.6.27-7, with a 2.27GHz 8-core Xeon E5520 processor with 2-way SMT and 10GB of memory. We evaluated our system using the PARSEC benchmarks [3] and several real-world applications.

6.2.1 PARSEC

We ran PARSEC programs with their `native` input set, and with each program's 8 thread configuration. We have omitted three of the PARSEC benchmarks because our compiler pass does not handle C++ exceptions.

6.2.2 Servers

MySQL-5.1.65 MySQL is an industrial-strength database server. To benchmark MySQL, we used `SysBench OLTP` running 8 threads, measuring MySQL's performance as the throughput reported by `SysBench`.

Apache-2.4.3 Apache-httpd is a popular web server. To benchmark Apache, we used `ApacheBench` to request a static `html` page 1,000,000 times using 8 threads, measuring Apache's performance to be the throughput reported by `ApacheBench`.

LevelDB-1.5.0 LevelDB is a high-performance key-value store written by Google developers. To benchmark LevelDB, we used the included `db_bench` utility, running its "Read while Writing" test with 8 threads. We measure LevelDB's performance to be the throughput reported by `db_bench`.

Memcached-1.4.4 Memcached is an in-memory key-value store frequently used as a cache for web services. We were unable to find a standard benchmark for Memcached. Instead, we wrote a C program that uses libmemcached-0.49 to issue a mixture of 10% load requests and 90% store requests for a single key simultaneously from 8 threads. We measured Memcached's performance to be the total time to complete 10,000 requests in each thread.

6.3 Performance Evaluation

The main performance result of our work is that our designs impose performance overheads that are low enough for use in production for many of our workloads. Figure 9 shows the slowdown suffered by our benchmarks due to LWS and CTraps running with "no-op" trap hooks (*i.e.*, no handlers), relative to the natively executing baseline. In these experiments, we consider a run time overhead of 10% or less to be ideal for production use and a run time overhead of 50% as a reasonable upper bound on acceptable overhead for production use.

Last Writer Slices The data show that LWS has very low overheads with a geometric mean of less than 10% across our server programs and less than 50% across PARSEC. Such low overheads are likely to be acceptable in production. In many cases (Apache, MySQL, `dedup`, `canneal`), overheads are negligible. In all but two cases (`vips` & `swaptions`), the overhead of collecting last writer slices is less than 100%.

CTraps CTraps imposes a geometric mean overhead of 14% for server applications and 110% for PARSEC applications. In 7 out of 13 of our tests, the overhead of CTraps is less than 50%. These seven low overhead benchmarks include all of our server programs, as well as `blackscholes`, `dedup` and `canneal` from PARSEC. The overhead for these applications is likely to be tolerable in production. `vips` and `swaptions` saw the highest overheads – around 400%. We discuss sources of high overheads in Sections 6.3.1 and 6.3.2.

By comparing CTraps and LWS, we see four applications (`dedup`, `streamcluster`, `fluidanimate`, `ferret`) that have CTraps overhead that is probably too high for production use (averaging 153%). By comparison, those applications have a LWS overhead that is likely acceptable for production use (averaging 38%). These programs perform relatively more reads than other applications, thus experiencing more overhead due to CTraps's read instrumentation.

These high-level results support our claim that our overheads are low enough for deployed systems for many applications, especially for servers.

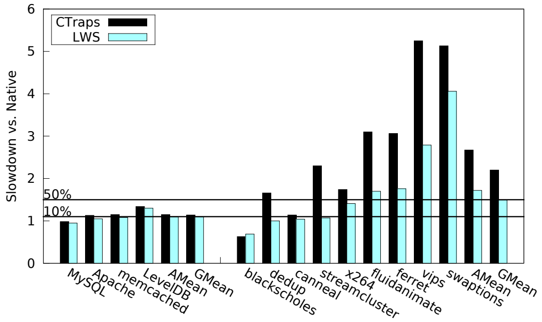


Figure 9: Runtime overhead of CTraps and LWS.

6.3.1 Overheads Due to Conservative Escape Analysis

`swaptions` had the highest run time overhead in our tests. We investigated the cause and determined it is largely due to shortcomings in our escape analysis. We located the inner-loop function `HJM_SimPath_Forward_Blocking`. The function uses two local matrices that are allocated through an external call. Our escape analysis cannot determine that the matrices are only used locally in that function, so it conservatively assumes the matrices escape. We manually inlined the matrix allocation calls (and a matching deallocation calls), and escape analysis eliminated their instrumentation. The program's overhead dropped from about 420% overhead for CTraps to about 300% overhead; the change for LWS was from 300% to about 130% overhead. We also looked at an output matrix passed into the inner-loop function. The matrix is allocated and deallocated in the inner-loop's caller, but is never shared. Eliminating instrumentation on accesses to this matrix yielded an overhead of 46% for CTraps and just 30% for LWS.

These manual improvements to the escape analysis reduced the run time overheads for these workloads from being unacceptable for any production environment to being reasonable for use in production systems. Better escape analysis would find these performance gains automatically.

6.3.2 PARSEC vs. Servers

We discern several reasons that server programs better tolerated the overheads of LWS and CTraps in our experiments.

Independent Parallelism In most cases, the servers we evaluated have abundant, completely independent parallel work. Server programs use a pool of threads to handle requests. Each thread incurs some instrumentation latency, but that latency is hidden by other threads making progress on independent requests. This analysis also applies to the PARSEC applications that had performance similar to the servers. For instance, `blackscholes` uses largely independent threads to compute on different regions of a matrix. In addition, threads performing independent computation (*i.e.*, non-communicating computation) are likely to operate mostly on non-escaping, local data. Local data are not instrumented by our compiler, sparing the overhead. We analyzed the amount of unshared data in each application and found that for our server workloads, 21–66% of accesses were to provably local data. Far fewer accesses are provably unshared for PARSEC: just 8–30%. The difference supports the fact that the servers have more independent work (*i.e.*, more local accesses) than PARSEC.

Interference with Optimization The PARSEC programs are heavily hand-optimized. Several use hand-coded assembly and cache-aware algorithms. The instrumentation that accesses the LWT may disrupt such delicately optimized behavior, leading to higher overheads. Such carefully optimized code is often written by an expert. As with other expert-written code (*e.g.*, lock-free data-structures), this code may be machine verified or simply not require as much debugging support as code written by an average programmer. Depending on the maturity and level of verification of code, a programmer can disable LWS and CTraps to preserve performance.

6.4 Context-Sensitivity

We briefly evaluated the cost of context-sensitivity using CTraps with no-op trap handlers. With 2-address context, the average overhead for server programs was 18% and for PARSEC was 145%. These data show that context-sensitive analysis is possible in CTraps with practically low overheads.

6.5 Memory Overheads

As we describe in Section 5 our prototype LWT is a 2GB hashtable of 8 byte entries. We use a lossy hash collision policy: all updates always overwrite existing entries. This strategy fixes memory overheads at 2GB, but frequent collisions may lead to imprecision. We instrumented our runtime to count collisions and found that for PARSEC applications the geometric mean rate of collisions was very low - 89 per 10,000 memory accesses. This summary result shows that our prototype implementation is reasonable, especially for debugging in production, when low memory use is key. Note that collisions did not introduce noticable imprecision in our debugging experiments. When precision is more important than memory overhead, (*e.g.*, for sound, offline analysis), a lossless hashtable is a better option.

6.6 Evaluating CTraps Applications

We evaluated our CTraps applications and our results show that they have modest overheads that scale with their analysis complexity.

Figure 10 shows the performance overheads of CTraps applications as the slowdown relative to native execution. The overheads vary across applications and the average run time overheads are higher than our no-op CTraps application (Section 6.3). In six cases (the four servers, `blackscholes`, and `dedup`), the overhead for CCI-Prev is tolerable for deployment use (around 50%). This result supports our claim that CTraps helps build useful analyses that have overheads low enough for deployment for an important class of applications.

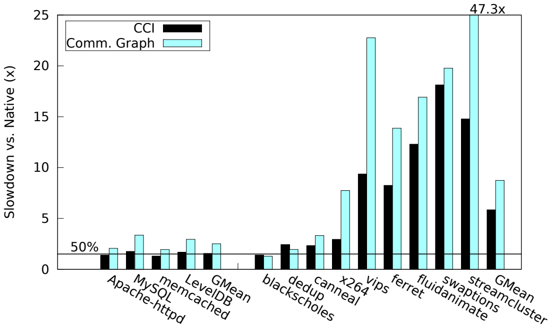


Figure 10: Performance overheads of CCI-Prev and Graph Collection implemented with CTraps.

Overheads for communication graph collection are generally higher and probably too high for production use in most instances. The higher overheads are not a negative result; instead, they illustrate the "complexity proportional" nature of CTraps' overheads. Our communication graph collector does more costly data structure manipulations on each trap, so its overheads are higher. CTraps provides a low-overhead foundation and applications scale their overhead as necessary.

The PARSEC benchmarks have higher overhead than the servers for CCI-Prev and graph collection. Graph collection overheads for `vips` and `streamcluster` are the worst cases we saw (23x and 47x, respectively). We described some reasons for the higher overheads in these tests in Section 6.3.2. The overheads discussed there are further exacerbated by the increased cost of reads in these applications. These overheads are too high for use in deployment; however, we note that PARSEC applications are batch focused and compute-intensive; we expect the long-running, event-driven nature of server applications makes them a more important target than PARSEC for these types of analyses in production.

7. Related Work

Debugging with Data Provenance Information Prior work has shown that some limited forms of provenance help with debugging [4, 15, 27].

Bad Value Origin Tracking [4] tracked code that wrote unusable values into variables [4]. That work showed that revealing the connection between an unusable value's origin and a failure it causes is helpful during debugging. Bad value origin tracking is the closest prior work to this work [4] and we directly compare to it in our evaluation. At a high level, bad value origin tracking is like our work because they also track a form of provenance for debugging.

Bad value origin tracking relates to our work in several other ways. First, their technique tracks the provenance of *unusable* values only – a fraction of all values in an execution. Our work, instead, tracks provenance for any value stored in a potentially shared memory location, which includes a broader set of provenance and targets our technique to concurrency debugging. Second, for Java programs, bad value origin tracking has runtime overheads acceptable for production, which was a goal for our system. Bad value origin tracking "piggybacks" provenance information on storage allocated by the Java runtime for bad values. Piggybacking is a key to their low overheads. For C/C++, the overhead of bad value origin tracking is much higher. In C/C++, they cannot use piggybacking and, instead, use heavy-weight binary instrumentation with overheads unacceptable for production. Our technique has overheads low enough for production for C/C++ programs, especially for server applications. Third, their technique does not explicity record thread information with tracked origins. Our work focuses on concurrency and we include thread information. As we show in Section 6.1 thread information is important for some bugs.

Other work showed that programmers debug more effectively when they can ask "why" questions about values during debugging [15, 27]. Like our work, the WhyLine debugger [14, 15] allows programmers to ask questions about why variables hold particular values during debugging. WhyLine answers by presenting data- and control-flows that influenced those variables. WhyLine differs from our work in that it uses slicing on recorded executions to answer provenance questions, it has high run time overheads, and does not focus on concurrency.

Program Slicing Program slicing techniques identify useful subsequences of a program's instructions. Slicing can work by analyzing program code statically or analyzing execution traces dynamically. Many slicing techniques discussed in the a recent survey [27] monitor control and data dependences to identify which parts of a program might be relevant to a task.

Thin Slicing [24], is a slicing technique that is related to our work. Thin Slicing yields the set of statements that computed on or wrote values that influenced a "seed" variable's value. Thin Slicing is similar to our work in that slices provide provenance and it excludes most control-flow information. Thin Slicing is different in that it has overheads too high for production and focused on minimizing the size of each slice compared to prior slicing techniques.

Communication and Sharing Analyses Several techniques have been proposed analyze communication in concurrent programs and are especially related to CTraps.

A closely related recent effort on dynamic dependence analysis is Octet [5], which exposes inter-thread communication soundly to concurrency analyses. Octet reports an average run time overhead of 26%, which is comparable to the overhead of LWS and CTraps.

To acheive these low performance overheads, Octet makes fundamental assumptions about its target system that LWS and CTraps do not make. To track communication, a thread in Octet must stop at "safe points" in an execution. At a safe point, the thread executes extra code to check a message queue that reports communication with other threads. Octet assumes safe points at loop back edges, method entries, and blocking function calls. This assumption is reasonable for Java, as many Java VMs use safe points to implement GC. Octet uses existing safe points and their performance baseline assumes a VM with existing instrumentation at such safe points. In contrast, C/C++ programs do not have a VM or existing safe points. Checking a queue on every function call and method entry is likely to impose a higher performance penalty than Octet saw in Java. LWS keeps overheads practical by relying on program synchronization to keep dependence information consistent — unlike Octet which is always sound, Data-races may leave LWS data inconsistent. However, LWS need not pay the cost of safe points and message queue checks.

Another important difference between Octet and this work is that this work illustrates the benefit of directly using LWS's provenance information for debugging. Octet does not explore provenance debugging.

CCI [13], Bugaboo [17], LBA [28], Recon [19], and DefUse [23] all explicitly track some thread interactions. Unlike our work, several of these techniques were not designed to run in production systems [19, 23]. A distinguishing characteristic of our work is that it is effective and fast enough for production without the need to aggregate information from multiple different sampled executions, like CCI [13] does. Additionally, our system does not require invasive hardware changes like Bugaboo [17] or LBA [28].

Extensible Program Instrumentation There are many systems for building analyses using program instrumentation. Binary instrumentation [6, 10, 20, 21] is general and tracking provenance and communciation is possible in such systems.

These systems are similar to CTraps in that they enable dynamic analyses. They are different from our work because they rely on dynamic binary translation, yielding overheads often too high for deployment. We trade some generality for performance. CTraps is effective mainly in the restricted domain of concurrency analyses, but has overheads low enough for production. Another difference is that our work handles some of the delicate, performance sensitive code required concurrency analyses, like last writer tracking. In general instrumentation systems, programmers write that code from scratch at their own risk.

8. Conclusions

In this work we proposed LWS, the first technique to collect data provenance information that is not limited to certain values, is targeted to concurrent programs, and has overheads low enough for production use. We showed that during debugging, the provenance information provided by last writer slices reveals crucial connections between different threads' otherwise disparate regions of code. Understanding how code in different threads interacts is essential to concurrency debugging and this work aids that understanding. Using last writer slices, we then built CTraps, an extensible framework for implementing concurrent program analyses that interpose on inter-thread communication. Trap handlers implement such analyses in just a few lines of code.

Acknowledgements

We would like to thank our anonymous reviewers for their invaluable comments. Our thanks also to Kathryn McKinley and the members of the Sampa group at the University of Washington. This work was supported in part by C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA. It was also supported by a gift from Google.

References

[1] Anonymous for submission. GCC Plugins Directory (links to our release). `http://gcc.gnu.org/wiki/plugins`, 2013.

[2] A. Bhattacharjee and M. Martonosi. Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors. In *ISCA*, 2009.

[3] C. Bienia, S. Kumar, J. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. Technical report, Princeton University, January 2008.

[4] M. D. Bond et al. Tracking bad apples: reporting the origin of null and undefined value errors. 2007.

[5] M. D. Bond, M. Kulkarni, M. Cao, M. Zhang, M. F. Salmi, S. Biswas, A. Sengupta, and J. Huang. Octet: Capturing and controlling cross-thread dependences efficiently. 2013.

[6] D. L. Bruening. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Cambridge, MA, USA, 2004.

[7] M. Curtis-Maury, A. Shah, F. Blagojevic, D. S. Nikolopoulos, B. R. de Supinski, and M. Schulz. Prediction models for multi-dimensional power-performance optimization on many cores. In *PACT '08*, 2008.

[8] M. D. Ernst. *Dynamically Discovering Likely Program Invariants*. Ph.D., University of Washington Department of Computer Science and Engineering, Seattle, Washington, Aug. 2000.

[9] C. Flanagan and S. N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *PLDI*, 2009.

[10] C. Flanagan and S. N. Freund. The roadrunner dynamic analysis framework for concurrent programs. In *PASTE*, PASTE '10, 2010.

[11] Q. Gao, F. Qin, and D. K. Panda. Dmtracker: finding bugs in large-scale parallel programs by detecting anomaly in data movements. In *2007 Conference on Supercomputing*, 2007.

[12] G. Jin et al. Automated concurrency-bug fixing. In *OSDI*, 2012.

[13] G. Jin, A. Thakur, B. Liblit, and S. Lu. Instrumentation and sampling strategies for cooperative concurrency bug isolation. OOPSLA '10, 2010.

[14] A. J. Ko and B. A. Myers. Designing the whyline: a debugging interface for asking questions about program behavior. In *CHI '04*, 2004.

[15] A. J. Ko and B. A. Myers. Debugging reinvented: asking and answering why and why not questions about program behavior. In *ICSE '08*, 2008.

[16] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *ASPLOS*, 2006.

[17] B. Lucia and L. Ceze. Finding Concurrency Bugs with Context-Aware Communication Graphs. In *MICRO*, 2009.

[18] B. Lucia and L. Ceze. Cooperative empirical failure avoidance for multithreaded programs. In *ASPLOS*, 2013.

[19] B. Lucia et al. Isolating and understanding concurrency errors using reconstructed execution fragments. PLDI '11, 2011.

[20] C.-K. Luk et al. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI*, 2005.

[21] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, PLDI '07, 2007.

[22] S. Park, R. W. Vuduc, and M. J. Harrold. Falcon: Fault Localization in Concurrent Programs. In *ICSE*, 2010.

[23] Y. Shi, S. Park, Z. Yin, S. Lu, Y. Zhou, W. Chen, and W. Zheng. Do I use the wrong definition?: Defuse: definition-use invariants for detecting concurrency and sequential bugs. In *OOPSLA*, 2010.

[24] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. In *PLDI*, 2007.

[25] S. Tallam, C. Tian, and R. Gupta. Dynamic slicing of multithreaded programs for race detection. In *ICSM '08*, 2008.

[26] D. Tam, R. Azimi, and M. Stumm. Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors. In *EuroSys '07*, 2007.

[27] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 1995.

[28] E. Vlachos et al. Paralog: enabling and accelerating online parallel monitoring of multithreaded applications. In *ASPLOS*, 2010.

[29] B. P. Wood, J. Devietti, L. Ceze, and D. Grossman. Code-centric communication graphs for shared-memory multithreaded programs. *Technical Report UW-CSE-09-05-02*.

[30] B. P. Wood et al. Composable specifications for structured shared-memory communication. In *OOPSLA 2010*, 2010.

[31] J. Yu and S. Narayanasamy. Collection of Concurrency Bugs . `http://web.eecs.umich.edu/~jieyu/bugs.html`, 2013.

[32] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. Reps. Conseq: detecting concurrency bugs through sequential errors. ASPLOS '11, 2011.