Time-Ordered Event Traces: A New Debugging Primitive for Concurrency Bugs

Martin Dimitrov
Software and Services Group
Intel Corporation[1]
Chandler, AZ
martin.p.dimitrov@intel.com

Huiyang Zhou
Department of Electrical and Computer Engineering
North Carolina State University
Raleigh, NC
hzhou@ncsu.edu

***Abstract*—Non-determinism makes concurrent bugs extremely difficult to reproduce and to debug. In this work, we propose a new debugging primitive to facilitate the debugging process by exposing this non-deterministic behavior to the programmer. The key idea is to generate a time-ordered trace of events such as function calls/returns and memory accesses across different threads. The architectural support for this primitive is lightweight, including a high-precision, frequency-invariant time-stamp counter and an event trace buffer in each processor core. The proposed primitive continuously records and timestamps the last *N* function calls/returns per core by default, and can also be configured to watch specific memory addresses or code regions through a flexible software interface.**

To examine the effectiveness of the proposed primitive, we studied a variety of concurrent bugs in large commercial software and our results show that exposing the time-ordered information, function calls/returns in particular, to the programmer is highly beneficial for diagnosing the root causes of these bugs.

Keywords-component: debugging; concurrent; bugs;

I. INTRODUCTION

Debugging shared-memory parallel programs is notoriously hard due to the inherent non-determinism present in these programs. The non-determinism stems from the fact that multiple threads may run simultaneously on different processor cores or alternate on a single core as scheduled by the operating system. Usually these threads are not independent and they cooperate through sharing resources. Concurrently executing threads may potentially interleave their accesses to shared resources in an arbitrary fashion. If software developers fail to anticipate all the possible thread interleavings, they open the door for possible erroneous behavior.

Since concurrency bugs are usually caused by such unforeseen thread interleavings, they may be very difficult to diagnose. In addition, since these bugs may manifest only under certain thread interleaving they may be very difficult to reproduce reliably, causing only sporadic failures. The difficulty in reproducing those bugs also makes the use of traditional cyclic debugging techniques, such as breakpoint debugging, even more difficult.

In this work, we propose a debug primitive to facilitate debugging parallel programs by making non-determinism visible to the programmer in the form of a time-ordered trace of events. The architectural support for the proposed primitive is lightweight and incurs negligible performance overhead. The key idea is to assign a high-precision, frequency-invariant time-stamp to different events of interest. Then, we continuously buffer the last *N* time-ordered events so as to reconstruct thread interleavings just before a failure point. The motivation is that when debugging a program, the programmer is usually not interested in the execution history of the entire program. Instead, she is interested in answers to specific questions related to a failure, such as: "Which threads were actively executing just before the crash, and how were they interleaved?". In a sense, the proposed primitive provides a way for post mortem analysis, like a core dump or stack trace for parallel program execution. Our case studies show that the time-ordered trace based on the last *N* function call/return events before a failure is typically the most effective way to reason about the root causes.

In addition, through a flexible application programming interface (API), we allow the programmer to direct the primitive to record different types of events and to dump traces of events at any point in time. For instance, the programmer can direct the primitive to monitor function interleaving in a specific region of code. This mechanism seeks to answer questions like "which functions were executing concurrently to function *foo()*, when the incorrect results were produced?". The programmer can also direct the primitive to monitor interleaved accesses to shared variables in a user-specified region. This mechanism serves similar goals to the 'watch' primitive [40] in a debugger but extends it to be a 'parallel' watch. Such a watch can also be used together with function call/return monitoring.

Compared to existing work on concurrency bug detection, the goal of our proposed scheme is not to provide an automated approach to detecting a specific type of bugs. Instead, it is a generic primitive, which we believe is helpful in a wide variety of scenarios. Compared to record/replay for reproducing concurrency bugs, our approach is much more light-weight, exposing non-deterministic events only for a limited (but likely the most useful) scope. Our focus towards light-weight, allows our approach to be always-ON and minimally perturb program execution. Overall, the contributions of this work include:

- A debugging primitive to construct time-ordered traces.

[1]This work was performed while at the University of Central Florida

- The lightweight architectural support and the software interface for the proposed primitive.

- An evaluation of the proposed primitive with a variety of bugs from large production software (MySQL and Mozilla), including: deadlock, atomicity violations, order violations and logical concurrency bugs. Among them, logical concurrency bugs are largely un-addressed by previous research.

The rest of this paper is organized as follows. Section II discusses the current state of art. Section III describes our proposed primitive and the architectural support. Section IV addresses the experimental methodology. Section V presents our detailed case studies. Section VI highlights the limitations of our approach and discusses the future work. Section VII concludes the paper.

II. CURRENT STATE OF THE ART

Given the difficulty of debugging concurrency bugs, various techniques have been proposed in previous work. Many of those techniques have focused on automatically detecting data races [11][12][28][31][34][35] and more recently atomicity violations [14][18][38] and order violations [21]. Unfortunately, such automated techniques usually focus on the synchronization of a single (or several [20][21]) variable(s) and fail in more complicated scenarios where the bug involves complex data structures, or the file system. Moreover, while these techniques are helpful for some bugs, our experiments show that there are many more bugs which remain unaddressed – for instance complex atomicity bugs, and logical concurrency bugs. We discuss these in more detail in our case studies in Section V.

Static [13] and dynamic [9] deadlock detection techniques have also been proposed. While static approaches are highly valuable, they require program annotations and may potentially cause a large number of false-positives. This would be especially true in database management systems (such as MySQL, DB2, etc.), where many deadlocks are not bugs and are handled automatically by the database management system. On the other hand, if a deadlock bug has already slipped into production, dynamic approaches incur substantial performance overheads and thus are not suitable for always-ON or production runs.

Another direction aims towards reproducing concurrency bugs, by utilizing record and replay tools, either purely in software [7][16][17][29][30] or with hardware assistance [15][23][24][26][27][37][39]. The purpose of these tools is to capture the sources of non-determinism in a multi-threaded program (e.g. the order of accesses to shared memory) so that when a bug is triggered, the program may be replayed with the same thread interleaving. Record/replay schemes enable cyclic debugging for parallel programs since the bug may be reproduced every time. Recording schemes may also provide the illusion of debugging backwards in time [10]. On the downside, these mechanisms either incur high performance overheads or require significant hardware changes. Moreover, large trace files, which grow at a high data rate, are typically required for record and replay. Such overheads may be prohibitive for always-ON use, especially in scenarios where the bug is very rarely triggered. The performance overheads, in addition, can perturb the program execution so much, that the bug is no longer triggered. As discussed in Section I, our proposed primitive is different from the record/replay schemes. It, however, can benefit from deterministic replay. For example, in a debugger with replay capability, the primitive can be used to monitor different code/memory regions in different reruns. On the other hand, our primitive can be a standalone mechanism and is not dependent upon deterministic replay.

Generating debug traces is not a new idea, and 'printf()' is arguably the most commonly used way for such a purpose. Ayers et al. [8] use static binary translation in order to automatically enable a piece of software to generate a debug trace. Many commercial software products are also able to produce a trace if compiled with a debug build. For instance the MySQL server is capable of producing a very detailed trace of function calls/returns, indented by the call-depth and marked with the thread ID of the executing thread. In this work, we revisit this fundamental concept of creating traces, in the context of multi-threaded programs. We believe that *efficiently* creating a trace is even more valuable in the context of concurrent programs. Unfortunately, creating traces completely in software, as in [8] or as in a MySQL debug build, incurs significant performance overheads (up to 2.5X in [8] and about 100X in MySQL in our experiments), which prevents the use of such traces for long periods of time, or always-ON. Moreover, software only traces are not able to reveal fine-grain time-ordering of events since trace collection (e.g., with 'printf()') may perturb thread interleaving.

Virtual machines, such as the Java VM have also been enhanced with the capabilities to record traces. The tracing facility in Java is also able to time stamp method calls with microsecond precision [6]. Still, there is no certainty that this precision is enough and performance may still be significantly impacted.

More recently, Nagarajan et al. [25] proposed to expose cache coherence events in a multicore system to the software. The proposed mechanism was used to support speculation and record-replay systems. However we believe that it is too low-level to be used directly by the programmer for debugging.

III. TIME-ORDERED EVENT TRACES

Our proposed approach consists of a software interface that allows the programmer to communicate with the trace collection engine, and a hardware component which facilitates the efficient collection of time-ordered event traces. In this section we elaborate on our design.

A. Software Interface

For the purpose of debugging parallel programs, we found that the program events likely to be most useful to the programmer were function calls/returns and memory reads/writes. Thus, in this work we focus on collecting only those events. However, our approach can easily be extended to collect other events as well (such as branch outcomes or branch mispredictions, cache misses, etc.) if they are deemed useful. By default our approach collects only function

call/return events but we allow the programmer to watch memory region(s) for their read/write events as well.

As discussed in Section I, the most common use-case that we encountered is, when the programmer is interested in the sequence of events which have occurred just before a program failure, such as a crash, incorrect results, etc. In this use-case, the programmer configures the primitive to dump last *N* events in each thread, when the program receives a signal, upon program termination or upon a condition which checks for incorrect results. The number of collected events, *N*, is fixed and we assume that *N* is equal to 2K loads/stores, or about 4K calls/returns (since we are able to combine calls and returns, see Section III-B). We chose this number *N*, since it has been sufficient in all the debugging scenarios that we have examined (see Section V).

In some cases, the programmer may require more control and may want to examine the time-ordered traces for specific code region or memory location (parallel watch). We provide such functionality by allowing the programmer to dump the last *N* events at any point in time, or to dynamically enable/disable tracing in different code/memory regions.

To support the envisioned debug primitive, several API calls are designed to communicate with the trace collection engine. These APIs may be invoked from within the application (by modifying the source code), or externally using a tool such as GDB. For instance if the program enters a deadlock, we would like to dump the last *N* event at that point without recompilation. Or if we know that the bug may be triggered by some external event, such as pressing on a button in a web browser, we might want to dump the trace to a file just after that point without modifying the binary. The proposed APIs include:

- *trace_dump(output_file)* – dump the trace (last *N* events) to a file.

- *trace_mem_start(addr_low, addr_high)* and *trace_mem_stop()*. The *trace_mem_start* API call enables the programmer to specify a memory region for watch (i.e., collecting load/store events) across all threads. It is terminated by *trace_mem_stop()* or when it leaves the scope of the calling function.

- *trace_start()/trace_stop()* – This API call is used to start/stop collecting a trace of function call/return events. The call to *trace_start()* is terminated when it leaves the scope of the calling function, or by a call to *trace_stop()*. When trace collection is enabled, it is enabled for all threads since we want to capture the thread interleaving. If multiple threads call *trace_start()*, trace collection terminates when all of the threads call *trace_stop()* or they all exit out of the scope of the function(s) from where they called *trace_start()*.

- *trace_skip(function_name)* – do not include function *function_name* in the trace. This API call can be useful if the programmer knows that a certain function, e.g., a standard library function, is not involved in the bug, but this function is called repeatedly and polluting the trace.

- *set_call_depth(depth)* - function calls/returns greater than *depth* will not be collected. This is useful for filtering out some un-interesting low-level function calls/ events.

B. Architectural Design

The main goals of using architectural support to construct traces are: fine-grain time-stamping of events and performance efficiency (so as to not perturb program execution). In this section, we discuss how to achieve those goals. We limit our discussion to multi-core and multi-processor shared-memory systems in this paper.

1) Time-Stamping Events

In our proposed approach, each thread logs its events into a local trace and uses a local time-stamp counter (TSC) to time-stamp events in its trace. The local traces of each thread are then merged to construct the final global time-ordered trace. The benefits of such approach are scalability and performance efficiency, since threads do not have to contend for a single resource such as a global TSC. However, we must ensure that the local counters are synchronized and run at a constant frequency, so that the global time can be reconstructed in the final trace. The local counters must remain consistent in the presence of context switches, thread migration, virtualized execution and different frequency and low-power state settings of each processor core.

Fortunately, invariant TSCs and global synchronization across processors are not unique requirements to our approach. In fact, similar hardware support is already present in recent microprocessor architectures, such as Barcelona/Phenom from AMD® and Nehalem/Westmere from Intel®. For example, recent Intel® processors support an "invariant TSC", which runs at a constant rate equivalent to the highest processor frequency, independent of the current operating state of the processor core [4], and thus providing us with nanosecond precision timestamps. The invariant TSC (referred to as TSC for the rest of this paper) is a 64-bit per-core register, initialized during machine reset and is guaranteed not to overflow for 10 years. The frequency of the TSC remains constant across processor frequency scaling (P-states) as well as processor sleep/low-power modes (C-states and T-states). The TSC can be used in a virtualized environment, by either forcing a VM-exit upon reading the TSC or by utilizing a 64-bit TSC-offset field [5].

Another interesting feature, present in current generation microprocessors is a component called Advanced Programmable Interrupt Controller (APIC) [3]. The APIC controller is responsible for accepting and generating interrupts. The APIC controller is also able to forward interrupts to remote processors in the system, using 3 dedicated wires (in the Pentium 6 family processors) or using the system bus or QPI (QuickPath Interconnect) links in more recent processors. Once an interrupt is delivered to APIC, it may forward this interrupt to the processor core, which invokes the interrupt handler. In a multi-processor setting, APIC is useful in a variety of scenarios [1]. In one use-case, a debug breakpoint hit by one thread, must stop the execution of all threads in the process (which is the default behavior of GDB [36].) This is achieved by having the thread which first reached the breakpoint to trap into the interrupt handler and distribute an interrupt to all other threads running on different cores.

To support the API calls described in Section III-A, we utilize such existing interrupt support. In particular, on a call to *trace_dump()/trace_start()/trace_mem_start(),* the thread triggering the event generates an interrupt to all other threads in this process. This is a similar mechanism to the one used for breakpoint in multi-threaded programs. The difference is that the cores receiving the signal simply drain the event queue or enable tracing a memory region and program execution does not have to be interrupted. Other API calls are handled similarly.

In our work, we leverage the presence of high-precision invariant TSCs and interrupt control in current microarchitectures, in order to maintain consistent time-stamps across threads, and to support the API presented in Section III-A. The architectural support that we propose is illustrated in Figure 1. , with the new components added by our approach colored in gray. The new components consist of a per-core trace engine (with associated control registers for keeping track of state, such as call-depth, memory regions to trace, etc.) and a per-core event queue. The trace engine uses only 32-bit timestamps (the low half of TSC register) for marking events, in order to minimize storage space. However, the trace engine can detect a counter overflow and push the full 64-bit TSC into the event queue so that the overflow can be corrected. To avoid overflow problems, the full 64-bit TSC is also pushed into the event queue, when the trace is dumped to file using an API *trace_dump()* call, or when the thread is context switched out.

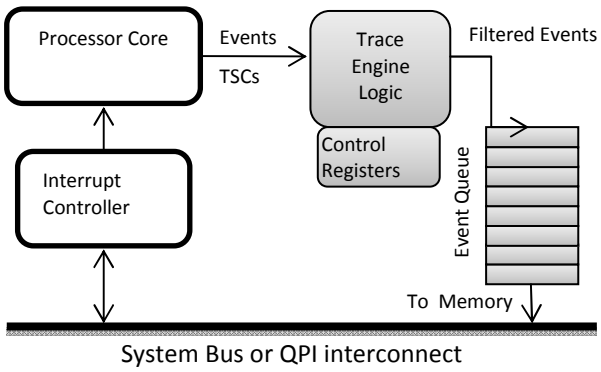

Figure 1. Proposed architectural support with the new components colored in gray.

2) Trace Collection

The trace engine in Figure 1. receives events (function calls/returns and/or load/stores) and their TSCs as the instructions retire from the pipeline. The trace engine filters those events and decides if they should be pushed into the trace. For instance, if the user has specified memory region(s) for monitoring, only those load/store instructions which have accessed the memory region will be included in the trace. The trace engine also keeps track of the current call depth of function calls/returns and includes them in the trace if their call depth is not greater than the one specified by the user. Keeping track of the call depth is also useful to detect when a function calling the API *trace_mem_start()* is returning, in which case we may need to stop tracing the memory region, as described in Secion III-A.

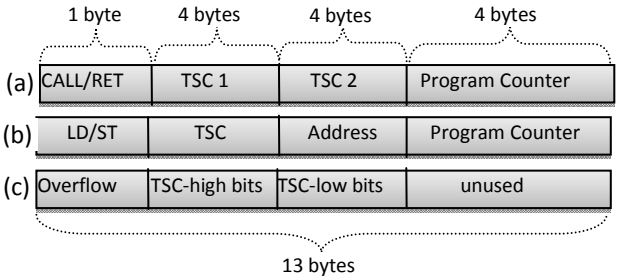

Figure 2. Time-ordered trace entry format. (a) combined call/return event (b) memory load/store event (c) 32-bit TSC counter overflow – store the full 64-bit TSC.

After the events have been filtered by the trace engine, they are pushed into an on-chip queue, called Event Queue. The event queue is managed as a circular buffer and maintains the last *N* events. In addition to buffering events, the event queue serves another useful purpose, which is to combine events and compress the footprint of the trace. For instance, if a function return event arrives, which matches a function call in the event queue, then the two events may be combined, as shown in Figure 2. (a). The first time stamp entry *TSC 1* corresponds to the call and the second time stamp *TSC 2* corresponds to the return. If the function call/return cannot be combined, then the second time slot is unused. In the common case, function calls/returns are combined, essentially reducing the storage requirements of the trace in half. Note that we do not need to perform a sequential search of the queue in order to combine call/return events. The trace engine keeps a small (e.g. 16 entries) stack of pointers, pointing to the last function call at each call-depth in the queue. When a return arrives at a given call depth it is directly forwarded to the event queue using the pointer. Load/store events may not be combined and we reuse the second time slot in the trace entry to store 32-bits of the load/store address as shown in Figure 2. (b). One additional event type stored in the trace is an overflow event, which occurs when the trace engine detects an overflow in the 32-bit counter, Figure 2. (c). In this case we push the full 64-bit TSC into the trace. From the figure we can see that each event occupies 13 bytes. Thus, the storage requirement for the event queue to maintain the last 2K events is 2k * 13 = 26kBytes per core. Due to combining of call/return events, this gives us about 4K function call/return history.

3) Performance/Area Overhead

As the trace engine and event queue receive retired instructions, they are off the critical path of the processor pipeline. They are simple logic and queue structures, with approximate area overhead of 0.15mm^2 and access latency of 0.29ns in a 32nm technology process, as computed using CACTI[2]. Given the queue size of 26KB, the time to transfer its content to memory at a memory bandwidth of 12.8 Gbytes/sec is 2 us. Such transfer happens only on a context switch, when the event queue is drained to memory in order to preserve its contents. However, the event queue

does not need to be restored when a thread is switched back in. Therefore, our debug primitive incurs performance overhead only on a context switch out, due to draining of the event queue to memory. We measured using VMstat the number of context switches/sec of various MySQL server workloads (the self-test workloads included in MySQL as well as case-studies from Section V) on our quad-core system, and we observed bursts of up to 10K context switches/second and an average of less than 1K context switches/second. Therefore, the performance overhead of our approach is up to 2% (=10K/s*2us) and 0.2% (=1K/s*2us) on average.

IV. Experimental Methodology

In order to evaluate the effectiveness of the proposed scheme, we implemented a prototype time-ordered tracing tool using dynamic binary instrumentation with Pin [22]. Using our tool, we are able to execute and trace unmodified x86 binaries on Linux. If the program executes one of the API calls (*trace_mem_start*, *trace_dump*, etc.) as inserted by the programmer, our tool detects the API calls and enables tracing of a memory region or dumps the trace to a file. We also implement a mechanism to control trace generation/dump externally by sending signals to the debugged program. Our Pin-tool intercepts the signal and dumps the trace or enables/disables tracing. Due to the performance overhead imposed by our binary instrumentation (up to 7x), some of the bugs were much more difficult to trigger. Thus we inserted sleep() or printf() when necessary to reproduce the bug. Here, note that the performance overhead of our PIN-tool cannot be attributed to our proposed hardware approach. PIN in this context is used only to evaluate the effectiveness of the time-ordered traces for bug detection and not the performance overhead, (which is addressed in Section III.B.3).

Where appropriate in the case studies, we also compare our approach to the automatic atomicity-violation detection tool AVIO [18]. AVIO works by monitoring the cache-coherence traffic and detecting unserializable interleaving to a variable. Those interleavings, which do not occur during passing program runs but only occur during a failing run, are reported to the developer. Since we do not have the extra ISA instructions to specify the I-instruction and the P-instruction used in AVIO, we slightly modified the implementation of AVIO while preserving its functionality. In our implementation, we keep track of two coherence states per word: the current coherence state and the "previous" coherence state. Whenever the current coherence state changes due to a remote access (invalidate or downgrade), we save it as the "previous" coherence state. On a subsequent access by the local thread, we check the previous coherence state, the current coherence state and the current access type and we determine if an unserializable interleaving has occurred.

V. Experimental Results

We first present a summary of our experimental results in TABLE I. . From our experiments, we found that monitoring the last *N* function calls/returns is effective to capture all the bugs that we studied. In TABLE I. we also reported the number of events *N* required to fully capture the bugs at a given call depth and we can see that and the maximum size of *N* is 3800. Based on this empirical observation, we selected 2K as the size of the event queue given its capability to combine calls and returns in one entry of the event queue. The call-depth parameter largely depends on how the software is structured and the level of detail required by the programmer. For instance a call-depth of 12 for MySQL presents a similar level of detail to a call-depth of 6 in Mozilla because of the different software structure. The call-depth may be determined statically by the programmer based on knowledge of the software structure, or it can be determined empirically by performing a couple of pre-production runs and deciding on the desired level of detail in the trace. In our current experiments, we found a static call-depth setting per program to be sufficient to prevent excessive details such as deep recursive events to pollute the traces.

In the rest of this section we present five case studies, one for each bug type: deadlock, atomicity violation, order violation and logical bugs. We also present a case study with bugs where our approach is *not* helpful.

TABLE I. Number of events N required. '*'-events required after using trace_skip() to filter out 3 functions (Section V.C).

Bug	Bug Type	N- number of Events	Call-depth
MySQL 12423	Deadlock	512	12
MySQL 29154	Deadlock	512	12
MySQL 791	Atomicity	512	12
MySQL 27499	Atomicity	2048	12
MySQL 2387	Atomicity	3072	12
MySQL 2385	Atomicity	2048	12
Mozilla 515403	Order	3800*	6
MySQL 12385	Logical	512	12
MySQL 28249	Logical	2048	12

A. Case study: Deadlock

Deadlocks occur when threads are involved in a circular wait for resources. Deadlock is one of the most common types of concurrency defects, estimated to be about 30% of all concurrency defects, according to a recent study [19]. Our experience with browsing the bug databases of MySQL and Mozilla, confirms these findings. Next, we present a detailed example of how time-ordered traces help to reveal a deadlock.

The deadlock that we present in this case study appears in the rare case, when two users connected to a MySQL server concurrently issue account management commands to the server. The account management commands may involve setting/changing passwords or permissions for databases. For instance, if one user issues the *GRANT* command, while at the same time another user issues the *FLUSH PRIVILEGES* command, MySQL may deadlock.

To debug this problem, the programmer may dump a time-ordered trace containing the last *N* events leading up to the deadlock. A simplified version of the trace is presented in

Figure 3 (a). From the figure we can easily see that threads 1 and 2 are involved in a circular wait for two resources: the *acl_cache* lock and a table lock. More importantly, from the time-ordered trace in Figure 3 (a), we can see exactly how the threads interleaved in order to reach this state. Note that this information is not available in traditional stack dumps, as we can see from Figure 3 (b). The traditional stack dump only reveals the last resource that each thread attempted to acquire. It does not reveal all the functions involved and their interleaving (the function *mutex_lock* returns successfully in thread 1, therefore does not show up in the stack dump). This is important information since in large software acquiring of locks may be nested in different functions. In fact, since we were not familiar with the MySQL code, we fully understood this deadlock only after we obtained the time-ordered traces.

	Thread 1: FLUSH PRIVILEGES	Thread 2: GRANT
1		< mysql_grant()
2		. < simple_open_n_lock_tables()
3	< acl_reload()	
4	. < mutex_lock(&acl_cache->lock)	
5	. > mutex_lock(&acl_cache->lock)	
6	. < acl_init()	
7	. . < simple_open_n_lock_tables()	
8		. . < lock_tables()
9		. . > lock_tables()
10		. > simple_open_n_lock_tables()
11		. < mutex_lock(&acl_cache->lock)
12	. . . < lock_tables() //deadlock	// deadlock

(a)

Thread 1: FLUSH PRIVILEGES	Thread 2: GRANT
< acl_reload()	< mysql_grant()
. < acl_init()	. < mutex_lock(&acl_cache->lock)
. . < simple_open_n_lock_tables()	**// deadlock**
. . . < lock_tables() //deadlock	

(b)

Figure 3. MySQL bug 12423 (a) Time-ordered trace. The first column shows the global time; next two columns show the function interleavings of the involved threads. The notation '<' means a function call, '>' means a return. The notation '...' represents the call-depth. (b) Stack dump. The stack dump alone does not reveal the deadlock interleaving leading to a deadlock.

Another deadlock bug that we investigated is MySQL bug 29154. This case is very interesting, since the innodb database engine for MySQL is able to detect and recover from deadlocks automatically. However, the problem occurs when, user_1 connected to a MySQL server sends a request to lock a number of database tables. The request may succeed for some tables, but then fail for other tables. MySQL detects this deadlock condition after a timeout, and aborts the transaction. The problem is that while aborting the transaction, MySQL does not properly release *all* the locks that user_1 has obtained. Subsequent transactions will also fail and get aborted, since they require the lock still held by user_1. Similar to our example in Figure 3, time-ordered traces are very helpful to reveal how the deadlock occurs. However, the deadlock in bug 29154 does not involve fine-grain function interleaving, and thus a regular trace (not time-ordered) could be equally helpful in this case.

B. *Case study: Atomicity Violation*

Another very common type of concurrency defect is an atomicity violation, accounting for about 30% of concurrency defects, according to Lu et al.[19]. Atomicity violations are caused when a section of the code is assumed to be atomic, however it is not properly guarded by synchronization and a remote thread may interfere. In the following case study, we also compare to the automated atomicity detection tool AVIO.

An example of atomicity violation is presented in Figure 4 (a). In this MySQL bug, the binary log is temporarily being closed, so that logging can continue to a different file. Such log rotation is usually performed, when the old log file becomes too large, or when a user explicitly issues a *FLUSH LOGS* command. Unfortunately, the developers did not realize that the closing of the old log file and the opening of the new log file must be performed atomically. In this bug, an *SQL INSERT* operation issued by a different thread fails to reach the binary log since it thinks that the log is closed.

```
new_file() {
  save_log_type=log_type; // log_type is LOG_BIN
  close() // close old log file.
    ↘ log_type = LOG_CLOSED;
            <---------- remote insert in sql_insert()
                  if(log_type != LOG_CLOSED)
                        mysql_bin_log.write() // not executed
  open(save_log_type) //open the new log file.
    ↘ log_type = LOG_BIN;
```

(a)

	Thread 1: FLUSH LOGS	Thread 2: INSERT
1	< new_file()	
2	. save_log_type=**log_type**;	
3	. < close()	
4	. . **log_type** = LOG_CLOSED;	
5	. > close()	
6		< mysql_insert()
7		. < open_and_lock_tables()
8		. > open_and_lock_tables()
9		. **log_type** != LOG_CLOSED

(b)

Figure 4. MySQL bug 791. (a) An insert operation is not being recorded into the binary log. (b) Time-ordered trace.

To start debugging this problem, we first look at the failure symptom - binary log missing an *INSERT* operation. Based on this failure, one hypothesis that the developer may have is that the *mysql_bin_log.write()* function did not execute inside function *sql_insert(),* highlighted in italics in the figure. To validate this hypothesis, and to understand what may cause this to happen, we inserted an API call to dump the time-ordered trace if the log appears to be closed. In other words, the trace is only printed out when the bug is triggered. In addition, to understand which other threads or functions modify variable *log_type*, we enabled monitoring of that memory address. The time ordered trace that we obtained is presented in Figure 4 (b). Our trace helps the developer find out which thread is causing the binary log to appear closed and why.

In our experiments, this bug was also easily detected by AVIO, which recognized the *Write (remote Read) Write*

interleaving as described in Figure 4 (a). AVIO is well suited to detect bugs involving only a single variable, such as this. However, as we show in our next example, some atomicity violations escape AVIO, since they involve more complex data structures or the file system.

The atomicity violation bug that we present in our next example occurs during concurrent execution of commands *DROP TABLE* and *SHOW TABLE STATUS*, submitted to the MySQL server. The outcome of the bug failure is that the command *SHOW TABLE STATUS* fails with an error message about an un-existing table. As we can see from Figure 5. , function *get_all_tables()* (implementing command *SHOW TABLE STATUS*) consists of two main logical components. First, it scans the file system and creates a list of tables, *make_table_list()*. Second, based on that list, it opens each table and displays the required status information, *open_normal_and_derived_tables()*. Creation of the table list and displaying the status information need to be performed atomically, since the table list may change between the two operations, e.g. a table may be dropped. Interestingly, the bug is not fixed by enforcing atomicity, since it would severely limit concurrency. Instead, atomicity violations are allowed to happen, but are later detected and handled correctly.

	Thread 1 DROP TABLE	Thread 2 SHOW TABLE STATUS
1		< get_all_tables()
2	< mysql_rm_table()	
3	. < mysql_rm_table_part2()	
4	. . < ***my_delete()***	
5		. < ***make_table_list()***
6		. . < ***add_table_to_list()***
7	. . > ***my_delete()***	
8		. . > ***add_table_to_list()***
9		. > ***make_table_list()***
10		. < open_normal_and_der_tables()
11	. . < Query_cache_invalidate()	
12	. . > Query_cache_invalidate()	
13		. . < open_table()
14	. > mysql_rm_table_part2()	
15		. . . < openfrm()
16	> mysql_rm_table()	
17		. . . . < my_error() // error displayed

Figure 5. MySQL bug 27499. Command DROP TABLE may race with command SHOW TABLE STATUS.

Since command *SHOW TABLE STATUS* results in an error message, our debugging strategy is to simply dump a time-ordered trace of the last *N* function calls/returns leading up to the error message. Once the error is triggered, we examine the trace, which is shown in Figure 5. . Interestingly, from the trace we can see that function *my_delete()* in thread 1 runs simultaneously with *make_table_list()* in thread 2, highlighted in italics. However, the table is added to the list just before it is deleted from the file system. Subsequently, function *open_normal_and_der_tables()* in thread 2 fails and we continue to print an error message.

AVIO is not able to catch this atomicity violation, since the race is to the file system and no variables were shared in memory; one thread is deleting a file while another thread is scanning the directory and reading files. In addition to this bug, we experimented with two other atomicity violation bugs involving the file system – MySQL bugs 2385 and 2387. Both of those bugs involve concurrently executing commands such as *CREATE TABLE* or *ALTER TABLE*. In both cases the atomicity violation results in a corrupted or overwritten table definition file (.frm). By creating a time-ordered trace during the execution of these commands, the bugs are clearly exposed with our approach. On the other hand, these bugs cannot be detected by AVIO. Notice that even though *CREATE* and *ALTER* may manipulate the same table, AVIO is not able to detect the bug. The reason is that when the threads open a table using *open_table()* they get a new table object populated from the table definition file (.frm). Thus there is no data race on the table object in memory.

C. Case study: Order Violation

Order violation bugs are caused when the desired order of two operations performed by different threads is flipped. Lu et al. [19] reported a large number of order violation bugs (15 out of 41 studied bugs) in Mozilla. Order violation bugs are particularly difficult to debug and to fix as we show in the following case study. Moreover, as pointed recently [19], order violations have received little attention in the research community.

void js_DestroyContext(…) { /* If last one to enter function*/ js_FinishRuntimeNumberState() }	void js_DestroyContext(…) /* If not last to enter function */ js_SweepScriptFilenames() }

(a)

	Thread 1: Next-to-last	Thread 2: Last
1		< js_DestroyContext /* last */
2	< js_DestroyContext /* not last*/	
3	. < js_GC	
4		. < ***js_FinishRuntimeNumState***
5		. > ***js_FinishRuntimeNumState***
6		. < js_GC
7		. > js_GC
8	// Crash	> js_DestroyContext
9	. . < ***js_SweepScriptFilenames***	

(b)

Figure 6. Mozilla bug 515403. (a) Function *js_FinishRuntimeNumberState* deallocates a hash table that *js_SweepAtomState()* later uses. The correct order is specified with an arrow. (b) Time-ordered trace. Function *js_SweepScriptFilenames* underlined in the trace is not captured by our trace, without filtering some repetitive functions, due to the space limitation of the event queue.

Consider the order violation bug in Mozilla shown in Figure 6 (a). This bug is present in the java script engine in Mozilla. After a java script thread has finished execution, it calls function *js_DestroyContext()* to cleanup and remove its context from the list of active contexts. In case the thread is the last one entering *js_DestroyContext()* it also performs more extensive garbage collection and de-allocates storage associated with the java script runtime. Unfortunately, under a certain rare thread interleaving, a next-to-last thread may take a very long time to execute in *js_DestroyContext()*. In the meantime, the last thread to enter the function advances

faster and de-allocates storage still in use by the previous thread, causing a crash. This bug has proven to be very difficult to reason about (we have presented only a much simplified version for clarity), since it has been fixed multiple times and reappeared in different forms for the last 7 years!

Since the bug results in a crash, a natural debugging strategy using our approach is to dump a time-ordered trace at the time of the crash, shown in Figure 6 (b). From the trace, we can observe that the two threads enter *js_DestroyContext()* almost simultaneously, however thread 2 appears to be the last one entering the function since it calls *js_FinishRuntimeNumberState()* (only the last thread calls this function) followed by garbage collection in function js_GC(). In the meantime, thread 1 has taken a long time to execute, and crashes during garbage collection in *js_GC(),* since some of the data that it uses has already been de-allocated. Our approach helps to expose the thread interleaving leading to the crash. However in this bug, our approach does not expose the exact location of the crash, *js_SweepScriptFilenames()* underlined in the figure, due to the call-depth limit that we have imposed. If we wanted to capture this additional level of detail by increasing the call-depth, then the trace would become too large (5000 call + return events) and not fit the event queue. The reason is that function *js_GC()* performs a lot of repetitive work and calls many functions in a loop, which polluted our trace. Fortunately, by using the API call *trace_skip(),* the programmer can filter only 3 such repetitive functions to reduce the size of the trace to 3800 events, which now fits in the event queue.

Another very interesting order violation bug that we studied is Mozilla bug 388714. This bug was very difficult to debug, taking about a year after it was first reported. Moreover this bug caused a lot of angry Mozilla customers, some even threatening to stop using Mozilla. The reason is that when the bug was triggered, by pressing the refresh button on a web-page, it caused Flash banners to appear blank. This of course is unacceptable since advertisers paid for these banners.

In Mozilla, Flash banners live in shells called *iFrame*. An *iFrame* is created by a call to function *doCreateShell()* and then it is populated with content by a call to function *OnStartRequest()*. During a buggy thread interleaving, function *OnStartRequest()* may execute before function *doCreateShell()* and thus attempt to stream content (flash movie) to a frame that doesn't exist.

To debug this problem, we would send a signal to Mozilla to dump a time-ordered trace, once we see the blank banner. Unfortunately, in our experiments we were not able to generate a trace for this bug, since our pin-tool was not able to link the PCs of instructions to the corresponding source-code functions. We tried compiling Mozilla with a static build, which would help identify the functions corresponding to each PC. The Mozilla version containing this bug also contained other bugs which prevented us from generating a static build. However, based on our understanding of the code, and based on discussions and stack dumps provided in the bug report, we believe that our approach would be helpful in this case. In particular, if we dump a trace after a flash banner appears blank, we should be able to observe the flipped order of execution of functions *doCreateShell* and *OnStartRequest* in our time-ordered traces.

D. *Case study: Logical Concurrency Bug*

One class of concurrency bugs that we encountered, and that has received little attention in the research community, is a type of bug that we call *logical concurrency bug*. In this type of bug, the thread collaboration and interleaving are legal and allowed so that concurrency is promoted. However, certain thread interactions to shared data structures are not well anticipated and may result in incorrect results. To make our discussion concrete, we look at an example.

MySQL connection 1	MySQL connection 2
> LOCK TABLE t1 READ LOCAL;	
	// concurrent insert
// Store output of query in the QC	> INSERT INTO t1 VALUES (),(),();
> SELECT * FROM t1;	// Obtain stale data from the QC
	> SELECT * FROM t1;

(a)

	Thread 1 SELECT	Thread 2 SELECT (reads QC)
1	< dispatch_command()	
2	. < mysql_parse()	
3	. . < mysql_exec_command()	
4	. . . < select_result()	
5	. . . > select_result()	
6	. . . < ***QC::store_query()***	
7	. . . > ***QC::store_query()***	
8	. . > mysql_exec_command()	
9		< dispatch_command()
10		. < mysql_parse()
11		. . < ***QC::send_result_to_client()***
12		. . > ***QC::send_result_to_client()***
13		. > mysql_parse()
14		> dispatch_command()

(b)

Figure 7. MySQL bug 12385 (a) Interleaving of commands issued by two users connected to the server. We assume that a table named t1 already exists in the database. (b) Time ordered trace.

MySQL server maintains a software structure called Query Cache (QC) in order to improve the performance of some queries. For example, if we issue a *SELECT* query to the server, the server retrieves the data from file, but it also buffers the data in the query cache for possible future reuse. If we issue the same *SELECT* query a second time and the database has not changed in the meantime, the data will be read from memory instead of the file system. However, the query cache has to be used with care in concurrent scenarios, especially if some of the threads are modifying data, which is exactly the issue with MySQL bug 12385. The problem is illustrated in Figure 7 (a) and involves two users submitting commands to MySQL server. First, user1 obtains a *READ LOCAL* lock on table *t1*. This lock allows other users to perform concurrent inserts on the table. However, these inserts will not be visible to the user holding the lock. Next, user2 performs some inserts to table *t1*. Then user1 queries the data in *t1*, and places the results into the query cache. Because of the read-local lock, this data does not contain the

inserts from user2. Finally, user2 queries the data and expects to get the latest copy. Instead, user2 receives stale data from the query cache. This bug is fixed by disabling the query cache when the read-local lock is held, and letting the user obtain a fresh copy of the data from file.

Our debugging strategy in this case, is to dump a time-ordered trace immediately after user2 discovers the incorrect results. This allows the user to determine where the results came from and the possible bug cause. Figure 7 (b) shows a simplified version of the dumped trace. The last several commands from thread 2 (corresponding to user2) clearly reveal that it has obtained its data from the query cache (functions are marked in italics in the figure). Going back a little further in history and we can find the thread responsible for placing the data into the query cache, command *store_query()* in thread 1. In this particular bug, the thread interleavings are rather coarse-grain, thus even a regular (not time-ordered) trace would probably be sufficient to reveal the issue. In this case our approach is still beneficial since it enables the *efficient* trace collection.

Another logical concurrency bug related to the query cache is MySQL bug 28249. This bug is more involved and requires at least three threads to reproduce. It occurs under the following interleaving. First, thread1 issues a *SELECT* statement joining two tables, *t1* and *t2*. The command obtains a lock on table *t1* and opens table *t1* for reading. However, table *t2* is already locked exclusively by thread2, causing thread1 to wait for the lock to be released. In the mean time, thread3 performs concurrent inserts to *t1*. After thread1 obtains the lock on *t2*, it completes the command and places the results in the query cache. The problem is that thread1 opened table *t1* for reading before the concurrent inserts, thus placing stale data in the query cache. To detect this problem, thread1 must check the size of the table (to detect the concurrent inserts) before placing the data in the query cache. To debug this problem, we dumped a trace of the last *N* event after obtaining the incorrect results. Similarly to the previous example in Figure 7, our approach captures the interleaving of the threads and reveals that the results are supplied by the query cache. In addition, it reveals that thread 3 performed concurrent inserts while thread 1 was waiting for a lock, and that thread1 is responsible for placing the data into the query cache.

We believe that capturing such complex logical bugs using a fully automated approach is very difficult, since understanding of the bug requires semantic knowledge of the program. For instance, the query cache was designed to be a concurrent data structure and simply monitoring data races to this data structure is not likely to reveal the bug. Our approach on the other hand is valuable is these cases, since it brings out the non-determinism or thread interleaving and facilitates the programmer in searching for the root cause.

E. *Case study: Concurrency Bugs Difficult to Debug With Time-Ordered Traces*

In our experiments, we found that time-ordered traces are most useful for debugging when the programmer inspects the sequence of the last *N* events leading up to a program failure, such as assertion failure, or crash. Alternatively, time-ordered traces are useful when the programmer has a specific hypothesis in mind and wants to observe the interleaving of events during a particular execution period, such as a function call. However, just as any other debugging primitive (e.g., watches, stack dump), time-ordered traces may not always be the best or most direct approach to debug a problem. In particular, in some cases the tracing a large trace may become tedious. In other cases, there simply is a more direct approach to debugging the problem than by using a trace. We illustrate this issue with the next example.

MySQL connection 1	MySQL connection 2
> FLUSH TABLES WITH READ LOCK > perform database backup > UNLOCK TABLES	> RENAME TABLE a TO b;

Figure 8. MySQL bug 2397. RENAME TABLE is not blocked by FLUSH TABLES WITH READ LOCK.

In this example, we present a MySQL bug, which may lead to a corrupted database backup. Figure 8. shows a sequence of commands submitted by two users to a MySQL server. The first user intends to perform a database backup and thus she issues the command *FLUSH TABLES WITH READ LOCK*. This command closes all open tables and acquires a global read lock. A read lock still allows other users to read from the tables, but not modify them. Next, the second user opens a connection to the database and issues a *RENAME TABLE* command. This command is expected to block until the global read lock is released. However, the implementation of the command does not follow the correct protocol and never checks the global read lock. Instead, the rename command succeeds immediately and renames a table potentially in the middle of a backup operation. The simple bug fix is to force the rename command to acquire the global read lock before proceeding.

The failure symptom of this bug is a potentially corrupted backup. In this case, if the programmer generated a trace covering the entire backup operation, the trace would be very large and tedious to examine. A much more direct approach to debugging this problem would be to examine the corrupted backup or the binary log and narrow down the problem to the renamed table and thus the *RENAME* operation. Alternatively, if the developer already knows that the problem lays in the *RENAME* command, then generating a trace during the execution of that command still does not reveal any *additional* useful information.

VI. LIMITATIONS AND FUTURE WORK

As addressed in Section V-E, due to the limited size of our on-chip event queue, the amount of history retained may not be sufficient to reveal the bug in some cases. In other cases it may become too tedious for the programmer to examine large debugging traces. We believe that a promising direction to address those issues is to automatically classify events as “un-interesting” and remove them from the trace. For instance, we may use the compiler to determine which functions access shared resources. Such functions will be retained in the trace, and the other functions discarded. Such an approach will decrease the clutter of un-important events

in the trace, and increase the amount of history that we can buffer. We implemented a simple filter, which removes functions which only access the stack and no other memory. Our preliminary results are promising and show that even such a simple filter can eliminate about 20% to 30% of function calls in some traces. Such automatic filtering is in addition to any manual filtering that the developer applies using API calls such as *skip_function()*.

In addition to filtering of un-interesting events, browsing through large amounts of trace will be significantly facilitated, if the user can zoom-in and zoom-out in terms of detail and call-depth. To achieve this, we implemented a prototype tool, which parses the trace into a clickable (expand/collapse) HTML/Javascript page. The tool is available at [41].

Since the proposed architectural support is able to provide very fine-grain time-stamps for function call/return events, we expect that such a tool will be very helpful for performance debugging, especially in analyzing multi-threaded real-time applications. We leave further investigation along this direction as part of future work.

Another future direction is a mechanism to support program evolution and not just debugging. For instance, once it is discovered that a certain thread interleaving is illegal, we could create a "concurrent assertion", which enforces that rule. For example: *assert(execute function B, only after function A has already been executed in another thread)* – to verify an interleaving and help with order violation bugs.

VII. CONCLUSIONS

In this paper, we present a new debugging primitive for parallel programs. With lightweight architectural support, the proposed primitive can produce a time-ordered event trace to expose thread interleavings of interest. We evaluate the primitive with a variety of concurrent bugs and our results show that the debugging process can be significantly facilitated with time-ordered traces, function call/return traces in particular. The primitive can also be used as parallel watches for specified memory addresses. Overall, based on its effectiveness and the low cost, we make a case for the debugging primitive to be supported in future multi-core processors.

ACKNOWLEDGMENT

This research is supported by an NSF grant CNS-0905223 and an NSF CAREER award CCF-0968667.

REFERENCES

[1] Mike Rieker, "Advanced *Programmable Interrupt Controller*", http://osdev.berlios.de/pic.html, 2009.

[2] CACTI, http://www.hpl.hp.com/research/cacti/, 2010.

[3] Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, chapter 10, 2010.

[4] Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, chapter 16.11, 2010.

[5] Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, chapter 21.6.5, 2010.

[6] "Java technology, IBM Style: Monitoring and problem determination,"www.ibm.com/developerworks/java/library/j-ibmjava5/. 2006.

[7] Gautam Altekar, and Ion Stoica. "*ODR: Output-Deterministic Replay for Multicore Debugging,*" In SOSP, 2009.

[8] A. Ayers, A. Agarwal, J. Rhee, "TraceBack: first fault diagnosis by reconstruction of distributed control flow", PLDI 2005.

[9] U. Banerjee, B. Bliss, Z. Ma, and P. Petersen. "*Unraveling data race detection in the intel thread checker*." In STMCS, 2006.

[10] Lewis Bil, "*Debugging Backwards in Time*", in Proc. of 5th Inter. Workshop on Automated Debugging (AADEBUG) 2003.

[11] R. O'Callahan and J.-D. Choi. "*Hybrid Dynamic Data Race Detection*". In PPoPP, 2003.

[12] J. Choi, et al."*Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs*." In PLDI, 2002.

[13] Dawson Engler, and Ken Ashcraft. "*RacerX: Effective, static detection of race conditions and deadlocks*." In SOSP, 2003.

[14] C. Flanagan and S. N. Freund. "*Atomizer: a Dynamic Atomicity Checker for Multithreaded Programs*." In POPL, 2004.

[15] Derek R. Hower , and Mark D. Hill. "*Rerun: Exploiting Episodes for Lightweight memory Race Recording*." In ISCA, 2008.

[16] Thomas Leblanc and John Mellor-Crummey. "*Debugging Parallel Programs with Instant Replay*." IEEE TC, no. 4 (1987).

[17] D. Lee et al., "*Respec: Efficient Online Multiprocessor Replay via Speculation and External Determinism*", In ASPLOS, 2010

[18] S. Lu, J. Tucek, F. Qin, Y. Zhou. "*AVIO: detecting atomicity violations via access interleaving invariants*." In ASPLOS, 2006.

[19] S. Lu, et al. "*Learning from Mistakes—a comprehensive study on real world concurrency bug characteristics*." ASPLOS, 2008.

[20] S. Lu, et al."*MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs.*"SOSP,2007.

[21] B. Lucia, L. Ceze, "Finding Concurrency Bugs with Context-Aware Communication Graphs", MICRO 2010.

[22] C. Luk et al., "*Pin: building customized program analysis tools with dynamic instrumentation*", PLDI, 2005.

[23] P. Montesinos, Matthew Hicks, Samuel T. King, Josep Torrellas. "*Capo: a software-hardware interface for practical deterministic multiprocessor replay*," ASPLOS, 2009.

[24] P. Montesinos, L. Ceze, J. Torrellas, "*DeLorean: recording and deterministically replaying shared-memory multiprocessor execution efficiently*." ISCA, 2008.

[25] Vijay Nagarajan, and Rajiv Gupta. "*ECMon: Exposing Cache Events for Monitoring.*" In ISCA, 2009.

[26] S. Narayanasamy, C. Pereira, and B. Calder. "*Recording Shared Memory Dependencies Using Strata*." In ASPLOS, 2006.

[27] S. Narayanasamy, et al, "*BugNet: continuously recording program execution for deterministic replay debugging*." ISCA, 2005.

[28] R. H. B. Netzer and B. P. Miller. "*Improving the accuracy of data race detection.*" In PPoPP, 1991.

[29] M. Olszewski, J. Ansel, S. Amarasinghe. "*Kendo: Efficient Deterministic Multithreading in Software.*" In ASPLOS, 2009.

[30] S. Park, Yuanyuan Zhou. "*PRES: Probabilistic Replay with Execution Sketching on Multiprocessors*." In SOSP, 2009.

[31] E. Pozniansky and A. Schuster. "*Efficient on-the-fly Data Race Detection in Multithreaded C++ Programs*" In PPoPP, 2003.

[32] D. Perkovic and P. J. Keleher. "*Online data-race detection via coherency guarantees*" In OSDI, 1996.

[33] M. Prvulovic. "*CORD: Cost Effective (and Nearly Overhead Free) Order Recording and Data Race Detection*." In HPCA, 2006.

[34] Y. Yu, Tom Rodeheffer, Wei Chen, "*Racetrack: Efficient Detection of Data Race Conditions via Adaptive Tracking*." In SOSP, 2005.

[35] S. Savage, et al. "*Eraser: A Dynamic Data Race Detector for Multithreaded Programs*." ACM TOCS 15, no. 4 (1997): 391-411.

[36] N. Sidwell, et al. "Non-stop Multi-Threaded Debugging in GDB," 2008. http://sourceware.org/ml/gdb/2007-11/msg00198.html.

[37] M. Xu, R. Bodik, Mark D. Hill. "*A Flight Data Recorder for Enabling Full-system Multiprocessor Deterministic Replay*."ISCA, 2003.

[38] M. Xu, R. Bodík, M. Hill. "*A Serializability Violation Detector for Shared-Memory Server Programs*." In PLDI, 2005.

[39] M. Xu, R. Bodík, M. Hill. "*A Regulated Transitive Reduction (RTR) for Longer Memory Race Recording*." In ASPLOS, 2006.

[40] P. Zhou, F. Qin, W. Liu, Y. Zhou, J. Torrellas, "*iWatcher: Efficient Architectural Support for Software Debugging*." In ISCA, 2004.

[41] M. Dimitrov, H. Zhou, "Time-Ordered Traces", http://people.engr.ncsu.edu/hzhou/TracesWeb/