FReD: Automated Debugging via Binary Search through a Process Lifetime

Ana-Maria Visan[†*] *Kapil Arya*[†*] *Tyler Denniston*[†*] *Gene Cooperman*[†*]

[†] *Northeastern University*
Boston, MA, USA
{*amvisan,kapil,tyler,gene*}*@ccs.neu.edu*

Abstract

Reversible debuggers have been developed at least since 1970. Today, GDBs "target record" facility provides eminently practical reversible debugging. Such a feature is useful when the cause of a bug is close in time to the bug manifestation. When the cause is far away, one resorts to setting appropriate breakpoints in the debugger and beginning a new debugging session. For these more difficult bugs (when the cause of a bug is far in time from its manifestation), bug diagnosis requires a series of debugging sessions with which to narrow down the cause of the bug.

For such "difficult" bugs, this work presents an automated tool to search through the process lifetime and locate the cause. As an example, the bug could be related to a program invariant failing. A binary search in time suffices, since the invariant expression is true at the beginning of the program execution, and false when the bug is encountered. This tool operates within a FReD (Fast Reversible Debugger).

FReD supports complex, real-world multithreaded programs, such as MySQL and Firefox. Past bug reports are reproduced. A failed program invariant is identified. One wishes to identify a program statement where the invariant holds, but it will fail at the next statement. FReD is used to automatically place the user at such a statement inside a familiar debugger such as GDB. The FReD methodology avoids the inefficiency of GDB's software watchpoints.

Further, the binary search is robust. It operates on multi-threaded programs, and takes advantage of multi-core architectures during replay.

[*]This work was partially supported by the National Science Foundation under Grants CCF-0916133 and OCI-0960978.

1 Introduction

Reversible debuggers have existed at least since 1970 [11, 38], and a steady stream of reversible debuggers have followed since [5, 9, 10, 12, 17, 18, 25, 29, 31, 32, 16]. Reversible debuggers allow one to execute program statements in the backwards direction via `reverse-step`, `reverse-next`, etc. Today, we have a highly robust reversible debugger in GDB (using the `target record` command).

Reversible debuggers alone are often not sufficient to easily track down a bug. For example, a program crashes because a null pointer was dereferenced. When was the pointer set to a null value? Similarly, a memory buffer is freed twice. An assert statement stops the program the second time that a memory buffer is freed. When was that particular memory buffer freed the first time? In both cases, repeatedly executing a reverse-next or reverse-step is impractical if the bug occurred millions of instructions ago. This work describes a new tool, *reverse expression watchpoints*, and applications thereof, on top of a reversible debugger platform, FReD *(Fast Reversible Debugger)*. This tool automates an otherwise impractical manual search for the original bug.

This work demonstrates an automated search for bugs, on top of a custom reversible debugger. It operates on complex, real-world programs and takes advantage of multi-core architectures for fast replay. The novelty lies in the automated search through a process lifetime. Nevertheless, a prerequisite of this work is a reversible debugger that supports multi-core architecture on replay. The support for multi-core is needed in order to replay at reasonable speeds on the emerging many-core CPUs. Support for determinism is needed not only to uniquely replay thread races, but also asynchronous signals (invocation of signal handlers), I/O (and particularly input), and system calls that poll the system clock.

Record-replay and deterministic replay are themselves old ideas. In 2000, Boothe [5] had already produced a

single-threaded reversible debugger based on recording system calls into a log, and then replaying — for the sake of determinism. More recently, there has been a wealth of systems providing support for deterministic replay through a variety of mechanisms [4, 7, 8, 14, 18, 19, 20, 21, 24, 28, 29, 35]. There is also interesting work on making the initial execution deterministic [3, 6, 22, 26]. It may be possible to employ one of these systems in the future, but at present, they are not sufficiently integrated with the use of standard debuggers such as GDB.

FReD itself is built as a Python script on top of largely standard components: an unmodified GDB debugger, the DMTCP checkpointing package [2], and a custom-written DMTCP module for deterministic record-replay. The DMTCP module employs wrapper functions around system calls, as well as using trampolines.

The goal of this work is to diagnose *difficult bugs*. There are many "gratuitous bugs", whose nature is quickly and easily diagnosed. The cause of such bugs is immediately apparent from a single run within a conventional debugger. We say that such bugs show good *temporal locality*.

The goal here is the *difficult* bugs that do not show good temporal locality. Unfortunately, simply using a reversible debugger (reverse-next, reverse-step, reverse-continue) is not a good match for these difficult bugs. Unless one knows the goal towards which the statements are executing, this has only limited advantages over standard execution in the forwards direction, with the use of breakpoints and repeated runs of a debugger.

Example of a Non-Gratuitous Bug. We illustrate with a simple (even simplistic) example. Suppose a program has crashed because a NULL pointer was dereferenced. In this simple regime, assert statements and GDB watchpoints may be applicable, but they would not scale toward the larger goals. The NULL pointer is the value of a global variable. One wishes to find the point in time when the variable first became NULL. With a forward debugger, one requires multiple runs to locate a possibly deeply nested function where the global variable is set to NULL. Typically, each debugging run brings one closer to the place where the global variable is set to NULL, and then a new breakpoint is set so that the next debugging run can "continue" to that breakpoint and resume the search for where the variable is being set to NULL.

This last example represents a difficult bug because it's manifestation is not close in time to its cause. The manual search within successively deeper nested functions causes the programmer to lose concentration on the high-level description of the bug. The last example, the programmer would have preferred to issue a single command: find the last occurrence in which the value of the given global variable changed from non-NULL to NULL. This delegates the search *in time* to an automated search routine. Unlike spatial searches, we are searching through the process lifetime, while probing the program state at specific points in time to evaluate an expression. In this instance, the expression is simply the value of a global variable (is it NULL or non-NULL), but everyone is familiar with scenarios requiring evaluation of more complex expressions.

Print Statements Traditionally, programmers have been modifying the target program to add `print` statements. By iteratively adding `print` statements to the program and thus re-executing it, programmers emulate some of the benefits of a reversible debugger. Of course, these methods are not interactive, and therefore they lack the benefits of a traditional debugger. More importantly, such strategies do not scale well either in code size or in time. Adding `print` statements to a program requires some knowledge of the structure of the program in order to know where each `print` statements should be placed.

Assert Statements One could also think of adding `assert` statements to guarantee the absence of cycles or absence of duplicates. However, there are two problems with using `assert` statements. First, an `assert` statement will require a linear number of `assert` executions. Second, the use of `assert` statements is not consistent with interactive debugging. Effectively, an `assert` statement is a conditional print statement. The ability to interactively call utility functions of the program (testing for cycles or duplicate entries) in the context of a debugger is qualitatively more powerful for the same reason that debugger software is a more powerful technology than the use of `print` or `assert` statements.

Binary Search With this motivation, we pull back and examine the process of diagnosing a difficult bug. Conceptually, one can divide the problem of debugging into two extremes.

1. There are bugs that could be fixed simply by a competent programmer employing a standard strategy using a symbolic debugger (no domain expertise required).

2. There are also bugs that could only be fixed by a domain expert familiar with the algorithm being debugged.

In practice, many bugs are a combination of those two extremes, and are solved in two phases. The first phase can be characterized as a search for the proximate location of the bug. A programmer employs a debugger to trace the forward execution of a process and form a hypothesis about the cause of a bug. In an effort to gather

more information, the programmer iteratively refines the hypothesis and begins new debugging sessions in an effort to locate the specific line of code causing the bug.

In the second phase, the symbolic debugger has pinpointed a local inconsistency in the state of a program. But to understand why that local inconsistency exists may take a global understanding of the algorithms and design of the program. For example, debugging a bug in a quicksort program leads one into this examination of the global algorithmic structure of a program.

Too often, a programmer spends much of his or her time in the first phase, above. For example, a null pointer is dereferenced. When was the pointer set to null? Why did the code cause the pointer to be set to null? Ideally, a reversible debugger would allow one to simply trace backwards in a program to answer the above questions. But this is a trial-and-error process.

This paper presents *reverse expression watchpoints*. This provides a novel automated search in the context of reversible debuggers. Further, it provides a powerful tool for a programmer to ask high-level questions in a program that greatly help in understanding the program.

Traditional Watchpoints Reverse expression watchpoints are a generalization of GDB watchpoints. Debuggers such as GDB and others have a "watch" command. One specifies a memory address and desires to know when the data at the address changes. This version of the GDB watch command is called a *hardware watchpoint* because it uses hardware support (i.e. the memory management unit) to efficiently stop execution when the data at that address changes.

GDB also provides a *software watchpoint* in which one invokes the watch command on an expression in the target program. Since there is no single hardware address to monitor, GDB implements watch by evaluating this expression after each statement of the target program is executed. While this works well over short periods of execution, it is not sufficiently efficient for longer execution periods. If N statements are executed, then the expression must be evaluated N times.

FReD's Watchpoints FReD demonstrates an implementation of reverse expression watchpoints that "stops" a program at a point in time when a given expression is about to change. More important, the method requires only $\log_2 N$ evaluations of the given expression. As we will see, reverse expression watchpoint is implemented through a binary search over the process lifetime. The $\log_2 N$ temporal search makes reverse expression watchpoints an important and novel tool in debugging.

The key point is that only $\log_2 N$ iterations are required for N statements executed over the life of a process. In contrast, GDB software watchpoints require N evaluations.

Outline of Paper. Section 2 describes the underlying components of FReD. Section 3 describes the core novelty of this work, reverse expression watchpoint and its implementation. Section 4 reviews the overall implementation of FReD. Section 5 provides an experimental evaluation of FReD. Section 6 describes the related work. Finally, the conclusion is in Section 7.

2 Underlying Components of FReD

FReD (Fast Reversible Debugger) incorporates both temporal search routines (search through the process lifetime) and an underlying reversible debugger. Ideally, we would have built FReD on top of an existing reversible debugger. For the reasons described below, it was required to build a custom reversible debugger.

FReD sits on top of and requires three other software packages:

1. an unmodified GDB
2. DMTCP checkpointing package
3. a custom record-replay package

First, FReD uses a standard, unmodified debugger, GDB, for its debugger. Second, it uses a transparent, user-space checkpointing package, DMTCP (Distributed MultiThreaded CheckPointing) [2]. A prerequisite for the choice of checkpointing package is one that supports checkpointing of GDB debugging sessions, which in turn must support debugging of multithreaded user programs. Third, FReD employs a custom record-replay package based on wrapper functions around system calls (calls to run-time libraries). A custom record-replay package was chosen for its ease of integration, by employing DMTCP's direct support for building third-party modules that implement wrapper functions.

In FReD, checkpoints of an entire GDB session (GDB and target application) are taken at regular intervals. The history of GDB debugging commands is recorded (in addition to recording system calls of the target application). Moving backwards in time consists of restarting from an earlier checkpoint and replaying until the desired time in the past history. Algorithms for decomposing debugging histories of commands were developed [33]. If, for example, the debugging history is `[continue, next]` and the user issues a `reverse-next`, then this is the equivalent of an `undo` command. However, if for the same debugging history, the user issues a `reverse-step` command (therefore not an `undo`), then the debugging history needs to be decomposed as in [33].

An alternative design for automated temporal search would have based FReD on top of an existing reversible debugger based on a virtual machine (VM). This was rejected for the following reason. Two recent examples of such VM-based debuggers are [12, 18]. DMTCP-based checkpoints were preferred over VM-based snapshots because DMTCP checkpoint and restart executes in about a second, while VM snapshots require half a minute or more. Further, while VM-based reversible debuggers support multithreaded executables, they do not support multi-core architectures without custom hardware support [8].

Note that several optimizations can be used to speed up checkpoints and restart — both for processes and for VM snapshots. For example, copy-on-write could be used to accelerate checkpointing of a VM, although the frequency of checkpoints is still limited by the bandwidth to disk. Nevertheless for our work, the restarts dominate over checkpoints in the binary search algorithm. King et al. [12] employ incremental checkpoints so that on restarting from a snapshot close in time to the current time, only a smaller number of memory pages must be updated. However, FReD needs to restore checkpoints that may be far away in time.

Another alternative design would have based FReD on top of an existing reversible debugger. Table 3 of Section 6 provides a review of reversible debuggers. A first prerequisite for such a debugger is that it be based on checkpoint/re-execute. As discussed earlier, VM-based debuggers are not fast enough for interactive use. (A single binary search through a process lifetime may require 50 or more checkpoints and restarts.)

A third alternative design would have based FReD directly on top of GDB or another debugger based on record/reverse-execute (see Table 3). GDB currently supports reversibility through its `target record` command. However, this family of debuggers saves the state of registers, etc., at each statement of the program. This has a serious problem. A binary search through a process lifetime requires frequent long jumps to distant portions of a program. For long-running programs, it is not practical to save and restore so much state, while maintaining a fast binary search.

Finally, since the future lies with many-core CPUs, we felt strongly about basing FReD on a reversible debugger with multi-core support on replay. Table 3 shows that only one debugger fits this criterion: the proprietary TotalView, with its Replay Engine [32]. Since the Replay Engine is based on saving and restoring state before each statement, it lacks the ability to make long jumps in time, discussed above.

2.1 Architecture of FReD

FReD uses a Checkpoint/Re-execute strategy to enable its reversibility. FReD sits between the end user and GDB (see Figure 1). FReD passes user commands to GDB and returns the debugger output. From FReD, the user interacts with GDD in the same way as without FReD. FReD uses DMTCP [2] (Distributed Multi-Threaded Checkpointing) to checkpoint the state of a debugging session to disk. One can revert to any point in the execution by restarting from a prior checkpoint image and re-executing. FReD takes multiple checkpoints so that the execution time since the prior checkpoint is never overly long. The higher layer that control GDB, DMTCP, and the record-replay mechanism is written in Python.

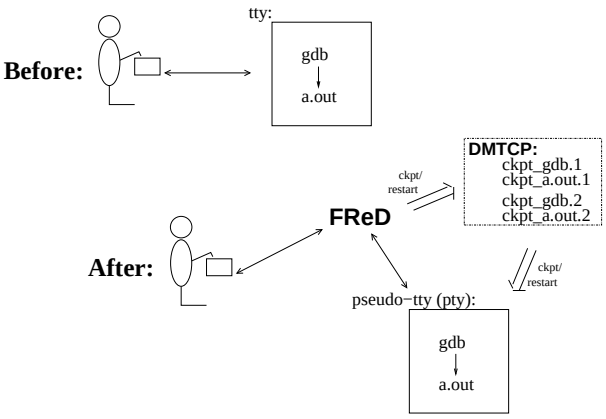

Figure 1: The architecture of FReD.

3 Reverse Expression Watchpoints

The core novelty of FReD is reverse expression watchpoints. With reverse expression watchpoints, FReD will transfer the user to the exact source statement causing the given expression to change value.

Figure 2 provides a simple example. Assume that a bug occurs whenever a linked list has length longer than one million. So an expression `length(linked_list)<=1000000` is assumed to be true throughout. Assume that it is too expensive to frequently compute the length of the linked list, since this would require $O(n^2)$ time in what would otherwise be a $O(n)$ time algorithm. (A more sophisticated example might consider a bug in an otherwise duplicate-free linked list or an otherwise cycle-free graph. But the current example is chosen for ease of illustrating the ideas.)

If the length of the linked list is less than or equal to one million, call the expression "good". If the length of the linked list is greater than one million, call the expression "bad". A "bug" is defined as a transition from

"good" to "bad". There may be more than one such transition or bug over the process lifetime. Our goal is simply to find any one occurrence of the bug.

The core of a reverse expression watchpoint is a binary search. In Figure 2, assume a checkpoint was taken near the beginning of the time interval. So, we can revert to any point in the illustrated time interval by restarting from the checkpoint image and re-executing the history of debugging commands until the desired point in time.

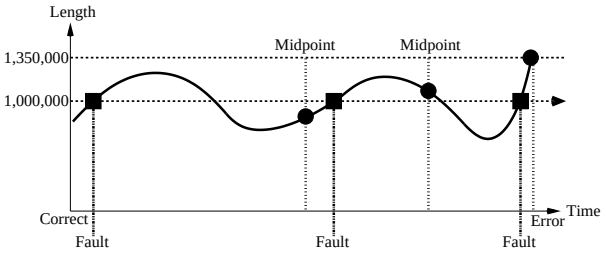

Figure 2: Reverse expression watchpoint for the bounded linked list example.

Since the expression is "good" at the beginning of Figure 2 and it is "bad" at the end of that figure, there must exist a buggy statement — a statement exhibiting the transition from "good" to "bad". A standard binary search algorithm converges to some instance in which the next statement transitions from "good" to "bad". By definition, FReD has found the statement with the bug. This represents success.

If implemented naively, this binary search requires that some statements may need to be re-executed up to $\log_2 N$ times. However, FReD can also create intermediate checkpoints. In the worst case, one can form a checkpoint at each phase of the binary search. In that case, no particular sub-interval over the time period needs to be executed more than twice.

3.1 Typical Running Times

As a binary search, the number of expression evaluations will be at most $\log_2 N$, for N statements executed. As an example, take $N = 10^{15}$ assembly instruction (the equivalent of several days of runtime on one core on a 1 GHz CPU). In this case, $\log_2 N$ is only 50.

By taking intermediate checkpoints, one can guarantee that a particular statement of code is never executed more than once during the binary search. Using this strategy, the left endpoint of the binary search will always correspond to a time in which a checkpoint is available.

In this way, the typical running time will be bounded by 50 checkpoints, 50 restarts and the time to re-execute the code in the time interval of interest. Checkpoint and restart typically proceed within seconds. So, for a reasonable running time of the code, this implies an order of magnitude of time for a reverse expression watch of between a minute and 100 minutes. This number is in keeping with the experimentally determined times of Table 1 in Section 5.

3.2 The Algorithm

The algorithm for reverse expression watchpoint uses binary search essentially at four levels:

(A) Perform a binary search on the available checkpoints in the process lifetime, to find a "bad" transition between adjacent checkpoints, or between the most recent checkpoint and the current point in time.

(B) Perform a binary search through the debugging history from the the checkpoint identified in step A. This binary search may require one to decompose the debugging history. An algorithm to decompose the debugging history is reviewed in Section 4 and addressed in [33].

(C) *Multithreaded Programs:* Perform a binary search through the deterministic replay log. This will identify a synchronized event close in time to the bug fault, and will ensure that the thread causing the fault is alive.

(D) *Multithreaded Programs:* Perform repeated `next` commands and history decomposition for each live thread, using gdb's *scheduler-locking* to discover which thread was responsible for changing the expression value. Turning scheduler-locking on disables all threads but the currently active thread from executing. If scheduler-locking is

Step D of this algorithm makes the reasonable assumption that there exists exactly one statement modifying exactly one datum which causes the expression evaluation to change. It follows that if an expression changes value, a single "step" instruction by a single thread must be enough to do it.

In performing a binary search over the debugging command history (in single or multithreaded programs), it is possible that one user debugging command was a "continue" which executed for a long time. To handle that case, FReD supports "timed reverse expression watchpoints". Hence, intermediate checkpoints are taken during the execution of a long-running "continue". In this way, FReD ensures that a moderate number of next or step instructions between checkpoints will always suffice.

3.3 Details of Algorithm

Step (A) (binary search through checkpoints) of the reverse expression watchpoints is performed first, to identify a checkpoint image from which step (B) can proceed.

Once FReD has identified a base checkpoint, a binary search through the user's debugging command history (step (B)) is performed. Once the command in the user's history after which the expression changes value has been identified, FReD "decomposes" that command by expanding "continue" and "next" commands into "next" and "step" as needed. The algorithm is described in [33] and discussed in Section 4.

FReD also ensures that a "step" command is not allowed to step into libc internal runtime functions. Certain libraries such as libc and libpthread are blacklisted, and FReD replaces a "step" by a "next" to step over calls to such libraries.

Steps (A) and (B) above suffice for a single-threaded program. Steps (C) and (D) are included only for reverse expression watchpoints on multithreaded programs.

Step (C) (binary search through the deterministic replay log) is used to ensure that the thread changing the expression is alive. Thread creation (whether through `clone()` or through another library call such as `pthread_create()`) are recorded as events in the replay log.

Finally, step (D) performs a round-robin search through the live threads, performing command expansion and decomposition (step (B)), until a candidate thread is found that caused the expression to change . A high-level description of the round-robin search of step (D) follows:

Step (D):

1. Do repeated "next" in the current thread until the expression changes (as in step (B). Then verify that this is the correct thread by re-executing the same series of debugger commands and enabling gdb scheduler locking on the last "next" command and observe if the expression still changes. If it does, we are guaranteed that this is the correct thread. If we see a deadlock, we don't know if this is the right thread. If the expression doesn't change, this is the wrong thread.

2. Undo the last "next", and replace by a single "step" followed by repeated "next" (no scheduler locking). If the expression changes on that first step, go to step 3 below. If the expression does not change, then go to step 4.

3. The expression changed on this "step". We must verify that it is due to this thread. Undo "step", enable gdb scheduler locking, and redo the the "step." If the expression changes, this is the right thread, and exit. If the expression does not change, or deadlocks, then this is not the right thread. Go to step 4.

4. Not the right thread, choose the next thread in step 1 above, and try again.

FReD uses a timeout (currently 20 seconds) in order to decide if a deadlock occuredinside step (D).

4 Implementation of FReD

As discussed in Section 2, the FReD reversible debugger consists of three components: an unmodified GDB, the DMTCP checkpointing package, and a tightly integrated custom record-replay module. The record-replay module is in fact a DMTCP module.

Record-Replay DMTCP module Record-replay is implemented in a standard way using dlopen/dlsym and, where necessary, trampolines. A single global log is used, which is mmap'ed to a file on disk so that the operating system can optimize lazy writes of the log to disk. On record, multiple threads compete for the log by using an "atomic increment". The log entries have a variable size, depending on the type of event that needs to be logged. The "atomic increment" allows a thread to reserve a log entry immediately when the event was triggered. Later on, the thread will fill in the reserved log entry.

On replay, when the thread makes a function call, the current entry of the head of the log is polled. As other threads execute synchronized events, the current entry is eventually advanced to the desired function call entry with the correct thread identifier and arguments, and the real function call is made.

Currently, each thread writes directly to the central log. In order to avoid issues of false sharing, there are opportunities for each thread to write to a local buffer, and then opportunistically merge the buffers.

Trampolines FReD mostly achieves its purpose through standard function wrappers around library functions such as libc and libpthread. In a few cases, the function was not globally visible. Interposition packages such as PIN and Dynamo implement trampolines [13, 30, 37] for this case when the address of a function is known, but no symbol is exported. However, these packages would bring added complexity. So, a simplified trampoline implementation was used.

The beginning of the function to be wrapped is overwritten with a jump to the desired wrapper function. The wrapper function must also execute the first few instructions of the target function, beforce calling the target

function beyond this prolog. On x86 and x86-64 CPUs, instructions are variable length. Further, only position-independent code can be executed inside the trampoline instead of in the target function. Since only a few functions must be wrapped with a trampoline, a simple pattern matching algorithm was used to determine the first few instructions, and verify that all instructions are position-independent.

Memory Accuracy One important feature of FReD is *memory-accuracy*. Memory accuracy ensures that the addresses of objects on the heap do not change between original execution and replay. Any reversible debugger without memory accuracy could change the address of a memory object on each iteration, and would find a poor reception among users.

In MySQL, a linked list was found to have a bad pointer in the last link, causing a segmentation fault. We needed to look backwards in time to when that pointer was first set. Since that pointer did not correspond to any variable name outside the scope of the current function, it was not possible to reversibly search by name. Only searching by address was possible, and then only with the guarantee of memory accuracy.

Memory-accuracy is accomplished by logging the arguments, as well as the return values of `malloc`, `calloc`, `realloc`, `free`, `mmap`, `mremap`, `munmap` and `libc_memalign` on record. On replay, the real functions or system calls are re-executed in the exact same order.

Implementation of Reverse-XXX The reverse commands `reverse-step`, `reverse-next`, `reverse-finish`, and `reverse-continue` each had to be written with some care, to avoid subtle algorithmic bugs. The implementation of the first three is described in [33]. The underlying principle is that a `continue` debugging instruction can be expanded into repeated `next` and `step`. Similarly, a `next` can also be expanded into repeated `next` and `step`. Thus, in a typical example, `[continue, next, next, reverse-step]` might expand into `[continue, next, step, next, step, reverse-step]`, where the last `next` expands into `[step, next, step]`. The last expression would finally reduce to `[continue, next, step, next]`. FReD uses repeated checkpoints and restarts to expand `next` into `[step, next, step]` in this example. See [33] for further details.

5 Experimental Evaluation

5.1 Methodology

All experiments were carried out on on a 16-core computer with 128GB of RAM. The computer has four 1.80 GHz Quad-Core AMD Opteron Processor 8346 HE and it runs Ubuntu version 11.10. The kernel is Linux kernel 3.0.0-12-generic. We used glibc version 2.13, gdb version 7.3-0ubuntu and gcc version 4.6.1-9ubuntu3. The kernel, glibc, gdb and gcc were unmodified.

While some other investigations have limited themselves to four cores, we felt it important to use a computer with at least 16 cores in order to test real-world scalability. Many of the recent assertions of a debugging crisis point to the difficulty of debugging highly multi-threaded software as the number of cores per CPU chip (and hence the concurrency) continues to rise. The use of 16 cores provides a crude approximation to the many-core computers of the future.

The reverse expression watchpoint feature of FReD was used to diagnose two real-world MySQL bugs (see Subsections 5.1.1 and 5.1.2), one real-world Firefox bug (see Subsection 5.1.3) and one real-world pbzip2 bug (see Subsection 5.1.4). These bugs do not satisfy the *temporal locality* property and they require examining the state of the process at least two points in time that were far apart.

For each of the following MySQL examples, the average number of entries in the deterministic replay log was approximately 1 million. The average size of an entry in the log was approximately 79 bytes.

5.1.1 MySQL Bug 12228 — Atomicity Violation

In order to reproduce MySQL bug 12228, a stress test scenario was set in which five threads issue concurrent client requests to the MySQL daemon. In our experience, this bug occurs approximately 1 time in 1000 client connections. This bug was reproduced using MySQL version 5.0.11.

The buggy thread interleaving and the series of requests issued by each client are presented in Figure 3. The bug occurs when one client, "client 1" removes the stored procedure `sp_2()`, while a second client, "client 2" is executing it. The memory used by procedure `sp_2()` is freed when "client 1" removes it. While "client 1" removes the procedure, "client 2" attempts to access a memory region associated with the now non-existent procedure. "Client 2" is now operating on unclaimed memory. The MySQL daemon is sent a `SIGSEGV`.

This bug was diagnosed with FReD in the following way: the user runs the MySQL daemon under FReD and executes the stress test scenario presented in Figure 3. The debug session is presented below. Some of the output returned by gdb was stripped for clarity.

```
(gdb) break main
(gdb) run
Breakpoint 1, at main().
(gdb) fred-checkpoint
(gdb) continue
```

Bug Number	Total Ckpt [s]	Total Restart [s]	Expr Eval [s]	#Ckpts	#Restarts	#Expr Eval	Avg Ckpt [s]	Avg Restart [s]	Avg Eval Expr [s]	Rev Watch [s]
MySQL 12228	24.84	524.81	1.54	6	62	107	4.14	8.47	0.01	812.15
MySQL 42419	8.09	316.62	1.16	4	52	80	2.02	6.09	0.01	452.69
pbzip2	3.102	48.58	0.39	1	17	27	3.10	2.86	0.02	64.79

Table 1: The bugs and the time it took FReD to diagnose them, by performing reverse expression watchpoint (in seconds). Other timings that are of interest are shown: the total and average times for checkpoint, restart and evaluation of the expression (in seconds), as well as the number of checkpoints, restarts and evaluation of the expression.

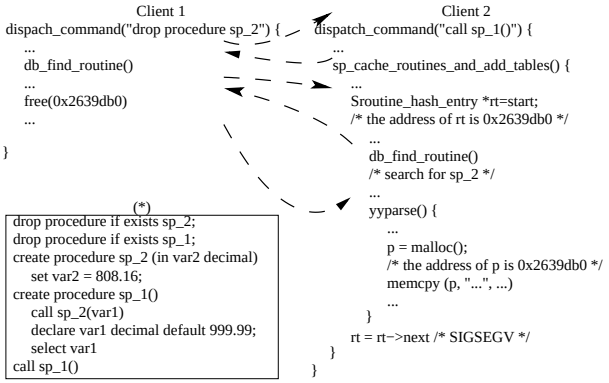

Figure 3: MySQL Bug 12228: the thread interleaving that causes the MySQL daemon to crash with SIGSEGV; (*) the sequence of instructions executed by each thread, in pseudo-SQL

```
Program received signal SIGSEGV.
in sp_cache_routines_and_table_aux at sp.cc:1340
sp_name name(rt->key.str, rt->key.length)
(gdb) print rt
$1 = 0x1e214a0
(gdb) print *rt
$2 = 1702125600
(gdb) fred-reverse-watch *(0x1e214a0) == 1702125600
```

When the SIGSEGV is hit, gdb prints the file and line number that triggered the SIGSEGV. The user prints the address and value of the variable rt. The value of rt is "bad", since dereferencing it triggered the SIGSEGV. From there it is a simple conceptual problem: at what point did the value of this variable rt change to the "bad" value? FReD's reverse expression watchpoint (or fred-reverse-watch as abbreviated above) is used to answer this question. The time for reverse expression watchpoint, as well as other useful information, are shown in Table 1.

5.1.2 MySQL Bug 42419 — Data Race

In order to reproduce MySQL bug 42419, two client threads which issue requests to the MySQL daemon (version 5.0.67) were used, as indicated in the bug report. MySQL bug 42419 was diagnosed with FReD. The debug session is shown next (some of the output returned by gdb was removed for clarity):

```
(gdb) break main
(gdb) run
Breakpoint 1, at main().
(gdb) fred-checkpoint
(gdb) continue
Program received signal SIGABRT
at sql_select.cc:11958.
if (ref_item && ref_item->eq(right_item, 1))
(gdb) where
at sql_select.cc:12097
(gdb) print ref_item
$1 = 0x24b9750
(gdb) print table->reginfo.join_tab->ref.items[part]
$2 =  0x24b9750
(gdb) print &table->reginfo.join_tab->ref.items[part]
$3 = (class Item **) 0x24db518
(gdb) fred-reverse-watch *0x24db518 == 0x24b9750
```

The crash (receiving a SIGABRT) was caused by the fact that the object ref_item did not contain a definition of the eq() function. In gdb, the value of ref_item seemed to be sane and thus the problem was not as immediately obvious as dereferencing a garbage value, for example. Then we looked at how the pointer ref_item was being created. The pointer ref_item was returned from a function part_of_refkey(). Therefore, we printed the address and value of the pointer returned by part_of_refkey(). reverse-watch takes us to the place where the pointer ref_item was assigned an incorrect value. This happens during a call to the function make_join_statistics():sql_select.cc:5295, instruction j->ref.items[i]=keyuse->val.

We then step through make_join_statistics() with next commands as in a regular gdb session and watch MySQL encounter a "fatal error." As part of the error handling, the thread frees the memory pointed to by &ref_item. But, crucially, it does not remove it from j->ref.items[]. When a subsequent thread comes along to process these items, it sees the old entry, and attempts to dereference a pointer to a memory region that has previously been freed. The time for reverse expression watchpoint, as well as other useful information, are shown in Table 1.

Reversible Debugger	Multi Threaded	Multi Core	Reverse Expression Watchpoint	Observations
IGOR [9]	No	No	x > 0	only monotonely varying single variables
Boothe [5]	No	No	x > 0	only probes where the debugger stops
King et al. [12]	Yes	No	x	detects the last time a variable was modified
FReD	Yes	Yes	Complex Expressions	detects the exact instruction that invalidates the expression

Table 2: Among checkpoint/re-execute based reversible debuggers, other examples are limited to examining single addresses, and do not support general expressions.

5.1.3 Firefox Bug 653672

This was a bug in Firefox (version 4.0.1) Javascript engine. We reproduced the bug using the test program provided with the bug report. The Javascript engine was not correctly parsing the regular expression provided in the test program and would cause a segmentation fault. The code causing the segmentation fault was just-in-time compiled code and so gdb could not resolve the symbols on the call stack causing an unusable stacktrace.

```
(gdb) break main
(gdb) run
(gdb) fred-checkpoint
(gdb) break dlopen
(gdb) continue
...
(gdb) continue
Program received signal SIGSEGV, Segmentation fault.
(gdb) where
#0  0x00007fffdbaf606b in ?? ()
#1  0x0000000000000000 in ?? ()
(gdb) fred-reverse-step
FReD: 'fred-reverse-step' took 6.881 seconds.
(gdb) where
#0  JSC::Yarr::RegexCodeBlock::execute (...)
    at yarr/yarr/RegexJIT.h:78
#1  0x7ffff60e3fbb in JSC::Yarr::executeRegex (...)
    at yarr/...
#2  0x7ffff60e47b3 in js::RegExp::executeInternal (...)
    at ...
...
```

While running the above commands to reproduce the error, we noted that the `SIGSEGV` was delivered shortly after the library `libXss.so` was loaded. We placed a breakpoint on `dlopen()` to capture the event. As soon as `dlopen("libXss.so")` was seen, we switched to issuing next commands until we hit the segmentation fault. At this point the stacktrace was already unusable and so we used "reverse-step" capability of FReD to return to the last statement where the stacktrace was still valid. The "reverse-step" took 6.881 seconds.

5.1.4 Pbzip2 — Order Violation

`pbzip2` decompresses an archive by spawning consumer threads which perform the decompression. Another thread (the output thread) is spawned which writes the decompressed data to a file. Unforunately, only the output thread is joined by the main thread. Therefore, it might happen that when the main thread tries to free the resources, some of the consumer threads have not exited yet. A segmentation fault is received in this case, caused by a consumer thread attempting to dereference the NULL pointer. The time for reverse expression watchpoint is shown in Table 1. The debugging session is presented below:

```
(gdb) break pbzip2.cpp:1018
(gdb) run
Breakpoint 1, at pbzip2.cpp:1018.
(gdb) fred-checkpoint
(gdb) continue
Program received signal SIGSEGV at
pthread_mutex_unlock.c:290.
(gdb) backtrace
#4 consumer (q=0x60cfb0) at pbzip2.cpp:898
...
(gdb) frame 4
(gdb) print fifo->mut
$1 = (pthread_mutex_t *) 0x0
(gdb) p &fifo->mut
$2 = (pthread_mutex_t **) 0x60cfe0
(gdb) fred-reverse-watch *0x60cfe0 == 0
```

6 Related Work

In this section, we compare FReD with other systems that implement reverse expression watchpoint (Subsection 6.1) and other reversible debuggers (Subsection 6.2). Deterministic replay systems are briefly mentioned (Subsection 6.3).

6.1 Reverse Expression Watchpoint

Table 2 presents other reversible debuggers that support reverse expression watchpoint.

Approach	Reversible Debugger	Info Captured	Multi Thrd	Multi Core On Replay	Forward Exec. Speed	Reverse Exec. Speed	Orth.
Record / Reverse-Execute	AIDS [11]	High	No	No	Slow	Depends on Cmd	No
	Zelkowitz [38]		No	No			No
	Tolmach et al. [31]		No	No			No
	GDB [10]		Yes	No			Yes
	TotalView '11 [32]		Yes	Yes			Yes
Record-Replay	King et al. [12]	Low	Yes	No	Fast	Slow	No
	Lewis et al. [18]	Low	Yes	No	Fast	Slow	No
Post-mortem Debugging	Omniscient Dbg [25]	Average	Yes	(*)	Slow	(*)	No
	Tralfamadore [16]	Average	Yes	(*)	Average	(*)	No
Checkpoint / Re-execute	IGOR [9]	Average	No	No	Average	Average	No
	Boothe [5]		No	No			No
	Flashback [29]		No	No			No
	ocamldebug [17]		No	No			No
	FReD	Average	Yes	Yes	Average	Fast	Yes

Table 3: The four primary approaches to reversible debugging. In the case of post-mortem debuggers, the reverse execution speed cannot be determined, since the process no longer exists. Also, post-mortem debuggers do not fit with the higher goal of this work: the capability of searching based on arbitrary expressions through the entire lifetime of the process.

Both IGOR [9] and the work by Boothe [5] support a primitive type of reverse expression watchpoint for single-threaded applications of the form `x>0`, where the left-hand side of the expression is a variable and the right-hand side is a constant. `x` is also a monotone variable. On the other hand, FReD supports general expressions.

In terms of how reverse expression watchpoint is performed, IGOR locates the last checkpoint before the desired point and re-executes from there. Boothe performs reverse expression watchpoint in two steps: the first step records the last step point at which the expression is satisfied and then the second step re-executes until that point. A step point is a point at which a user issued commands stops. In other words, Boothe can only probe the points where the debugger stops. But a `continue` command can execute many statements. FReD, on the other hand, brings the user directly to a statement (one that is not a function call) at which the expression is correct, but executing the statement will cause the expression to become incorrect.

The work of King et al. [12] goes back to the last time a variable was modified, by employing virtual machine snapshots and event logging. While the work of King et al. detects the last time a variable was modified, FReD takes the user back in time to the last point an expression had a correct value. Similarly to Boothe [5], the reverse watchpoint is performed in two steps and only the points where the debugger stops are probed.

6.2 Reversible Debuggers

Throughout the years, four different approaches to build a reversible debugger have been observed: *record/reverse-execute*, *record/replay*, *checkpoint/re-execute*, *post-mortem debugging*. Table 2 groups FReD and previous reversible debuggers according to the approach taken to build a reversible debugger.

Each different approach can be characterized by the following: the amount of information captured while executing forwards (Table 2, column 3), the type of applications that the reversible debugger can be used with (mainly, can it be used with multithreaded applications? — Table 2, column 4), the type of architectures the reversible debugger can be used on (does it run on multicore architectures? — Table 2, column 5), the forward execution speed (Table 2, column 6), the reverse execution speed (Table 2, column 7) and orthogonality (Table 2, column 8).

The amount of information captured during the forward execution is clasified as: Low (these reversible debuggers use virtual machines), Average (enough information is stored to guaranteed deterministic replay) or High (logging the state after each instruction is executed).

Forward execution speeds can be: Slow (due to excessive logging), Average (as in the case of reversible debuggers that capture enough information to guarantee deterministic replay) and Fast (native speed via the use of virtual machines).

Reverse execution speeds can be: Slow (due to large memory footprints), Average (due to the deterministic re-

play strategy), Fast (through the use of checkpoints and binary search) or can depend on the type of reverse command issued (reverse-continue and reverse-next tend to be slow, while reverse-step is fast).

A reversible debugger is considered orthogonal if it requires no modifications to the kernel, compiler and interpreter. Otherwise, the reversible debugger is non-orthogonal.

6.3 Deterministic Replay

Deterministic replay is a prerequisite for any reversible debugger that wants to support multithreaded applications. There are many systems that implement deterministic replay in the literature, through a variety of mechanisms: [1, 4, 7, 8, 9, 14, 15, 18, 20, 21, 23, 24, 27, 28, 29, 34, 35, 36]. There are also many systems whose goal is to make the initial execution deterministic [3, 6, 19, 22, 26]. It may be possible to employ one of these systems in the future, but at present, they are not sufficiently integrated with the use of standard debuggers such as GDB. Therefore, we had to implement our own system that supports deterministic replay via logging of important system calls, pthread and glibc functions. While this is not a novel approach per se, it was enough to demonstrate the novelty of our reverse expression watchpoint.

7 Conclusion

A reverse expression watchpoint algorithm has been presented for automating a binary search through a process lifetime. Reverse expression watchpoint searches for a statement at the level of source code that causes a particular GDB expression in the program to transition from a "good" value to an "bad" value. The end user determines such an expression that is associated with the bug being diagnosed.

FReD is robust enough to support reversible debugging in such complex, and highly multithreaded, real-world programs as MySQL and Firefox. All tests were run on a 16-core computer. The times required to execute reverse-watch varied from 65 seconds to 812 seconds.

References

[1] ALTEKAR, G., AND STOICA, I. ODR: Output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (2009).

[2] ANSEL, J., ARYA, K., AND COOPERMAN, G. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *23rd IEEE International Symposium on Parallel and Distributed Processing (IPDPS-09)* (2009), pp. 1–12.

[3] BERGER, E. D., YANG, T., LIU, T., AND NOVARK, G. Grace: safe multithreaded programming for c/c++. In *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications* (2009).

[4] BERGHEAUD, P., SUBHRAVETI, D., AND VERTES, M. Fault tolerance in multiprocessor systems via application cloning. In *ICDCS* (2007).

[5] BOOTHE, B. Efficient algorithms for bidirectional debugging. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)* (2000), ACM, pp. 299–310.

[6] DEVIETTI, J., LUCIA, B., CEZE, L., AND OSKIN, M. Dmp: deterministic shared memory multiprocessing. In *ASPLOS* (2009).

[7] DUNLAP, G. W., KING, S. T., CINAR, S., BASRAI, M. A., AND CHEN, P. M. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *OSDI* (2002).

[8] DUNLAP, G. W., LUCCHETTI, D. G., FETTERMAN, M. A., AND CHEN, P. M. Execution replay of multiprocessor virtual machines. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments* (New York, NY, USA, 2008), VEE '08, ACM, pp. 121–130.

[9] FELDMAN, S. I., AND BROWN, C. B. IGOR: a system for program debugging via reversible execution. *SIGPLAN Notices 24*, 1 (1989), 112–123.

[10] GDB TEAM. GDB and reverse debugging, 2009. `http://www.gnu.org/software/gdb/news/reversible.html`.

[11] GRISHMAN, R. The debugging system AIDS. In *AFIPS '70 (Spring): Proceedings of the 1970 Spring Joint Computer Conference* (New York, NY, USA, 1970), ACM, pp. 59–64.

[12] KING, S. T., DUNLAP, G. W., AND CHEN, P. M. Debugging operating systems with time-traveling virtual machines. In *Proc. of 2005 USENIX Annual Technical Conference, General Track* (2005), pp. 1–15.

[13] KUMAR, N., AND PERI, R. Transparent debugging of dynamically instrumented programs. *ACM SIGARCH Computer Architecture News 33*, 5 (Dec. 2005), 57–62. Special issue on the 2005 Workshop on Binary Instrumentation and Application (WBIA-05); PIN home page at `http://www.pintool.org`.

[14] LAADAN, O., VIENNOT, N., AND NIEH, J. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems* (New York, NY, USA, 2010), SIGMETRICS '10, ACM, pp. 155–166.

[15] LEE, D., SAID, M., NARAYANASAMY, S., YANG, Z., AND PEREIRA, C. Offline symbolic analysis for multi-processor execution replay. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture* (2009).

[16] LEFEBVRE, G., CULLY, B., FEELEY, M. J., HUTCHINSON, N. C., AND WARFIELD, A. Tralfamadore: Unifying source code and execution experience. In *EuroSys '09: Proceedings of the 4th ACM European Conference on Computer systems* (New York, NY, USA, 2009), ACM, pp. 199–204.

[17] LEROY, X., DOLIGEZ, D., GARRIGUE, J., RÉMY, D., AND VOUILLON, J. The Objective Caml system: release 3.11; documentation and user's manual, 2008. `http://caml.inria.fr/pub/docs/manual-ocaml/`.

[18] LEWIS, E. C., DHAMDHERE, P., AND CHEN, E. X. Virtual machine-based replay debugging, 30 October 2008. Google Tech Talks: `http://www.youtube.com/watch?v=RvMlihjqlhY`; further information at `http://www.replaydebugging.com`.

[19] LIU, T., CURTSINGER, C., AND BERGER, E. D. Dthreads: Efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)* (2011), pp. 327–336.

[20] MONTESINOS, P., HICKS, M., KING, S. T., AND TORRELLAS, J. Capo: a software-hardware interface for practical deterministic multiprocessor replay. In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2009), ASPLOS '09, ACM, pp. 73–84.

[21] NARAYANASAMY, S., POKAM, G., AND CALDER, B. Bugnet: Continuously recording program execution for deterministic replay debugging. In *Proceedings of the 32nd annual international symposium on Computer Architecture* (Washington, DC, USA, 2005), ISCA '05, IEEE Computer Society, pp. 284–295.

[22] OLSZEWSKI, M., ANSEL, J., AND AMARASINGHE, S. Kendo: Efficient deterministic multithreading in software. *SIGPLAN Notices: Proc. of Architectural Support for Programming Languages and Operating Systems (ASPLOS-09) 44*, 3 (2009), 97–108.

[23] PARK, S., ZHOU, Y., XIONG, W., YIN, Z., KAUSHIK, R., LEE, K. H., AND LU, S. Pres: probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009).

[24] PATIL, H., PEREIRA, C., STALLCUP, M., LUECK, G., AND COWNIE, J. Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization* (2010).

[25] POTHIER, . G., TANTER, E., AND PIQUER, J. Scalable omniscient debugging. In *Proc. 22nd ACM SIGPLAN Conf. Object-Oriented Programming Systems, Languages and Applications (OOPSLA 07)* (2007), ACM Press, pp. 535–552.

[26] RONSSE, M., AND DE BOSSCHERE, K. Recplay: a fully integrated practical record/replay system. *ACM Trans. Comput. Syst.* (1999).

[27] SAITO, Y. Jockey: a user-space library for record-replay debugging. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging* (New York, NY, USA, 2005), AADEBUG'05, ACM, pp. 69–76.

[28] SORIN, D. J., MARTIN, M. M. K., HILL, M. D., AND WOOD, D. A. SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *ISCA '02: Proceedings of the 29th annual International Symposium on Computer Architecture* (Washington, DC, USA, 2002), IEEE Computer Society, pp. 123–134.

[29] SRINIVASAN, S. M., KANDULA, S., ANDREWS, C. R., AND ZHOU, Y. Flashback: a lightweight extension for rollback and deterministic replay for software debugging. In *In Proceedings of the Annual Conference on USENIX Annual Technical Conference* (2004), pp. 29–44.

[30] THAIN, D., AND LIVNY, M. Multiple bypass: Interposition agents for distributed computing. *Cluster Computing 4*, 1 (2001), 39–47.

[31] TOLMACH, A. P., AND APPEL, A. W. Debugging Standard ML without reverse engineering. In *LFP '90: Proceedings of the 1990 ACM Conference on LISP and Functional Programming* (1990), ACM, pp. 1–12.

[32] TOTALVIEW TEAM. TotalView debugger, 2011. `http://www.roguewave.com/products/totalview-family/replayengine.aspx` (retrieved Jan., 2012).

[33] VISAN, A.-M., ARYA, K., COOPERMAN, G., AND DENNISTON, T. URDB: A universal reversible debugger based on decomposing debugging histories. In *Proc. of 6th Workshop on Programming Languages and Operating Systems (PLOS) (part of Proc. of 23rd ACM Symp. on Operating System Principles (SOSP))* (2011). electronic proceedings at `http://sigops.org/sosp/sosp11/workshops/plos/08-visan.pdf`; software for latest version, FReD (Fast Reversible Debugger), at `https://github.com/fred-dbg/fred`.

[34] WEERATUNGE, D., ZHANG, X., AND JAGANNATHAN, S. Analyzing multicore dumps to facilitate concurrency bug reproduction. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems* (2010).

[35] XU, M., BODIK, R., AND HILL, M. D. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2003), ISCA '03, ACM, pp. 122–135.

[36] ZAMFIR, C., AND CANDEA, G. Execution synthesis: a technique for automated software debugging. In *EuroSys* (2010).

[37] ZANDY, V. C., MILLER, B. P., AND LIVNY, M. Process hijacking. In *8th IEEE International Symposium on High Performance Distributed Computing* (1999), pp. 177–184.

[38] ZELKOWITZ, M. V. Reversible execution. *Communications of the ACM 16*, 9 (1973), 566.