# Semi-Automated Debugging via Binary Search through a Process Lifetime

Kapil Arya *
Northeastern University
kapil@ccs.neu.edu

Tyler Denniston*
MIT
tyler@csail.mit.edu

Ana-Maria Visan*
Google Inc.
amvisan@google.com

Gene Cooperman
Northeastern University
gene@ccs.neu.edu

## Abstract

A common programmer experience is to execute a long-running computation only to see a bug crash the program after hours or days. While it is often easy to capture a "buggy" expression value at the point of the crash, it is less easy to discover the point in the program where the expression became buggy. For such "difficult" bugs, this work presents an automated tool based on binary search through a process lifetime. The tool operates both in single-threaded and multi-threaded program. The underlying algorithm depends on on checkpoints, deterministic replay, and decomposition of debugging histories. The tool is scalable in the sense that the running time is a small constant factor beyond the standalone running time. Further, it requires only a logarithmic number of probes of the expression value — an advantage when the time to execute the expression is large. The algorithm is demonstrated for such real-world programs as MySQL.

## 1. Introduction

This work describes a new debugging technique, *reverse expression watchpoints*, designed to find bugs whose cause appears in the middle of a long-running program, but which only manifest themselves much later.

The goal here is the difficult bugs that do not show good *temporal locality*. The cause of the bug is far from when the bug manifests itself. The technique used is

*reverse expression watchpoints*,

which employ a binary search both backward and forward in the lifetime of the program.

The binary search through the program is based on maintaining a history of GDB debugging commands, and using checkpoint-based record-replay. The binary search is executed on the history of GDB commands. This requires expansion of a "continue" command into repeated "next", and "next" into repeated "step". Algorithms for doing this were previously described in our workshop paper [28].

The expansion of debugging commands [28] makes it possible to execute a variant of a binary search through the history of debugging commands. A key additional point of novelty is that the algorithm correctly handles *multithreaded programs*.

Although the emphasis of the algorithm is on multithreaded programs, we are not aware of a similar approach in the literature, even for the single-threaded case. The algorithm here is reminiscent of the "git-bisect" command for binary search through source code revisions. The infrastructure presented here also relates to Tralfamadore [15]. Tralfamadore presents a coarse execution trace, while allowing programmers to interactively refine their view of the execution, but it does not include any algorithm similar to the reverse expression watchpoints presented here.

The efficiency of the algorithm is best explained by noting that a binary search will require at most $\log_2 N$ probes (evaluations) of the expression over the process lifetime. The cost of running on the algorithm depends on the number of probes and the average number of times each statement is executed. Each statement is executed on average a small constant number of times. The number of probes is logarithmic in $N$, the number of statements executed by the program. For example, while a multi-core CPU can execute one billion statements per second, or $N = 8.64 \times 10^{13}$ statements in a day, the number of probes is only $\log_2 N \approx 46$.

***Outline of Paper.*** Section 2 describes reverse expression watchpoint and its implementation. Section 3 clarifies some limitations of this approach, and some alternatives. Section 4 provides an experimental evaluation. Section 5 describes the related work. Finally, the conclusion is in Section 6.

## 2. Reverse Expression Watchpoints

Figure 1 provides a simple example for motivating reverse expression watchpoints. 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, where $n$ is the length of the linked list. (A more sophisticated example might consider a bug in an otherwise duplicate-free linked list or an otherwise

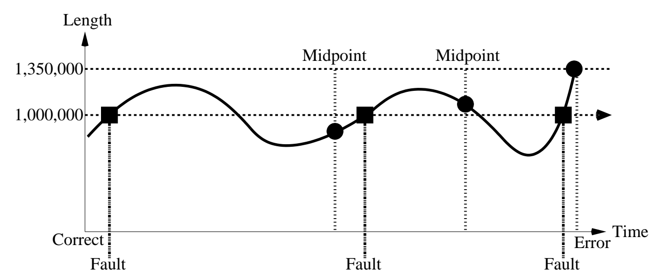


**Figure 1.** Reverse expression watchpoint for the bounded linked list example.

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

*PLOS'13*, November 03-06, 2013, Farmington, PA, USA.
Copyright © 2013 ACM 978-1-4503-2460-1/13/11. . . $15.00.
http://dx.doi.org/10.1145/2525528.2525533

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 1, 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.

Since the expression is "good" at the beginning of Figure 1 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, we have 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, we 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.

### 2.1 Pre-requisites

To implement reverse expression watchpoint, the following four systems are required:

1. A debugger;
2. A checkpoint-restart package that checkpoints the entire debugging session, including with the debugger;
3. Deterministic record-replay with memory accuracy; and
4. The ability to decompose a history of debugging commands (continue $\rightarrow$ next $\rightarrow$ step).

For the debugger, we use GDB. For checkpoint-restart, we use the transparent, user-space checkpointing package, DMTCP (Distributed MultiThreaded CheckPointing) [2]. A simple deterministic record-replay module is built to support this work, and an implementation is built based on the algorithms for decomposing debugging histories presented in [28].

The checkpoints of an entire debugging session (debugger and the target application) are taken at regular intervals. The history of 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.

***Decomposing debugging histories:*** Algorithms for decomposing debugging histories of commands were developed [28]. 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 [28]. 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 `step`.

### 2.2 The Simplified Algorithm

A reverse expression watchpoint is invoked by the programmer when the target process has stopped in GDB at an "error". The programmer uses GDB to determine an error condition that caused the program to stop. It may be as simple as "a pointer at this address has NULL value and was dereferenced". It may be more complex, as in the examples: "a linked list is too long" or "a representation of a dynamic graph is no longer connected".

The programmer specifies a Boolean expression associated with that error condition. The Boolean expression must be suitable for printing by GDB's "print" command. This Boolean expression is called the *watched expression*. The watched expression has one value (for example, "false") at the time of the error. At an earlier point in the program, before the error condition was met, the watched expression has the opposite Boolean value. We care only that the two Boolean values be opposite, and we will refer to the earlier Boolean value as "good" (no error), and the Boolean value at the error condition as "bad" (error observed).

The goal of a reverse expression watchpoint is to identify a transition of the watched expression from "good" to "bad". This is a point in the timeline at which the expression is "good", but at the next statement execution by a single thread, the expression becomes "bad".

Since the program execution begins at a statement for which the watched expression is "good", and it ends at an expression which is "bad", there must be at least one transition by a single statement from "good" to "bad". If there are multiple such transitions, the algorithm produces just one of those transitions. This is enough, since each such transition is associated with an occurrence of a bug.

The algorithm assumes a single checkpoint was taken at the very beginning of the debugging session. The programmer can take additional checkpoints at any time during the debugging session. Alternatively, additional checkpoints can be taken automatically at regular intervals.

The simplified algorithm employs just two steps: Step A and Step D. The missing Steps B and C will be described in Section 2.3. These will be added for reasons of efficiency and to better present the context of the bug to the programmer.

Step A searches for the bug at a coarse level, through a binary search of the checkpoint images. This finds an interval between consecutive checkpoint images during which the watched expression changes from "good" to "bad". Step D then identifies the exact thread and statement which caused the watched expression to change value.

- (A) *Search-Ckpts:* Binary search to find two successive checkpoint images evaluated as "good" and "bad". It can happen that all previous checkpoint images were "good". In this case, the desired checkpoint interval is from the most recent checkpoint image until the current point in time (when the watched expression must be "bad").
- (D) *Local-Search-With-Scheduler-Locking:* Replay the code with GDB's *scheduler-locking* parameter on. Switch deterministically in a round-robin fashion among the threads of the target application. Execute "step" commands in the active thread. If executing a "step" in the active thread causes the expression to transition from "good" to "bad", this must be the target thread and statement. Else, reaching the end of the interval or a deadlock, repeat the process in each other thread.

Step D (*Local-Search-With-Scheduler-Locking*) requires a more detailed explanation. Its purpose is to provide a deterministically replayable series of GDB commands that allows the end user to observe which thread caused the transition from "good" to "bad" occurred.

The algorithm of Step D 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.

Deterministically assigning blame to a culprit thread and statement is done by using GDB's *scheduler-locking* parameter. In this mode, a single "step" command causes just one thread to execute.

In this mode, the algorithm deterministically single steps through a single active thread. Because no background threads execute in this mode, there are only two possible outcomes. Either some statement execution by the active thread will cause a good-to-bad transition, or else some single step execution will block on a lock, causing deadlock. The deadlock is detected via a timeout of the single step. At this point, the last single step is "undone" (for example, by replaying from the previous checkpoint until the previous step), and another thread is then chosen as the active thread.

The actual details of Step D differ for reasons of efficiency:

(D.1) Select a thread as the "active" thread, and do repeated "next" commands to that thread (without scheduler locking) until the expression changes. Then determine if 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.

(D.2) Undo the last "next", and replace by a single "step" followed by repeated "next" (without scheduler locking). If the expression changes on that first step, go to step D.3 below. If the expression does not change, then go to step D.4.

(D.3) The expression changed on this "step". We must verify that it is due to the active thread. Undo "step", enable GDB scheduler locking, and redo the "step." If the expression changes, this is the right thread, and exit. If the expression does not change, or if deadlock ensues, then this is not the right thread. Go to step D.4.

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

The algorithm uses a timeout (currently 20 seconds) in order to decide if a deadlock occurred inside step (D.1) or step (D.3).

### 2.3 The Full Algorithm

The full algorithm inserts Steps B and C into the simplified algorithm of the previous section. Step B is essentially a simplified version of Step D. As in Step D, the purpose of Step B is to provide a series of GDB commands that the programmer can replay to observe the good-to-bad transition and determine the cause of the bug. Step B does not require scheduler-locking and suffices in the case of a single-threaded application. Even in the multi-threaded case, Step B brings the programmer closer to the bug, while postponing the need for the sometimes confusing use of scheduler-locking.

The purpose of Step C is greater efficiency. This is important when the portion of the program begin debugged has many system calls — and especially when there are many manipulations of locks. Without this, the use of scheduler-locking in Step D would incur many instances of deadlock, each one requiring us to restart and replay.

Step C consists of a binary search through the event log (similar to the binary search through the checkpoint images of Step A). The sequence of checkpoint images and the event log both provide a deterministic overview of a portion of the execution.

(A) *Search-Ckpts:* described in Step A of Section 2.2.

(B) *Search-Debug-History:* Step A identified a checkpoint interval, with a "good" checkpoint image, followed by a point in time with a "bad" watched expression. The "good" checkpoint image has associated with it a history of debugging commands until the following checkpoint image. Execute a binary search in the debug history between the "good" checkpoint image and the " bad" point in time. In the debugging history, expand GDB "continue" command into repeated "next" and "step" commands as needed to identify a transition from "good" to "bad" when a single GDB "step" command is executed. (Visan et al. [28] shows how to expand the GDB commands.)

(*) REMARK: In a single-threaded program, the algorithm stops at Step (B) above, with the desired transition. In a multi-threaded program, further work is needed. GDB may execute multiple threads in a single "step" command, the transition from "good" to "bad".

(C) *Search-Determ-Event-Log:* Binary search through the portion of the deterministic replay log corresponding to the last "step" command, as identified by Step B. Identify two consecutive events, such that the watched expression transitions from "good" to "bad" when replaying the events. [ Since multiple threads may have executed, multiple log events may have occurred. ] (Note that a background thread in the target application may be responsible for the transition of the watched expression to "bad". Since the background thread may not yet have been created, a binary search through the event log will guarantee that the execution progresses far enough to guarantee that the background thread exists, since thread creation is one of the events that is logged.)

(D) *Local-Search-With-Scheduler-Locking:* described in Step D of Section 2.2.

### 2.4 Correctness of the Algorithm

By default, the end user interactively creates checkpoint images at points of interest while executing within GDB. If a GDB "continue" command executes for a long time, the user may not be able to create a checkpoint during such a long period of time. To handle that case, this work supports the ability to transparently create intermediate checkpoints during the execution of a long-running "continue". This is particularly important in Step B, in which a "continue" command may be expanded into repeated "next" and "step" commands. The intermediate checkpoints ensure that one needs to search over only a moderate number of "next" and "step" GDB commands between checkpoints.

Note that the transition from "good" to "bad" may occur due to a background thread of the target application. This executes asynchronously with the primary thread (the current thread, responsible for executing the GDB commands). Hence, the transition from "good" to "bad" may be asynchronous with respect to the debug history. The algorithm makes two assumptions to account for this:

**1. Stability:** If a transition from "good" to "bad" is observed during the original record phase or during a replay phase, then during any replay phase, one will see a transition from "good" to "bad" within a reasonable time. (In cases of replaying a debug history, if the transition was caused by a background thread of the target application, the transition may occur only after the primary thread has replayed the entire debug history.)

In a binary search, at each iteration one must execute until a midpoint. Due to an asynchronous background thread, there is no guarantee that the watched expression will be deterministic after replaying the debug history until a midpoint. It could be "good" one time, and "bad" another time. The solution is to checkpoint

when an expression evaluates to “good”. This is the essence of a progress condition.

**2. Progress:** In binary search, assume that at the current iteration one replays from a checkpoint image that evaluates to “good”. One replays until the midpoint of the debug history under consideration. If an evaluation of the watched expression at the midpoint evaluates to “good”, then one checkpoints and makes that midpoint the left endpoint of the next iteration in the binary search. If an evaluation of the watched expression at the midpoint evaluates to “bad”, then one discards the second half of the debug history (the portion after the midpoint), and continues to the next iteration in the binary search. In each case, a progress condition guarantees eventual termination of the binary search with a “good” left endpoint, and a “bad” right endpoint, separated by a single GDB “step” command.

Note that while the stability condition and progress condition are described in terms of binary search over the debug history in Step B, the condition applies equally well to the binary search over the event log in Step C.

## 3. Analysis and Limitations

The size of a typical checkpoint image, for example for MySQL, is 60.5 MB. The event log requires 5.6 MB storage. Today’s terabyte disks allow one to store many checkpoint images. However, where storage is limited, an aging policy can be used to remove older checkpoint images. Only the last two checkpoint images and their associated event logs are needed for the algorithm of Section 2. This is useful in the case of such applications as MySQL, where the many locks incurred during initialization result in large event logs.

Because only the last two checkpoint images are required for the reverse expression watchpoint algorithm, it is tempting to use the “checkpoint” command of GDB version 7. This command essentially uses copy-on-write to fork a copy of the target process being debugged. However, GDB’s “checkpoint” command is valid only for single-threaded target processes. This appears to be related to the restriction that the Linux fork() system call preserves only one thread in the child process.

Another limitation of the current approach is for applications that employ shared memory variables that are unprotected by a lock. In this case, the event log fails to enforce determinism. Some programs use reads unprotected by a lock in an attempt to avoid the need for locks.

## 4. Experimental Evaluation

### 4.1 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 GDB version 7.3-0ubuntu. The kernel, glibc, gdb and gcc were unmodified. For DMTCP, we used svn revision 1956 and for FReD, we used git revision 62c7edd2.

First, the performance of reverse expression watchpoints is examined in Subsection 4.2 in a controlled benchmark program to demonstrate logarithmic runtime with exponential problem size growth.

Next, the reverse expression watchpoint feature was used to diagnose two real-world MySQL bugs (see Subsections 4.3 and 4.4), and one real-world pbzip2 bug (see Subsection 4.5). 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 1.2 million. The average size of an entry in the log was approximately 4.03 bytes.

### 4.2 Scaling with increasing computation time

This experiment is meant to represent a typical application that a developer might run on a desktop or laptop computer. It differs from the remaining sections, which are meant to show the use of reverse expression watchpoint to analyze bugs concerning large, real-world multi-threaded programs.

A C program was developed that adds edges to a large graph. The imagined application assumes that the graph will always be acyclic. In this scenario, the program is found to crash, and the developer must discover when in the program a cycle was created. Efficient algorithms for cycle detection exist based on implementing connecting components, but in this scenario the developer assumed this was not necessary, since cycles should never appear. A cycle has now appeared, and the developer wishes to find when this occurred.

For the initial graph, we used a large literature citation graph with 34,546 nodes and 421,578 edges from [10]. For multiples of this size, the graph was duplicated $N$ times with a single edge connecting each duplicate to maintain the connected property.

Three debugging options are available. One option is to run the program under GDB using its software watchpoint facility to check for a cycle upon executing each statement. Because a hardware watchpoint cannot be used here, this approach is too slow to finish in less than a day, and so is not represented in our results.

A second option is to add an assert statement into the program to check for the existence of a cycle after each edge is added. The third option is to use reverse expression watchpoint to find the point in program execution when a cycle was added. In both cases, the developer writes a small, straightforward function that does a depth-first search in testing for the existence of a cycle. This is invoked either in the assert statement or for reverse expression watchpoint.

| Graph Size | Runtime w/ Assert | Rev-Watch (s) | Ckpt (#) | Rstr (#) | Ckpt (s) | Rstr (s) | Eval Expr (s) |
|---|---|---|---|---|---|---|---|
| $N$ | 93.458 | 15.125 | 4 | 28 | 2.52 | 9.28 | 0.603 |
| $2N$ | 228.094 | 15.159 | 4 | 25 | 2.74 | 7.61 | 0.925 |
| $4N$ | 525.594 | 18.748 | 2 | 23 | 2.49 | 7.77 | 1.358 |
| $8N$ | 1058.033 | 19.143 | 3 | 21 | 4.50 | 7.30 | 2.751 |

**Table 1.** The time (in seconds) required to find the faulty statement that inserted a cycle in a directed acyclic graph. $N$ is the graph size with 34,546 nodes and 421,576 edges. $2N$, $4N$ and $8N$ represent graphs with a multiple of this number of nodes and edges.

The results for the latter two debugging strategies are summarized in Table 1. As shown, the runtime with assert statements grows approximately linearly with the problem size. However, due to binary search, the runtime with reverse expression watchpoints grows logarithmically with the problem size.

The number of checkpoints for larger graphs in this experiment is counterintuitively lower. Due to disk space constraints in the testing environment, the algorithm was adjusted to take fewer checkpoints as the size of the graph increased.

### 4.3 MySQL Bug 12228 — Atomicity Violation

In order to reproduce MySQL bug 12228, a stress test scenario was set in which ten 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 5.0.10.

The buggy thread interleaving and the series of requests issued by each client are presented in Figure 2. 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 in the following way: the user runs the MySQL daemon under this system and executes the stress test scenario presented in Figure 2. 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) checkpoint
(gdb) continue
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) reverse-watch *(0x1e214a0) == 1702125600
'reverse-watch' took 406.24 seconds.
(gdb) list
344     memcpy(pos,str,len);
```

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? Reverse expression watchpoint (or `reverse-watch` as abbreviated above) is used to answer this question. In the case of this bug, an unchecked `memcpy()` call was overwriting the region of memory containing the `rt` pointer, leading to the `SIGSEGV`.

The time for reverse expression watchpoint, as well as other useful information, are shown in Table 2.

### 4.4 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

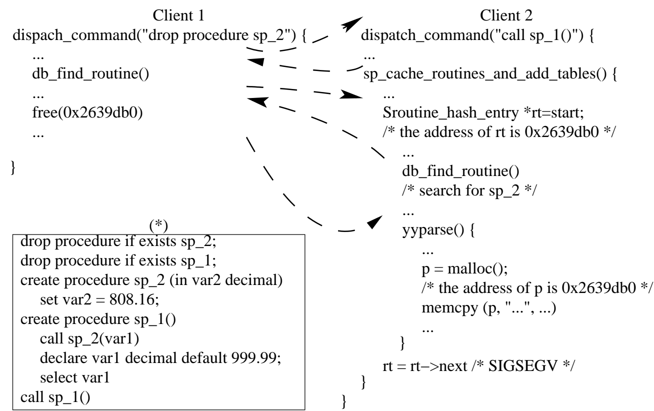


**Figure 2.** MySQL Bug 12228: the thread interleaving that causes the MySQL daemon to crash with `SIGSEGV`; (*) the sequence of instructions executed by each thread, in pseudo-SQL

| Bug Number | Rev-Watch (s) | Ckpt (#) | Rstr (#) | Ckpt (s) | Rstr (s) | Eval expr (s) |
|---|---|---|---|---|---|---|
| MySQL 12228 | 406.24 | 4 | 60 | 3.45 | 24.49 | 1.69 |
| MySQL 42419 | 161.68 | 6 | 55 | 6.17 | 22.59 | 1.06 |
| pbzip2 | 29.22 | 1 | 17 | 0.99 | 5.60 | 0.41 |

**Table 2.** The bugs and the time it took to diagnose them, by performing reverse expression watchpoint (in seconds). The number of checkpoints and restarts and the total times for checkpoint, restart and evaluation of the expression (in seconds) are also shown.

indicated in the bug report. 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) 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) 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 reasonable 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 the incorrect value (i.e. the current value). This happens during a call to the function `make_join_statistics():sql_select.cc:5295` at 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 2.

### 4.5 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. 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 2. The debugging session is presented below:

```
(gdb) break pbzip2.cpp:1018
```

```
(gdb) run
Breakpoint 1, at pbzip2.cpp:1018.
(gdb) 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) reverse-watch *0x60cfe0 == 0
```

## 5. Related Work

### 5.1 Reverse Expression Watchpoint

Both IGOR [9] and the work by Boothe [5] support a primitive type of reverse expression watchpoint for monotonically varying single variables only. It is detected when the variable value exceeds a threshold, but there is no support for decomposing histories. The work of King et al. [11] is similar, but limited to going back to the last time a variable was modified, by employing virtual machine snapshots and event logging. Whyline [12] allows the programmer to ask "why" or "why not" questions about program execution, but does not allow specification of general expressions. The authors also state it is unintended for debugging program executions exceeding several minutes.

The idea of a general reverse expression watchpoint was first developed by the authors in a technical report [27]. Precise algorithms for decomposing debugging histories (continue → next → step) were then developed to produce a robust algorithm [28].

### 5.2 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, 7–9, 13, 14, 16, 18, 19, 21, 22, 24–26, 29–31]. There are also many systems whose goal is to make the initial execution deterministic [3, 6, 17, 20, 23].

## 6. Conclusion

A reverse expression watchpoint algorithm has been presented for automating a binary search through a process lifetime. The end user must determine an expression that is associated with the bug being diagnosed. Using this expression, the system places the user in GDB just when the expression indicates a bug.

## References


[1] G. Altekar and I. Stoica. ODR: Output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, 2009.

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

[3] E. D. Berger, T. Yang, T. Liu, and G. Novark. 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] P. Bergheaud, D. Subhraveti, and M. Vertes. Fault tolerance in multiprocessor systems via application cloning. In *ICDCS*, 2007.

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

[6] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. Dmp: deterministic shared memory multiprocessing. In *ASPLOS*, 2009.

[7] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *OSDI*, 2002.

[8] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay of multiprocessor virtual machines. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '08, pages 121–130, New York, NY, USA, 2008. ACM.

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

[10] J. Gehrke, P. Ginsparg, and J. Kleinberg. Overview of the 2003 KDD cup. *SIGKDD Explorations*, 5(2):149–151, 2003.

[11] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proc. of 2005 USENIX Annual Technical Conference, General Track*, pages 1–15, 2005.

[12] A. Ko and B. Myers. Debugging reinvented: asking and answering why and why not questions about program behavior. *ICSE*, 2008.

[13] O. Laadan, N. Viennot, and J. Nieh. 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*, SIGMETRICS '10, pages 155–166, New York, NY, USA, 2010. ACM.

[14] D. Lee, M. Said, S. Narayanasamy, Z. Yang, and C. Pereira. Offline symbolic analysis for multi-processor execution replay. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009.

[15] G. Lefebvre, B. Cully, M. J. Feeley, N. C. Hutchinson, and A. Warfield. Tralfamadore: Unifying source code and execution experience. In *EuroSys '09: Proceedings of the 4th ACM European Conference on Computer systems*, pages 199–204, New York, NY, USA, 2009. ACM.

[16] E. C. Lewis, P. Dhamdhere, and E. X. Chen. 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`.

[17] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: Efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 327–336, 2011.

[18] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. 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*, ASPLOS '09, pages 73–84, New York, NY, USA, 2009. ACM.

[19] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, ISCA '05, pages 284–295, Washington, DC, USA, 2005. IEEE Computer Society.

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

[21] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. Pres: probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009.

[22] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. 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.

[23] M. Ronsse and K. De Bosschere. Recplay: a fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 1999.

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

[25] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *ISCA '02: Proceedings of the 29$^{th}$ annual International Symposium on Computer Architecture*, pages 123–134, Washington, DC, USA, 2002. IEEE Computer Society.

[26] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: a lightweight extension for rollback and deterministic replay for software debugging. In *In Proceedings of the Annual Conference on USENIX Annual Technical Conference*, pages 29–44, 2004.

[27] A. M. Visan, A. Polyakov, P. S. Solanki, K. Arya, T. Denniston, and G. Cooperman. Temporal debugging using URDB. arXiv:0910.5046v1 [cs.OS], `http://arxiv.org/abs/0910.5046`; software at `http://urdb.sourceforge.net`, 2009.

[28] A.-M. Visan, K. Arya, G. Cooperman, and T. Denniston. 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`;.

[29] D. Weeratunge, X. Zhang, and S. Jagannathan. 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.

[30] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ISCA '03, pages 122–135, New York, NY, USA, 2003. ACM.

[31] C. Zamfir and G. Candea. Execution synthesis: a technique for automated software debugging. In *EuroSys*, 2010.