URDB: A Universal Reversible Debugger Based on Decomposing Debugging Histories

Ana-Maria Visan*, Kapil Arya*, Gene Cooperman*, Tyler Denniston†
Northeastern University, Boston, MA, USA
{amvisan,kapil,gene,tyler}@ccs.neu.edu

ABSTRACT

Reversible debuggers have existed since the early 1970s. A novel approach, URDB, is introduced based on checkpoint/re-execute. It adds reversibility to a debugger, while still placing the end user within the familiar environment of their preferred debugger. The URDB software layer currently includes modes that understand the syntax for four debuggers: GDB for C/C++/Java/Fortran, Python (pdb), MATLAB, and Perl (perl -d). It does so by adding a thin URDB software layer on top of the DMTCP checkpoint-restart package. URDB passes native debugging commands between the end user and the underlying debugging session. URDB models the four common debugging primitives that form the basis for most debuggers: `step`, `next`, `continue`, `break`. For example, given a debugging history of the form [`step`, `next`, `step`], URDB's `reverse-step` produces a new history, [`step`, `next`]. Further, subtle algorithms are described for `reverse-xxx`. For example, `reverse-step` operates correctly when the last instruction of the history is `next` or `continue`.

URDB calls DMTCP to create a checkpoint during a debugging session, and then replays the history from there. An essential novelty of this work is the extension of DMTCP to be the first checkpointing package capable of checkpointing a GDB sesssion to disk (through checkpointing the Linux `ptrace` system call). This was a significant barrier to earlier attempts toward checkpoint/re-execute for GDB. Support for the GDB debugger is important to any reversible debugger claiming universality. Experimental results are described for GDB, MATLAB, Python (pdb), and Perl.

1. INTRODUCTION

URDB, a universal reversible debugger based on checkpoint, restart and re-execute is presented. A history of debugging commands since the last checkpoint is maintained.

*This work was partially supported by the National Science Foundation under Grant OCI 09-60978.

†This work was partially supported by the National Science Foundation under Grant CCF 09-16133.

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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PLOS '11, October 23, 2011, Cascais, Portugal.
Copyright 2011 ACM 978-1-4503-0979-0/11/10 ...$10.00.

This yields a simple mechanism for reversibility: re-execution of that history. Hence, if `reverse-step` operates on a history [`next`, `step`], it produces a new history, [`next`]. Replaying the new history places the user at proper point in time. In this simple example, `reverse-step` is equivalent to an `undo-command`. However, when a reverse primitive, `reverse-xxx`, follows a primitive other than `xxx`, then a more sophisticated decomposition algorithm is required (see Section 4).

URDB currently adds reversibilty for four debuggers: GDB for C/C++/Java/Fortran, Python (pdb), MATLAB, and Perl (perl -d). The methodology employs external checkpointing of unmodified binaries. In particular, MATLAB is closed source, and so any other methodology would face major barriers. Reversibility can be added to a new debugger in less than a day.

This work also describes an advance in the state of the art for checkpointing: the checkpointing package DMTCP (Distributed MultiThreaded CheckPointing) [1] has been enhanced as DMTCP/ptrace, to support transparent checkpointing of GDB sessions under Linux. To the best of our knowledge, DMTCP/ptrace is the first checkpointing package able to checkpoint a GDB debugging session.

Hence, the novelty of URDB lies in:

1. universality, placing the user within the framework of a familiar debugger (GDB/Python/MATLAB/Perl) *(Reversibility can be added in less than a day.)*;
2. the decomposition algorithms for manipulating a history of debugging primitives; and
3. transparent checkpointing of GDB sessions in DMTCP.

While an earlier technical report [16] covers these three points, this work describes an improved algorithm for the decomposition, and also provides some implementation details for checkpointing of GDB through `ptrace` support.

Past reversible debuggers have been based either on logging of instructions (at the assembly or source level), or else on post-mortem debuggers that create a "temporal database" of the state of the running process throughout its history. For example, in our tests on GDB's "target record", GDB stored 104 bytes of information per C statement executed. Note that:

- the two primary approaches above require comparatively orders of magnitude greater amounts of storage, with implications for the length of a debugging session.

Previous implementations of the checkpoint/re-execute approach also exist [2, 3, 12, 8], but did not capture either

URDB's transparency (no modification to target binary) or URDB's universality.

Outline of Paper.

Section 2 describes the architecture of URDB, as well as its support for universality. Section 3 describes the implementation of DMTCP/ptrace for checkpointing of GDB sessions. Section 4 describes the reversibility algorithms, using checkpointing and decomposition of debugging histories. Section 5 presents the experimental results. Section 6 presents a brief history of reversible debugging. Finally, Section 7 presents the conclusion.

2. URDB

URDB sits between the end user and the target debugger (see Figure 1). For the most part, URDB passes user commands to the target debugger and returns the debugger output (including interrupts (ctrl-C) by the user). When a checkpoint or restart is requested, URDB passes the command to the checkpointing package.

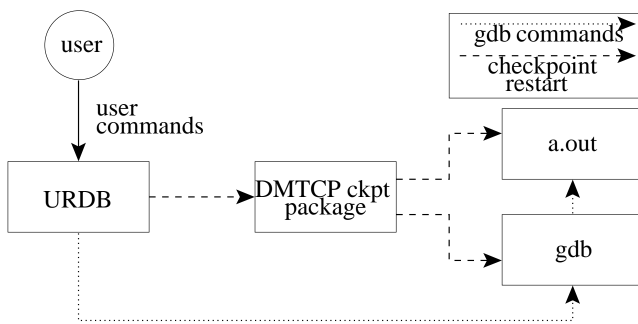

Figure 1: The Architecture.

For the sake of exposition, we describe the reverse commands in the special case that only one checkpoint is taken during the life of the process. (Section 4.5 presents the algorithms in the presence of multiple checkpoints.) A reverse command will decompose the history of debugging commands, generate a modified history corresponding to the result of the reverse command, and then restart from the checkpoint, and re-execute the modified history.

For efficiency, the re-execution coalesces debugger commands such as `next` and `step`. For example, a debugger command [`next`, `next`, `next`] is replaced by [`next 3`]. The existence of interim debugger breakpoints in the middle of a coalesced debugger command adds a subtle point. Such interim breakpoints are temporarily disabled as part of a coalescing of `next` or `step`.

In terms of size, URDB has approximately 3,000 lines of Python code. Further, each debugger-specific personality file is roughly 150 lines of Python code: 100 lines for MATLAB; 90 lines for Python/pdb; 130 lines for Perl (perl -d); and 220 lines for GDB. The DMTCP support for `ptrace` (GDB) adds approximately 2,000 lines of C/C++.

3. DMTCP/PTRACE

The original DMTCP package was extended as part of this work to checkpoint `ptrace` and GDB. We refer to this extension as DMTCP/ptrace. The `ptrace` Linux system call allows a superior process (e.g. GDB) to trace an inferior process (target application) at the binary level.

The inferior process must stop tracing the superior process at the time of a checkpoint. To understand why, note that the inferior process is normally being traced by a user's thread in the superior process. But during a checkpoint, the DMTCP checkpoint thread has control, and not the user's thread. DMTCP then arranges for the user's thread to resume tracing the inferior process at the time of resuming (after checkpoint) or restarting (from a checkpoint fie).

Some noteworthy issues in developing DMTCP/ptrace are presented next.

1. *eflags register.* `ptrace` is based on the eflags hardware register of the x86 architecture. Once the superior process starts tracing the inferior process, the eflags trace bit is set for the inferior process. At restart time, the eflags trace bit was no longer set, causing the inferior process to run away.
2. *Inside DMTCP code at restart time.* Upon restart by DMTCP, both the superior and inferior processes begin life inside DMTCP's own signal handler. (At checkpoint time, DMTCP had quiesced the user process by forcing it into a DMTCP signal handler.) The superior process needs to single-step the inferior process out of the signal handler into user code. Single-stepping is required to detect the exit from the signal handler.
3. *Tid virtualization.* GDB sends a `ptrace` command to a specific inferior, identified through a unique thread id. Upon restart, each thread is given a new thread id by the operating system. So, a thin virtualization of the relevant system calls was needed in order to translate between the original thread id and current thread id.

4. REVERSIBLE ALGORITHMS

We take the "universal" debugging primitives to be `step` (step into a function call), `next` (do not step into any function calls) and `continue` (until next breakpoint). Conceptually, it is useful to consider a fourth debugging primitive, `next/bkpt` (`next` interrupted by breakpoint). (Note that `step` and `continue` do not have special analogs. `step` can never be interrupted by hitting a breakpoint, and `continue` is always interrupted by hitting a breakpoint.)

The corresponding reversible commands are `reverse-next` (go to previous statement of function, or to caller if at beginning of function), `reverse-step` (same as reverse-next, except that if the previous statement was a function call, then step backwards into the last statement of the function), and `reverse-continue` (go to last breakpoint encountered). In implementation, this causes the trimmed history to be re-executed from the last checkpoint. Figure 2 illustrates these commands. If the debugger implements the `finish` primitive (until end of function), then `reverse-finish` is supported (return to the statement of the function that called the current function). An additional reversible command, `undo-command` takes the current history and removes the last debugging command.

Three utility functions in the algorithms of this section are defined here: *deeper()* returns true if the current stack depth is deeper than the original stack depth just before beginning the `reverse-xxx` command, and false otherwise; *shallower()*

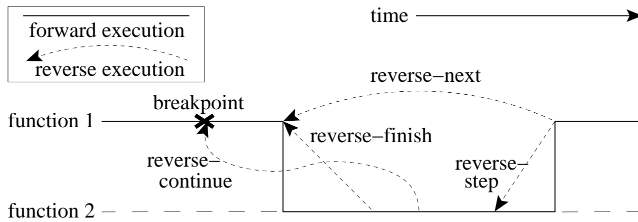

Figure 2: The reverse analogs of standard debugging commands.

returns true if the current stack depth is shallower than the original stack depth; *same()* returns true if the current stack depth is equal to the original stack depth.

In the pseudo-code, there is always an implied history of debugger commands since the last checkpoint. The algorithms take that history as input, and produce a new history as output. For example, `reverse-step`([*next*, *step*]) → [*next*]. The pseudo-code will often refer to the *last command* to mean the last debugging command in the current history. A statement such as "execute `next`" implies that the `next` command is also appended to the history.

The reversible debugging algorithms work across multiple checkpoints. Each checkpoint has an associated history of debugging commands, denoted *history*. One can not reverse past the first checkpoint, since there is no earlier history. But if needed, a checkpoint can be taken right at the beginning of the program, allowing one to reverse throughout the entire lifetime of the process.

4.1 Reverse-next

The `reverse-next` command is described in Algorithm 1. It replaces the history since the last checkpoint by a new, modified history. This brings the program to the same execution point as if the `reverse-next` command of Figure 2 were literally executed backwards in time. Conceptually, there are two classes of commands to be analyzed: `continue` and `next/bkpt`; and `step` and `next`. For motivation, note that a `next/bkpt` command has the same result as a `continue` command. Both commands terminate at a breakpoint. Lines 12 – 16 describe the processing for a `continue` command.

A final `continue` command will expand into a `step` followed by repeated `next` commands until the breakpoint is reached, and further iterations may return to these same lines in order to again replace a final `next/bkpt` by a `step` and repeated `next` commands.

The remaining pseudo-code is meant to handle the simpler cases of `step` and `next`, and two special cases for `next/bkpt`. The handling of `next/bkpt` at line 7 is essentially the same case as at line 21. In this situation of exiting a function, it is irrelevant that the command ends at a breakpoint. The `reverse-finish` at lines 9 and 23 has the purpose of maintaining the original stack depth.

4.2 Reverse-step

Reverse-step is conceptually simpler than `reverse-next`. As before, if the last command was `continue`, it is expanded into `step` and repeated `next` commands (lines 11 – 15) ensure that if the last command is `next`, then it will be decomposed into a `step` (possibly causing the stack to grow deeper), followed by repeated `next` commands until the original stack depth (*same*()) is satisfied. If the new last command is again `next`, then the loop will be repeated until a final `step` command is generated. The final `step` command can then be stripped at lines 7 – 9 in order to honor the semantics of `reverse-step`.

A similar analysis applies to lines 2 – 6. However, in this case, the repeated `next` commands terminate at a breakpoint instead of when one reaches the original stack depth.

```
 1: while true do
 2:     if last command is continue or next/bkpt then
 3:         set cmd ← last command
 4:         execute undo-command
 5:         if cmd is next/bkpt and same() then
 6:             break
 7:         else if cmd is next/bkpt and deeper() then
 8:             {next/bkpt had exited a function}
 9:             execute reverse-finish
10:             break
11:         else
12:             {else shallower() or cmd is continue}
13:             execute step
14:             while we are not at breakpoint do
15:                 execute next
16:             {go to to next iter of while loop}
17:     else if last command is step or next then
18:         execute undo-command
19:         if same() or shallower() then
20:             break
21:         else if deeper() then
22:             {next had exited a function}
23:             execute reverse-finish
24:             break
```

Algorithm 1: Reverse-next

```
 1: while true do
 2:     if last command is continue or next/bkpt then
 3:         execute undo-command
 4:         execute step
 5:         while we are not at breakpoint do
 6:             execute next
 7:     else if last command is step then
 8:         execute undo-command
 9:         break
10:     else
11:         {last command is next}
12:         execute undo-command
13:         execute step
14:         while deeper() do
15:             execute next
```

Algorithm 2: Reverse-step

4.3 Reverse-continue

For each of the four primitive debugging commands in the history, we add a Boolean attribute `at_bkpt` for the sake of the `reverse-continue` command. The attribute indicates whether the program was at a breakpoint at the end of that command. In particular, note that the attribute will always be true for `continue` and for `next/bkpt`. The attribute will always be false for `next`. The attribute may have either

value for `step`. This yields the relatively short pseudo-code of Algorithm 3.

```
1: repeat
2:     execute undo-command
3: until we are at a breakpoint
4: {NOTE: An optimization can scan the history and
   replay it until the last bkpt before the current stmt}
```

Algorithm 3: Reverse-continue

4.4 Reverse-finish

The `reverse-finish` algorithm follows a logic similar in spirit to that of `reverse-next` and `reverse-step`. It can be implemented by executing a sequence of `reverse-next`'s and checking the stack depth after each `reverse-next`. While this makes `reverse-finish` and `reverse-next` mutually recursive, the algorithms can easily be shown to terminate.

4.5 Multiple checkpoints

There is an obvious extension of the preceding algorithms to handle multiple checkpoints. Multiple checkpoints are useful as an optimization to reduce the cost of replaying histories. Each checkpoint has associated with it a history that continues until the next checkpoint. If an algorithm executes an `undo-command` when the history is currently empty, then one reverts to the earlier checkpoint and its associated history. In the future, taking extra checkpoints will be automated.

5. EXPERIMENTAL RESULTS

Two types of experiments are presented: the performance of URDB as applied to each of four common debuggers; and timing comparisons with the gdb-7.2 reversible debugger. All experiments were performed on a quad-core AMD Opteron 8346 HE CPU with 2 MB of L2 and L3 cache.

Experiments on URDB across Debuggers.

The times to execute command `reverse-xxx` were measured on a program that inserts twenty numbers into a linked list. Each insertion is done by making a function call. This program was ported to C, MATLAB, Python and Perl.

The timings are presented in Table 1. For both `reverse-next` and `reverse-step`, two breakpoints were added: one at the main function and another one after the insertion of all twenty elements into the list. A checkpoint is taken at the main function. `reverse-next` and `reverse-step` are issued at the second breakpoint.

For `reverse-continue`, one breakpoint was added at main, one inside the function that inserts an element into the linked list and another one after the insertion of all twenty elements. The `reverse-continue` was issued at the last breakpoint. To test `reverse-finish` we make use of the first two breakpoints mentioned above. Once the second breakpoint was hit, a `reverse-finish` command was issued.

The timings for checkpoint-restart across the four debuggers are presented in Table 2. The primary conclusion is that a checkpoint/re-execute strategy for reversibility using DMTCP is fast enough for interactive use in a reversible debugger.

Command	gdb-7.2	MATLAB	Perl	Python
reverse-next	20.44s	21.61s	16.75s	12.93s
reverse-step	22.14s	18.40s	16.42s	12.80s
reverse-continue	7.78s	7.43s	5.77s	5.62s
reverse-finish	3.67s	1.86s	0.88s	0.78s

Table 1: URDB: Times for reverse-next, reverse-step, reverse-continue, and reverse-finish in seconds.

Command	gdb-7.2	MATLAB	Perl	Python
checkpoint	1.86s	2.02s	0.17s	0.18s
restart	1.20s	1.65s	0.20s	0.17s

Table 2: URDB: Times for checkpoint/restart in seconds.

Timing comparison with reversibility in gdb-7.2.

It was decided to compare the timings of URDB with gdb-7.2, since that reversible debugger is readily available as a timing benchmark. The `gdb-7.2` debugger [4] achieves reversibility by logging each assembly level instruction.

For testing, a C program was written to create a linked list with 1,000,000 elements. The program allocated its own memory and avoided the use of C malloc, to model a purely CPU-intensive program. The times measured in running the program in the forward direction were 0.063 s (native C program), 0.144 s (C under gdb-7.2 under URDB), and 1,440.27 s (C under gdb-7.2 using `target record` mode for reversibility). Hence, URDB was 5,200 times faster than the target record mode of gdb-7.2. In both cases, a reverse instruction runs with reasonable interactive time. For gdb-7.2, the reverse time depends on the number of reverse steps executed, while for URDB, the reverse time depends on the number of forward instructions from the last checkpoint. In URDB, intermediate checkpoints can be taken for higher reverse performance. Gdb-7.2 also incurs a memory consumption per instruction in target record mode. For our C program, this was measured at 104 bytes per C statement.

6. RELATED WORK

A brief discussion of the current four different approaches to building a reversible debugger is presented next. URDB is an example of checkpoint/re-execute. Unlike other examples, URDB checkpoints to disk, thus allowing it to manage the many checkpoints needed by a long-running program.

Record/Reverse-execute.

The record phase logs the state of each instruction as it is executed. In addition to logging instructions, one can log external I/O, signals, and other events, for better replay. On replay, the information from the log is used. As an example, on record, an assembly store instruction will cause the previous value in memory to be saved in the log entry. On reverse-execute, the old value in memory is restored.

While the benefits are clear, there are also significant disadvantages. The need for logging instructions means that the debugger executes at less than near native speed. Further, the size of the log files can be significant.

Among the reversible debuggers implementing the record/ reverse-execute approach are: the AIDS debugger [5] for FORTRAN, Zelkowitz [17] for PL/I, the work of Appel and Tolmach [13, 14] for Standard ML and more recently, gdb-

7.2 [4]. Another example is TotalView [15], a proprietary debugger that provides reversibility by saving the state information as the program executes. The saved state information is the program's execution history, which is limited to several megabytes. Also, TotalView's re-execute phase creates up to thirty extra processes.

Record/Replay.

This approach employs virtual machine snapshots and event logging. It was demonstrated in the work of King et al. [6] and Lewis et al. [9]. Snapshots record the state of the machines at given intervals. A reproducible clock is achieved through values of certain CPU registers, such as the number of loads and stores since startup. This allows asynchronous events to be replayed according to the time of the original clock when they occurred.

Snapshots have the advantage that forward execution can be extremely fast, running much closer to full native speed, especially if there are few events to log. This approach does not need to log each instruction, but only external events. The main disadvantages of this approach are: the cost of virtual machine snapshots is high (about 30 seconds per snapshot and GigaBytes footprints) and it does not extend to SMP multi-core hardware.

Checkpoint/Re-execute.

This approach typically uses *live checkpoints* as the checkpointing strategy. During live checkpointing, checkpoints are created by forking off a child process of the debugged application, such as IGOR [3] (modified compiler and kernel, and special interpreter during re-execution), Flashback [12] (via an OS extension), ocamldebugger [8, Part III, Chapter 16] and the work by Boothe [2] (modified compiler). Boothe [2] uses event counters that are added during compilation at each statement, as well as the entry and exit point of each function. This leads to high overhead. Another limitation is the number of live checkpoints that can be maintained at any given time.

Post-mortem debugging.

In this approach, a database on disk is created that logs all events of interest. Debugging is then done after the process of interest has terminated. Only the database of events is required. This approach was demonstrated by Omniscient Debugger [10, 11] and Tralfamadore [7]. The Omniscient Debugger is limited by the speed of disk, with 89 megabytes per second being generated. Tralfamadore represents a somewhat different approach to post-mortem debugging, by unifying an execution trace with the source code itself. One begins by examining the code at a high level, looking, for example, for frequent paths through the code.

7. CONCLUSION

URDB is a universal reversible debugger. Reversibility for a new debugger can be added in less than a day. URDB uses multiple checkpoints, with re-execute via a history of debugging commands. Two enabling techniques were developed for this purpose: decomposition of debugging commands, and checkpointing of GDB sessions. A representation of standard debugging commands is the key to universality. URDB has been demonstrated on four widely different debuggers: GDB, Python (pdb), MATLAB and Perl (perl -d). In the example of GDB, URDB was demonstrated to run 5,200 times faster than the GDB reversibility mode (using "record target").

Previous reversible debuggers have emphasized either logging of instructions or databases for post-mortem analysis. Boothe [2] represents a limited version of the DMTCP approach (a modified compiler inserting an event counter for each statement; a custom debugger; and the use of live checkpoints through fork() (one process per checkpoint)). URDB presents a fully transparent approach that adapts to each existing debugger.

8. REFERENCES

[1] J. Ansel, K. Arya, and G. Cooperman. DMTCP: Transparent checkpointing for cluster computations and the desktop, 2009.

[2] B. Boothe. Efficient algorithms for bidirectional debugging. In *PLDI*, 2000.

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

[4] GDB team. GDB and reverse debugging. `http://www.gnu.org/software/gdb/news/reversible.html`.

[5] R. Grishman. The debugging system AIDS. In *AFIPS*, 1970.

[6] S. T. King, G. W. Dunlap, and P. M. C. V. Corporation). Debugging operating systems with time-traveling virtual machines. In *USENIX*, 2005.

[7] G. Lefebvre, B. Cully, M. J. Feeley, N. C. Hutchinson, and A. Warfield. Tralfamadore: unifying source code and execution experience. In *EuroSys*, 2009.

[8] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml system: release 3.11; documentation and user's manual, 2008. `http://caml.inria.fr/pub/docs/manual-ocaml/`.

[9] E. C. Lewis, P. Dhamdhere, and E. X. Chen. Virtual machine-based replay debugging, 2008. Google Tech Talks: `http://www.youtube.com/watch?v=RvMlihjqlhY`.

[10] G. Pothier and E. Tanter. Back to the future: Omniscient debugging. *Software*, 26(6):78–85, 2009.

[11] G. Pothier, E. Tanter, and J. Piquer. Scalable omniscient debugging. In *OOPSLA*, 2007.

[12] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: a lightweight extension for rollback and deterministic replay for software debugging. In *USENIX ATC*, 2004.

[13] A. P. Tolmach and A. W. Appel. Debugging Standard ML without reverse engineering. In *LFP '90*, 1990.

[14] A. P. Tolmach and A. W. Appel. A debugger for Standard ML. *J. Funct. Program.*, 5(2):155–200, 1995.

[15] TotalView team. TotalView debugger. `http://www.roguewave.com/products/totalview-family/totalview.aspx`.

[16] 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`, 2009.

[17] M. V. Zelkowitz. Reversible execution. *Communications of the ACM*, 16(9):566, 1973.