

URDB: A Universal Reversible Debugger Based on Decomposing Debugging Histories

Ana-Maria Visan Kapil Arya Gene Cooperman
Tyler Denniston

Northeastern University
{amvisan,kapil,gene,tyler}@ccs.neu.edu

October 23, 2011

Your Typical Debugging Session

Launch program → Fault → Bug

Your Typical Debugging Session

Launch program → **Fault** → Bug

Your Typical Debugging Session

Launch program → Fault → Bug

Your Typical Debugging Session

Launch program → Fault → Bug

Why is it so frustrating and time consuming?

- ▶ Analyze the bug near the error
- ▶ Produce a hypothesis about the cause of the bug
- ▶ Restart the program from the beginning

Your Typical Debugging Session

Launch program \longrightarrow Fault \longrightarrow Bug

Why is it so frustrating and time consuming?

- ▶ Analyze the bug near the error
- ▶ Produce a hypothesis about the cause of the bug
- ▶ Restart the program from the beginning

Wouldn't it be great to be able to go backwards?

Introducing URDB

URDB is a universal reversible debugger based on:

- ▶ checkpoint
- ▶ restart
- ▶ re-execution of debugging histories
- ▶ decomposing of debugging histories

Introducing URDB

URDB is a universal reversible debugger based on:

- ▶ **checkpoint**
- ▶ restart
- ▶ re-execution of debugging histories
- ▶ decomposing of debugging histories

Introducing URDB

URDB is a universal reversible debugger based on:

- ▶ checkpoint
- ▶ restart
- ▶ re-execution of debugging histories
- ▶ decomposing of debugging histories

Introducing URDB

URDB is a universal reversible debugger based on:

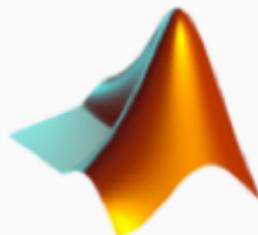
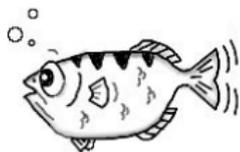
- ▶ checkpoint
- ▶ restart
- ▶ re-execution of debugging histories
- ▶ decomposing of debugging histories

Introducing URDB

URDB is a universal reversible debugger based on:

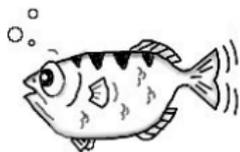
- ▶ checkpoint
- ▶ restart
- ▶ re-execution of debugging histories
- ▶ decomposing of debugging histories

URDB: The Novelty



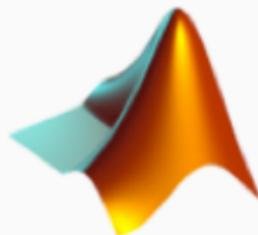
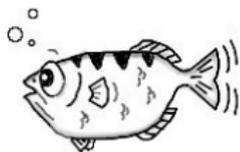
- ▶ Universality, placing the user within the framework of a familiar debugger (GDB/Python/MATLAB/Perl)
(Reversibility can be added in less than a day)

URDB: The Novelty



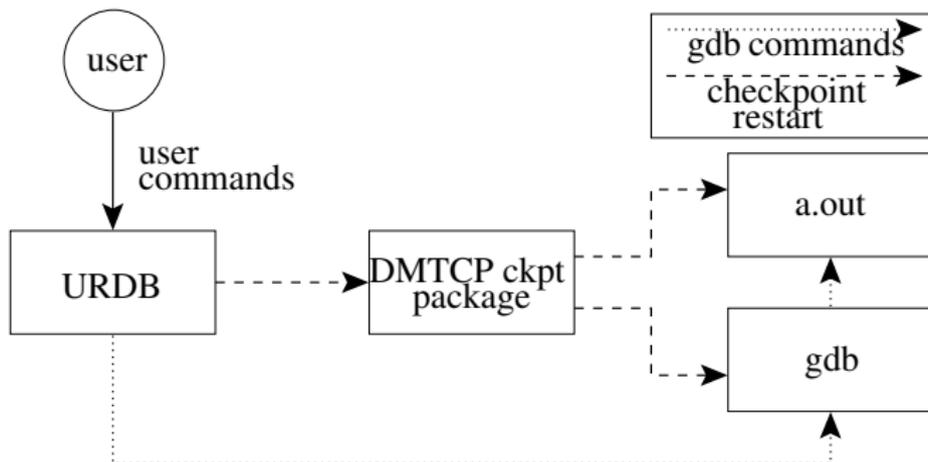
- ▶ Universality, placing the user within the framework of a familiar debugger (GDB/Python/MATLAB/Perl)
(Reversibility can be added in less than a day)
- ▶ The decomposition algorithms for manipulating a history of debugging primitives

URDB: The Novelty



- ▶ Universality, placing the user within the framework of a familiar debugger (GDB/Python/MATLAB/Perl)
(Reversibility can be added in less than a day)
- ▶ The decomposition algorithms for manipulating a history of debugging primitives
- ▶ Transparent checkpointing of GDB sessions in DMTCP
(Distributed MultiThreaded CheckPointing)

URDB: The Architecture



DMTCP: checkpointing an entire GDB session

`ptrace` is a system call that allows a superior process (e.g. GDB) to trace an inferior process (target application) at the binary level.

DMTCP: checkpointing an entire GDB session

`ptrace` is a system call that allows a superior process (e.g. GDB) to trace an inferior process (target application) at the binary level.

Why is this difficult?

The inferior process is normally being traced by GDB.

DMTCP: checkpointing an entire GDB session

`ptrace` is a system call that allows a superior process (e.g. GDB) to trace an inferior process (target application) at the binary level.

Why is this difficult?

The inferior process is normally being traced by GDB.

During a checkpoint, DMTCP has control, and not GDB.

DMTCP: checkpointing an entire GDB session

`ptrace` is a system call that allows a superior process (e.g. GDB) to trace an inferior process (target application) at the binary level.

Why is this difficult?

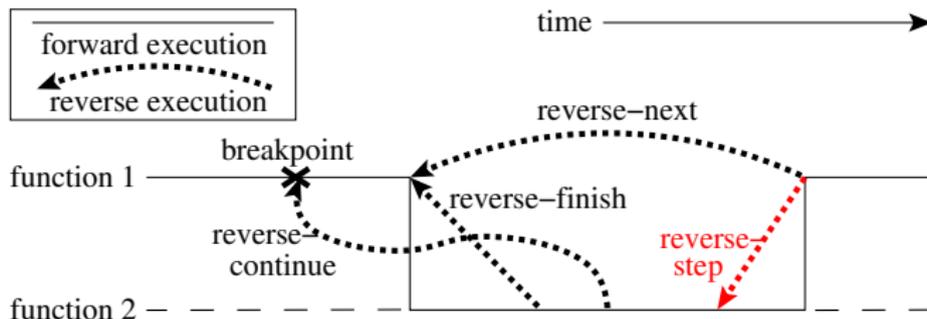
The inferior process is normally being traced by GDB.

During a checkpoint, DMTCP has control, and not GDB.

DMTCP then arranges for GDB to resume tracing the inferior process at the time of resume or restart.

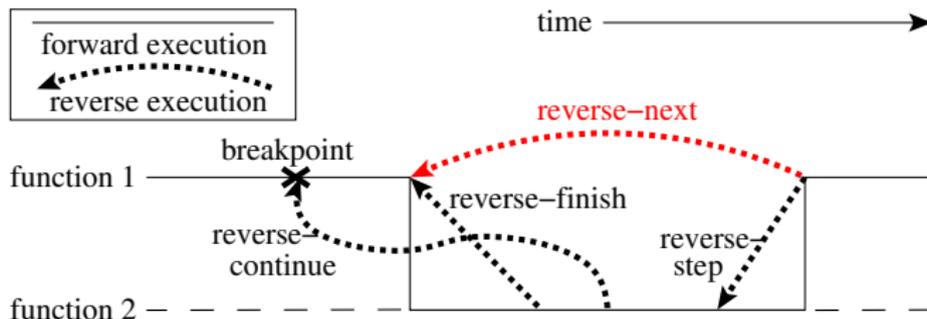
Debugging Primitives and Their Reverse Analogs

- ▶ **step**: enter a function call
- ▶ **next**: do not enter any function calls
- ▶ **continue**: until next breakpoint
- ▶ **finish**: until end of function



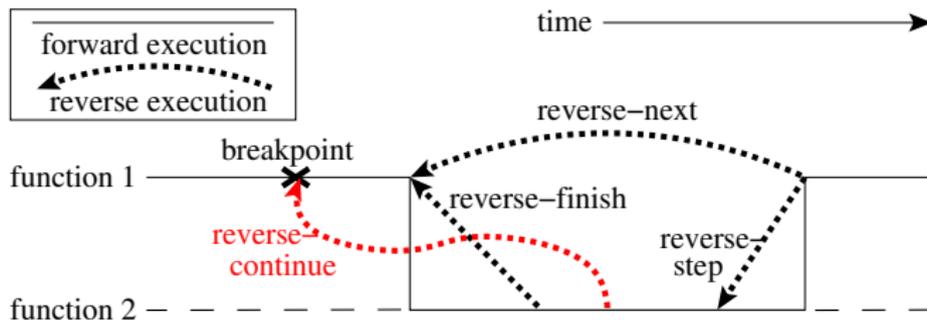
Debugging Primitives and Their Reverse Analogs

- ▶ `step`: enter a function call
- ▶ `next`: do not enter any function calls
- ▶ `continue`: until next breakpoint
- ▶ `finish`: until end of function



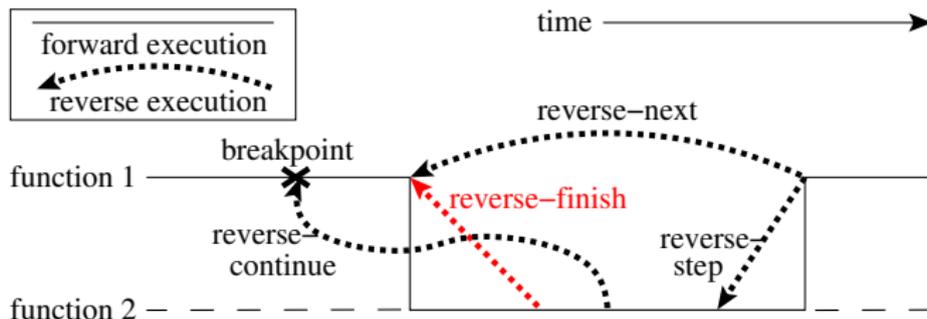
Debugging Primitives and Their Reverse Analogs

- ▶ `step`: enter a function call
- ▶ `next`: do not enter any function calls
- ▶ `continue`: until next breakpoint
- ▶ `finish`: until end of function



Debugging Primitives and Their Reverse Analogs

- ▶ `step`: enter a function call
- ▶ `next`: do not enter any function calls
- ▶ `continue`: until next breakpoint
- ▶ `finish`: until end of function



When a Reverse Command is Issued ...

Prerequisites

- ▶ checkpoint C
- ▶ debugging history h

Algorithm

restart from checkpoint C

$new_h \rightarrow decomposition_algorithm(h)$

/ new_h is the decomposed debugging history */*

restart from checkpoint C

re-execute new_h , minus the last command

URDB: An Example

`reverse-step([step, next, step]) → [step, next]`

URDB: An Example

`reverse-step([step, next, step]) → [step, next]`

`reverse-step([continue, next]) → ?`

Reverse-step

[continue, next]

while True **do**

if last command is continue or next/bkpt **then**

 undo_command()

 do_step()

while we are not at breakpoint **do**

 do_next()

else if last command is step **then**

 undo_command()

 break

else */* last command is next */*

 undo_command()

 do_step()

while deeper() **do**

 do_next()

Reverse-step

[continue]

```
while True do
  if last command is continue or next/bkpt then
    undo_command()
    do_step()
    while we are not at breakpoint do
      do_next()
  else if last command is step then
    undo_command()
    break
  else /* last command is next */
    undo_command()
    do_step()
    while deeper() do
      do_next()
```

Reverse-step

[continue, step]

while True **do**

if last command is continue or next/bkpt **then**

 undo_command()

 do_step()

while we are not at breakpoint **do**

 do_next()

else if last command is step **then**

 undo_command()

 break

else */* last command is next */*

 undo_command()

 do_step()

while deeper() **do**

 do_next()

Reverse-step

[continue, step, next, ..., next, next]

while True **do**

if last command is continue or next/bkpt **then**

 undo_command()

 do_step()

while we are not at breakpoint **do**

 do_next()

else if last command is step **then**

 undo_command()

 break

else */* last command is next */*

 undo_command()

 do_step()

while deeper() **do**

 do_next()

Reverse-step

[continue, step, next, ..., next, **next**]

while True **do**

if last command is continue or next/bkpt **then**

 undo_command()

 do_step()

while we are not at breakpoint **do**

 do_next()

else if last command is step **then**

 undo_command()

 break

else */* last command is next */*

 undo_command()

 do_step()

while deeper() **do**

 do_next()

Reverse-step

[continue, step, next, ..., next]

while True **do**

if last command is continue or next/bkpt **then**

 undo_command()

 do_step()

while we are not at breakpoint **do**

 do_next()

else if last command is step **then**

 undo_command()

 break

else */* last command is next */*

 undo_command()

 do_step()

while deeper() **do**

 do_next()

Reverse-step

[continue, step, next, ..., next, **step**]

while True **do**

if last command is continue or next/bkpt **then**

 undo_command()

 do_step()

while we are not at breakpoint **do**

 do_next()

else if last command is step **then**

 undo_command()

 break

else */* last command is next */*

 undo_command()

do_step()

while deeper() **do**

 do_next()

Reverse-step

[continue, step, next, ..., next, step]

while True **do**

if last command is continue or next/bkpt **then**

 undo_command()

 do_step()

while we are not at breakpoint **do**

 do_next()

else if last command is step **then**

 undo_command()

 break

else */* last command is next */*

 undo_command()

 do_step()

while deeper() **do**

 do_next()

Reverse-step

[continue, step, next, ..., next, **step**]

while True **do**

if last command is continue or next/bkpt **then**

 undo_command()

 do_step()

while we are not at breakpoint **do**

 do_next()

else if **last command is step** **then**

 undo_command()

 break

else */* last command is next */*

 undo_command()

 do_step()

while deeper() **do**

 do_next()

Reverse-step

[continue, step, next, ..., next]

while True **do**

if last command is continue or next/bkpt **then**

 undo_command()

 do_step()

while we are not at breakpoint **do**

 do_next()

else if last command is step **then**

 undo_command()

 break

else */* last command is next */*

 undo_command()

 do_step()

while deeper() **do**

 do_next()

Reverse-step

[continue, step, next, ..., next]

while True **do**

if last command is continue or next/bkpt **then**

 undo_command()

 do_step()

while we are not at breakpoint **do**

 do_next()

else if last command is step **then**

 undo_command()

break

else /* last command is next */

 undo_command()

 do_step()

while deeper() **do**

 do_next()

Multiple Checkpoints

Usefulness

To reduce the cost of replaying histories.

Multiple Checkpoints

Usefulness

To reduce the cost of replaying histories.

Core Idea

- ▶ If the decomposed debugging history is currently empty, then one reverts to the earlier checkpoint and its debugging history.
- ▶ Future Work: extra checkpoints will be taken automatically.

URDB Timings Across Debuggers

The test program inserts (via a function call) twenty numbers into a linked list.

URDB Timings Across Debuggers

The test program inserts (via a function call) twenty numbers into a linked list.

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
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

URDB versus gdb-7.2

- ▶ Test program: a C program that inserts 1,000,000 elements into a linked list (the program allocated its own memory and avoided the use of C malloc).

URDB versus gdb-7.2

- ▶ Test program: a C program that inserts 1,000,000 elements into a linked list (the program allocated its own memory and avoided the use of C malloc).
- ▶ URDB was 5,200 times faster than the target record mode of gdb-7.2!

URDB versus gdb-7.2

- ▶ Test program: a C program that inserts 1,000,000 elements into a linked list (the program allocated its own memory and avoided the use of C malloc).
- ▶ URDB was 5,200 times faster than the target record mode of gdb-7.2!
- ▶ gdb-7.2: the reverse time depends on the number of reverse steps executed.

URDB versus gdb-7.2

- ▶ Test program: a C program that inserts 1,000,000 elements into a linked list (the program allocated its own memory and avoided the use of C malloc).
- ▶ URDB was 5,200 times faster than the target record mode of gdb-7.2!
- ▶ gdb-7.2: the reverse time depends on the number of reverse steps executed.
- ▶ URDB: the reverse time depends on the number of forward instructions from the last checkpoint.

Related Work: 4 previous approaches

(1) Record/Reverse-execute via logging the state of each instruction

- ▶ Grishman, AFIPS, 1970
- ▶ Zelkowitz, Communications of the ACM, 1973
- ▶ Tolmach and Appel, LFP, 1990
- ▶ TotalView
- ▶ gdb-7.2

Related Work: 4 previous approaches

(1) Record/Reverse-execute via logging the state of each instruction

- ▶ Grishman, AFIPS, 1970
- ▶ Zelkowitz, Communications of the ACM, 1973
- ▶ Tolmach and Appel, LFP, 1990
- ▶ TotalView
- ▶ gdb-7.2

(2) Record/Replay via virtual machine snapshots

- ▶ King, Dunlap, and Chen, USENIX, 2005

Related Work: 4 previous approaches

(3) Checkpoint/Re-execute via live checkpoints

- ▶ Feldman and Brown, SIGPLAN Notices, 1989
- ▶ Srinivasan, Kandula, Andrews, and Zhou, USENIX, 2004
- ▶ Leroy, Doligez, Garrigue, Rémy, and Vouillon, OCaml 3.11, 2008
- ▶ Boothe, PLDI, 2000

Related Work: 4 previous approaches

(3) Checkpoint/Re-execute via live checkpoints

- ▶ Feldman and Brown, SIGPLAN Notices, 1989
- ▶ Srinivasan, Kandula, Andrews, and Zhou, USENIX, 2004
- ▶ Leroy, Doligez, Garrigue, Rémy, and Vouillon, OCaml 3.11, 2008
- ▶ Boothe, PLDI, 2000

(4) Post-mortem debugging via logging of events to a database

- ▶ Pothier, Tanter, and Piquer, OOPSLA, 2007
- ▶ Lefebvre, Cully, Feeley, Hutchinson, and Warfield, EuroSys, 2009

Thank you!

Questions?

Reverse-next

while true do

if last command is continue or next/bkpt then

 undo-command()

if last command is next/bkpt and same() then break

else if last command is next/bkpt and deeper() then

/ next/bkpt had exited a function */*

 reverse-finish()

 break

else */* shallower() or last command is continue */*

 do_step()

while we are not at breakpoint do do_next()

else if last command is step or next then

 undo-command()

if same() or shallower() then break

else if deeper() then */* next had exited a function */*

 reverse-finish()

 break

Reverse-next

while true do

if last command is continue or **next/bkpt** **then**

 undo-command()

if last command is next/bkpt and same() **then** break

else if last command is next/bkpt and deeper() **then**

/ next/bkpt had exited a function */*

 reverse-finish()

 break

else */* shallower() or last command is continue */*

 do_step()

while we are not at breakpoint **do** do_next()

else if last command is step or next **then**

 undo-command()

if same() or shallower() **then** break

else if deeper() **then** */* next had exited a function */*

 reverse-finish()

 break

Reverse-next

while true do

if last command is continue or next/bkpt **then**

 undo-command()

if last command is next/bkpt and same() **then** break

else if last command is next/bkpt and deeper() **then**

/ next/bkpt had exited a function */*

 reverse-finish()

 break

else */* shallower() or last command is continue */*

 do_step()

while we are not at breakpoint **do** do_next()

else if last command is step or next **then**

 undo-command()

if same() or shallower() **then** break

else if deeper() **then** */* next had exited a function */*

 reverse-finish()

 break

Reverse-next

while true do

if last command is continue or next/bkpt **then**

 undo-command()

if last command is next/bkpt and **same()** **then** break

else if last command is next/bkpt and **deeper()** **then**

/ next/bkpt had exited a function */*

 reverse-finish()

 break

else */* shallower() or last command is continue */*

 do_step()

while we are not at breakpoint **do** do_next()

else if last command is step or next **then**

 undo-command()

if same() or shallower() **then** break

else if deeper() **then** */* next had exited a function */*

 reverse-finish()

 break

Reverse-next

while true do

if last command is continue or next/bkpt **then**

 undo-command()

if last command is next/bkpt and same() **then** break

else if last command is next/bkpt and deeper() **then**

/ next/bkpt had exited a function */*

 reverse-finish()

 break

else */* shallower() or last command is continue */*

 do_step()

while we are not at breakpoint **do** do_next()

else if last command is step or next **then**

 undo-command()

if same() or shallower() **then** break

else if deeper() **then** */* next had exited a function */*

 reverse-finish()

 break

Reverse-continue

repeat

 undo-command()

until we are at a breakpoint

Reverse-continue

repeat
 undo-command()
until we are at a breakpoint

Note

An optimization can scan the history and replay it until the last breakpoint before the current statement, where `reverse-continue()` was issued.

Reverse-finish

- ▶ The algorithm follows a logic similar in spirit to that of `reverse-next()` and `reverse-step()`.
- ▶ Implementation: execute a sequence of `reverse-next()`'s and checking the stack depth after each `reverse-next()`.