Checkpointing using DMTCP, Condor, Matlab and FReD

Gene Cooperman (presenting)
High Performance Computing Laboratory
College of Computer and Information Science
Northeastern University, Boston
gene@ccs.neu.edu

Joint work with:

joint work with Kapil Arya, Tyler Denniston, and Ana-Maria Visan,
Northeastern University, {kapil,tyler,amvisan}@ccs.neu.edu

Thanks to Peter Keller and Bill Taylor,
Condor Project, U. of Wisconsin-Madison,
for their collaboration in this work.

DMTCP Overview

DMTCP (Distributed MultiThreaded CheckPointing):

- *Mature:* seven years in development
- *Robust:* current user base in hundreds and growing
- *Non-invasive:* no root privilege needed; no kernel modules; transparently operates on binaries, no application source code needed
- *Fast:* checkpoint/restart in less than a second (dominated by disk time)
- *Versatile:* works on OpenMPI, MATLAB, R, Python, bash, gdb, X-Windows apps, etc.
- *Open Source:* freely available at `http://dmtcp.sourceforge.net` (LGPL)
- *Debian package:* Debian testing
- *OpenMPI checkpoint-restart service:* soon to support MTCP/DMTCP. (Courtesy of Alex Brick, with help from Jeffrey Squyres and Joshua Hursey)

STANDALONE USAGE:

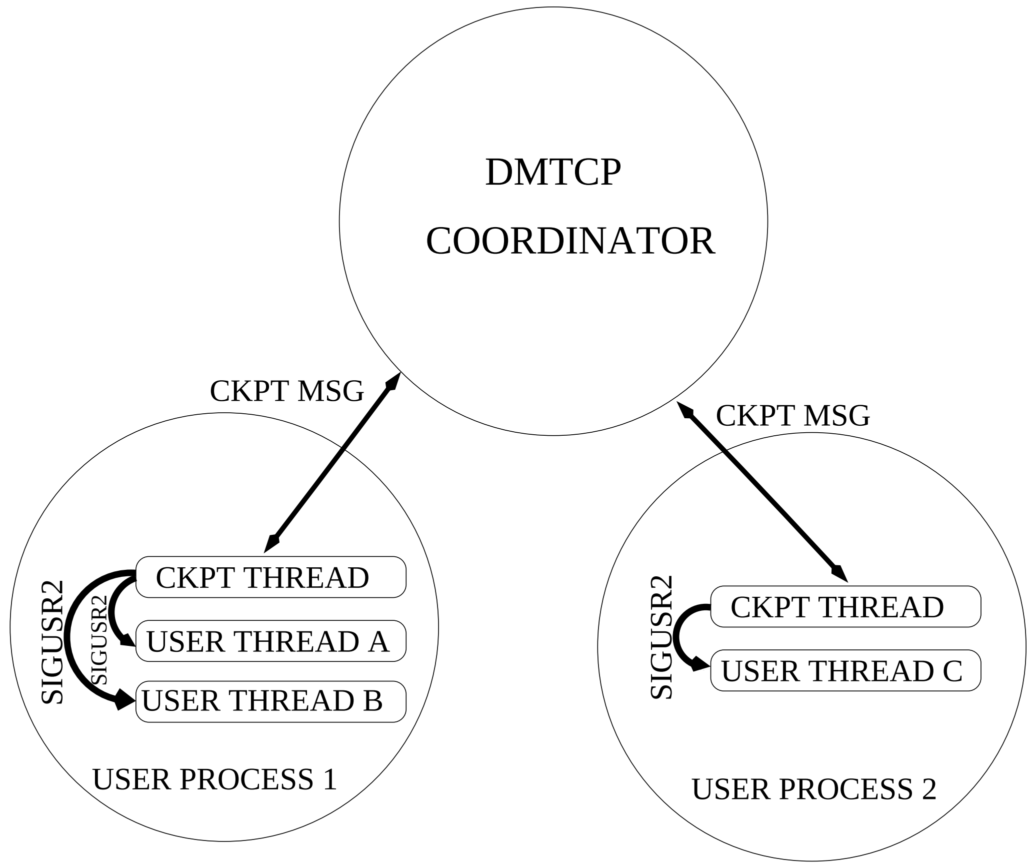
```
dmtcp_checkpoint a.out
dmtcp_command --checkpoint
dmtcp_restart ckpt_a.out_*.dmtcp
```

DMTCP: NEWS

- Undergoing validation with respect to feature set of standard universe
- Passed all automated validation tests; now undergoing “manual” tests
- Now supported by a National Science Foundation grant

DMTCP: How Does It Work?

Provides fast checkpoint-restart (typically less than a second)
10 MB checkpoint disk image typical (typical, based on footprint in RAM)
Dynamic compression of checkpoint images (enabled by default)



Design of DMTCP Checkpoint Threads

- Guarantees that the process is either in user mode or checkpoint mode — not both
- When in user mode, checkpoint thread listens on socket to DMTCP coordinator
- When CKPT message received, checkpoint thread sends SIGUSR2 signal to *each* user thread to force it into DMTCP signal handler
- After all user threads are waiting in signal handler, checkpoint thread copies memory, kernel state and network data to disk
- DMTCP then releases all user threads and returns to listening on socket to coordinator
- Because multiple user threads are handled, this works both in the Condor Standard Universe and the Condor Vanilla Universe.
- (In contrast, current Condor checkpointing in Standard Universe based always on a single thread — either that thread operates in user mode, or it receives a signal forcing it into checkpoint mode.)

DMTCP Features

- Distributed MultiThreaded CheckPointing
- Works with Linux kernel 2.6.9 and later
- Supports sequential and multi-threaded computations across single/multiple hosts
- Entirely in user space (no kernel modules or root privilege)
- Transparent (no recompiling, no re-linking)
- DMTCP Team centered around Northeastern U., with collaborators from MIT and Siberian State U. of Telecom. and Informatics
- LGPL, freely available from `http://dmtcp.sourceforge.net`
- No Remote I/O (except through certain extensions)

Some DMTCP Features Relevant to Condor

Features useful for Standard and for Vanilla Universe:

- Multiple processes allowed: fork() is supported
- Multiple threads allowed: POSIX Threads is supported
- Calls to mmap() are supported
- No need to re-link : Original binary is supported
- *Supports Matlab and R jobs* (see later in slides)

DMTCP Process Migration across Linux Kernels

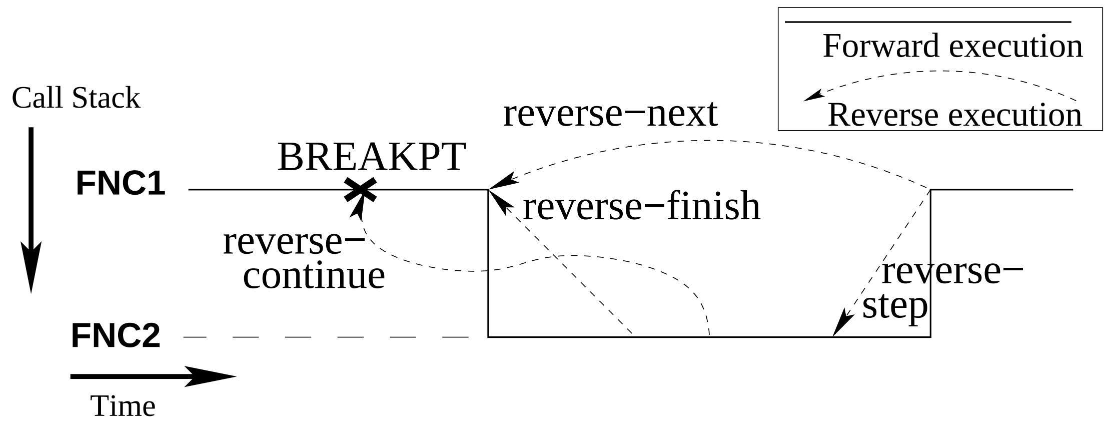
- Compatibility Level 1: As of DMTCP-1.2.1, it can be compiled on a Linux kernel between 2.6.18 and 2.6.35, and run on another kernel in that range. (Thanks to a major corporation for helping test this across a variety of hosts.)
- Compatibility Level 2: In the upcoming DMTCP-1.2.2 release, it can checkpoint under Linux kernel 2.6.35 and run under Linux kernel 2.6.18 (backward compatibility), as well as in the other direction (forward compatibility). (Backward compatibility still undergoing further testing.)
- Some process migration compatibility with Linux kernel 2.6.9 is observed. However, this is not a current priority.

Condor/DMTCP/Matlab (and R)

- *NEWS:* New “shim script” enables checkpointing of Condor (and R) jobs in Condor Vanilla Universe.
  (based on shim script of Pete Keller)
- *ISSUES:* Under process migration, absolute pathnames change and symbolic links are not preserved
  1. This affected DMTCP (fixed in March, 2011)
  2. This also affects the shim script: DMTCP produces `dmtcp_restart_script.sh`, which is a symbolic link to: latest `dmtcp_restart_script_`**LONG_ID**`.sh`
- *SOLUTION:* In consultation with Pete Keller and Bill Taylor, writing a modified shim script to fix this.
- *Collaborators welcome for testing in a production Condor environment*

Same principles can be used to support R.

FReD: Fast Reversible Debugger



Undo: *if n commands beyond the last checkpoint, then restart and re-execute first n – 1 commands*
(Note: for non-deterministic programs, re-execution can leave the process in a different state; more about that later)

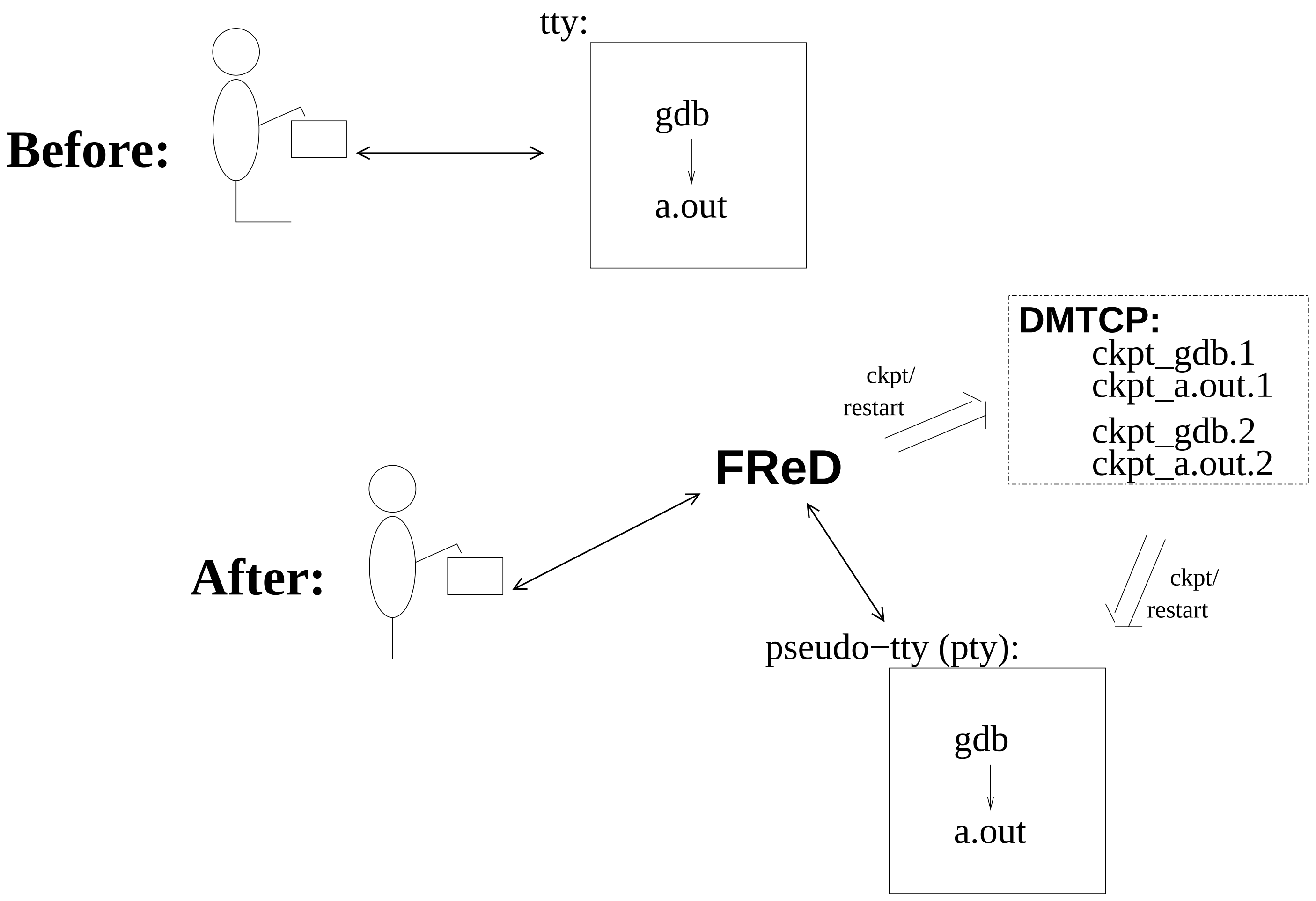
Extend to: reverse-step, reverse-next, reverse-finish, reverse-watch, etc.

FReD and GDB

- GDB already has reversibility (since gdb-6.8)
- USAGE: `(gdb) help target record;` `reverse-next`, etc.
- Works great for *single-threaded* computations that run for *less than one second*; slow due to interpreted nature
- Method: Interpret at assembly language and log information at each assembly instruction (if `store`, log the old value in main memory; if `load`, log old value in register)

FReD is intended for long-running programs, and multi-threaded programs. Tested on such real-world programs as MySQL (many threads, much concurrency).

Architecture of FReD



Implementation: Logging of most system calls

Key Point: Fast lightweight logging of *just enough* to ensure determinism: anything stronger than that would make it too slow; anything weaker would risk a different thread schedule on replay
NOTE: Issues of strong and weak determinism outside scope of this talk.

Implementation: Standard mechanisms like dlopen/dlsym allow one to add wrappers around all library calls.

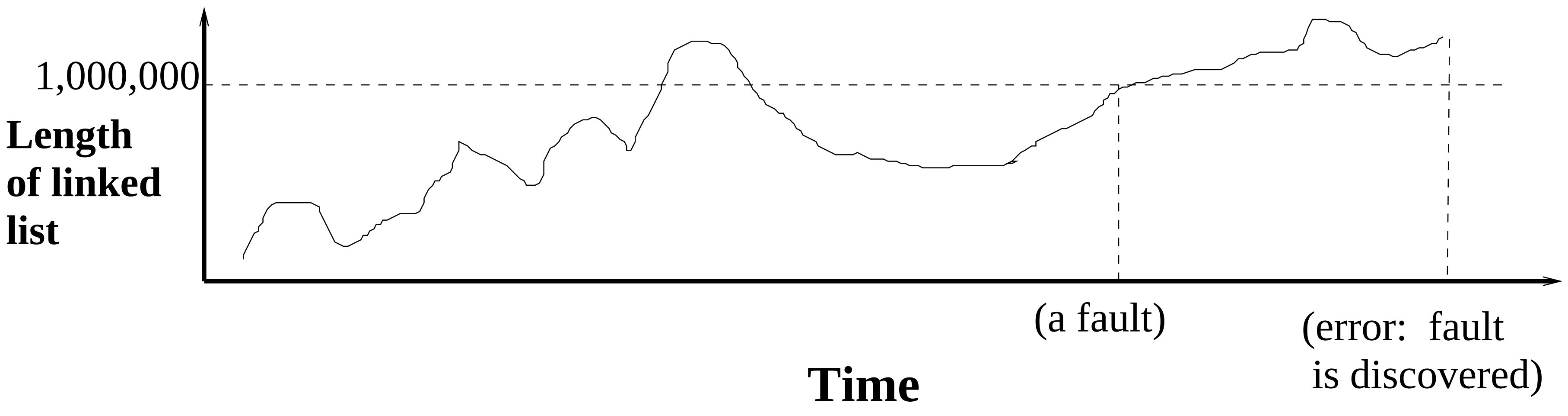
- Logging all system calls that touch disk: *Never touch disk on re-execute.*
- No need to save open files of program at checkpoint time — not needed during re-execute.
- Log all calls for thread synchronization (mutex, semaphore, condition variables, etc.)
- Log all calls to malloc, free, etc. On re-execute, memory allocation calls must be replayed in same order in order to guarantee memory-accuracy in multi-threaded programs.

Memory Accuracy for Reversible Debugging

Definition of memory accuracy: Absolute address of object at re-execute is same as on original execution.

- Memory accuracy is easy for single-threaded programs, but . . .
- *Note:* With multiple threads, races possible among two malloc's
- *Importance:* Consider trying to reversibly debug a linked list if the address of a link can change when re-executing the same statement

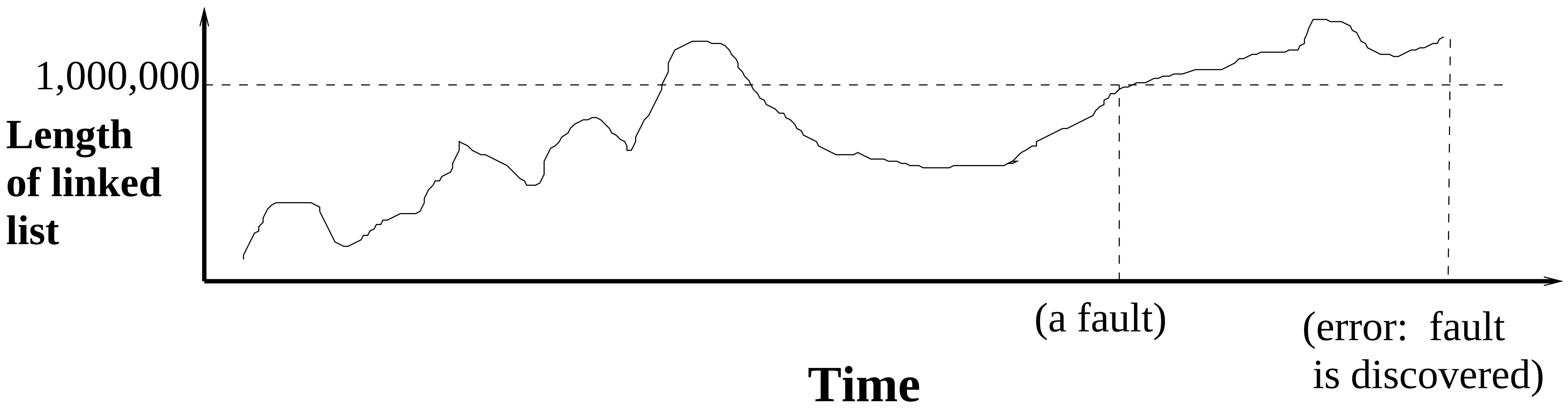
Temporal Debugging: Problem



Scenario (two-point bug: fault and error): A bug (error) manifests. We want to run the program backwards to find the cause of the bug.

1. Examine bug and determine an expression that has the “wrong” value.
2. At an earlier time, the expression had the “right” value. When did it take on the “wrong” value?
3. If the expression were a single memory address or variable, other good technologies exist (e.g. gdb hardware watchpoints), if the expression depends on many addresses, a more efficient, general solution is needed.

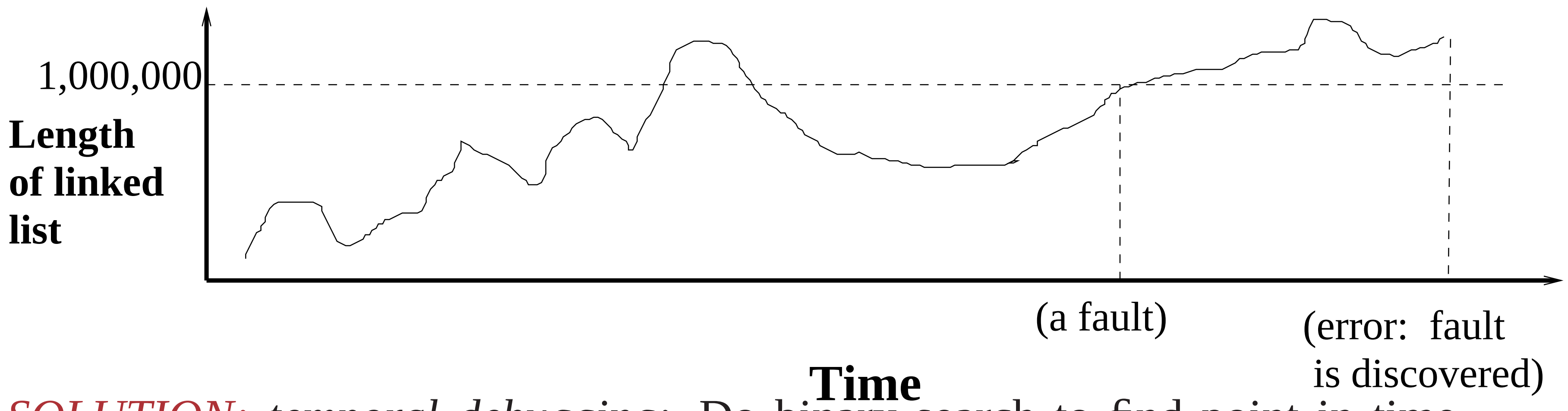
Temporal Debugging: Solution: Reverse Expression Watchpoint



- *SOLUTION using FReD/DMTCP:* Do a binary search through the time dimension to find the point in time when the expression first took on the "wrong" value.

Example: A data structure is occasionally re-computed based on a linked list. The linked list is assumed to always have length less than 1,000,000. The error shows that the linked list now has length 1,050,000. How did this happen?

Reverse-Watch via Binary Search



SOLUTION: temporal debugging: Do binary search to find point in time when expression is "right", but will be "wrong" at next statement.
Let N be the number of statements executed over the program lifetime.
For most programs, $N < 10^{15}$ statements, and so $\log_2 N < 50$.

- *Checkpoint/restart time:* 50 checkpoints and 50 restarts (~ 100 sec.)
- *Running time:* Approximately original runtime using additional checkpoints, and $\log_2 N$ probes evaluating expression at $\log_2 N$ different times.
- **NOTE:** FReD/DMTCP runs at the native speed of the application when not checkpointing or restarting.

Demo: Part 1

```
gene@bsn89k1:~/pthread-wrappers/fred$ ./fredapp.py --fred-demo gdb ../test-programs/test_list
GNU gdb (GDB) 7.0-ubuntu
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
...
Reading symbols from /home/gene/pthread-wrappers/test-programs/test_list...done.
(gdb) b main
Breakpoint 1 at 0x4005fc: file test_list.c, line 21.
FReD: 'b main' took 0.054 seconds.
Starting program: /home/gene/pthread-wrappers/test-programs/test_list

Breakpoint 1, main () at test_list.c:21
(gdb) r
21    head = newItem(1);
FReD: 'r' took 0.892 seconds.
(gdb) fred-ckpt
FReD: Created checkpoint #0.
FReD: 'fred-ckpt' took 1.841 seconds.
```

Demo: Part 2

```
(gdb) list
17 int main() {
18   item * tail;
19   int i;
20
21   head = newItem(1);
22   tail = head;
23   printf(" NODE VAL: %d\n", tail->val);
24   for(i=2;i<=20;i++) {
25     tail->next = newItem(i);
26     tail = tail->next;
27     printf(" NODE VAL: %d\n", tail->val);
28   }
29   printf("Linked list length is now: %d\n", list_len(head));
...
33 }
34
35 item * newItem(int i) {
36   item * tmp = malloc(sizeof(item));
37   tmp->val = i;
38   tmp->next = NULL;
39   return tmp;
40 }
```

Demo: Part 3

```
FReD: 'list' took 0.036 seconds.
(gdb) b 30
Breakpoint 2 at 0x4006a2: file test_list.c, line 30.
FReD: 'b 30' took 0.040 seconds.
(gdb) c
Continuing.
 NODE VAL: 1
 NODE VAL: 2
...
 NODE VAL: 20
Linked list length is now: 20

Breakpoint 2, main () at test_list.c:30
30   printf ("Ok we crossed the limit."
FReD: 'c' took 0.032 seconds.
```

Demo: Part 4

```
(gdb) fred-reverse-watch list_len(head)<7
===================== KILLING gdb =====================
===================== RESTARTING gdb =====================
next
22    tail = head;
next
23    printf(" NODE VAL: %d\n", tail->val);
next 2
 NODE VAL: 1
25      tail->next = newItem(i);
next 4
 NODE VAL: 2
25      tail->next = newItem(i);
next 8
 NODE VAL: 3
 NODE VAL: 4
25      tail->next = newItem(i);
```

Demo: Part 5

```
next 16
 NODE VAL: 5
 NODE VAL: 6
 NODE VAL: 7
 NODE VAL: 8
25     tail->next = newItem(i);
===================== KILLING gdb =====================
===================== RESTARTING gdb =====================
dmtcp_coordinator starting...
    Port: 7770
    Checkpoint Interval: disabled (checkpoint manually instead)
    Exit on last client: 1
Backgrounding...
next 24
 NODE VAL: 1
 NODE VAL: 2
...
 NODE VAL: 6
25     tail->next = newItem(i);
```

Demo: Part 6a

```
===================== KILLING gdb =====================
===================== RESTARTING gdb =====================
next 28
 NODE VAL: 1
 NODE VAL: 2
...
 NODE VAL: 7
25      tail->next = newItem(i);
===================== KILLING gdb =====================
===================== RESTARTING gdb =====================
next 26
 NODE VAL: 1
 NODE VAL: 2
...
 NODE VAL: 6
27      printf(" NODE VAL: %d\n", tail->val);
===================== KILLING gdb =====================
===================== RESTARTING gdb =====================
next 25
 NODE VAL: 1
 NODE VAL: 2
...
 NODE VAL: 6
26      tail = tail->next;
```

Demo: Part 6b

```
===================== KILLING gdb =====================
===================== RESTARTING gdb =====================
next 24
 NODE VAL: 1
 NODE VAL: 2
...
 NODE VAL: 6
25      tail->next = newItem(i);
===================== KILLING gdb =====================
===================== RESTARTING gdb =====================
next 24
 NODE VAL: 1
 NODE VAL: 2
...
 NODE VAL: 6
25      tail->next = newItem(i);
step
newItem (i=7) at test_list.c:36
36    item * tmp = malloc(sizeof(item));
next
37    tmp->val = i;
next
38    tmp->next = NULL;
```

Demo: Part 6c

```
next 2
40 }
next 4
 NODE VAL: 7
25      tail->next = newItem(i);
===================== KILLING gdb =====================
===================== RESTARTING gdb =====================
next 24
 NODE VAL: 1
 NODE VAL: 2
...
 NODE VAL: 6
25      tail->next = newItem(i);
step
newItem (i=7) at test_list.c:36
36    item * tmp = malloc(sizeof(item));
next 6
27      printf(" NODE VAL: %d\n", tail->val);
```

Demo: Part 6d

```
===================== KILLING gdb =====================
===================== RESTARTING gdb =====================
next 24
 NODE VAL: 1
 NODE VAL: 2
...
 NODE VAL: 6
25      tail->next = newItem(i);
step
newItem (i=7) at test_list.c:36
36   item * tmp = malloc(sizeof(item));
next 5
main () at test_list.c:26
26      tail = tail->next;
===================== KILLING gdb =====================
===================== RESTARTING gdb =====================
next 24
 NODE VAL: 1
 NODE VAL: 2
...
 NODE VAL: 6
25      tail->next = newItem(i);
```

Demo: Part 6e

```
step
newItem (i=7) at test_list.c:36
36   item * tmp = malloc(sizeof(item));
next 4
40 }
===================== KILLING gdb =====================
===================== RESTARTING gdb =====================
next 24
 NODE VAL: 1
...
 NODE VAL: 6
25     tail->next = newItem(i);
```

Demo: Part 7

```
step
newItem (i=7) at test_list.c:36
36   item * tmp = malloc(sizeof(item));
next 4
40 }
step
main () at test_list.c:26
26     tail = tail->next;
===================== KILLING gdb =====================
===================== RESTARTING gdb =====================
next 24
 NODE VAL: 1
...
 NODE VAL: 6
25     tail->next = newItem(i);
step
newItem (i=7) at test_list.c:36
36   item * tmp = malloc(sizeof(item));
next 4
40 }
```

Demo: Part 8

```
FReD: ’fred-rw list_len(head)<7’ took 27.158 seconds.
(gdb) where
#0  newItem (i=7) at test_list.c:40
#1  0x0000000000400645 in main () at test_list.c:25
FReD: ’where’ took 0.044 seconds.
(gdb) list
35 item * newItem(int i) {
36   item * tmp = malloc(sizeof(item));
37   tmp->val = i;
38   tmp->next = NULL;
39   return tmp;
40 }
41 int list_len(item *elt) {
42   int count = 0;
43   while (elt != NULL) {
44     elt = elt->next;
```

Demo: Part 9

```
(gdb) p list_len(head)
$1 = 6
FReD: 'p list_len(head)' took 0.048 seconds.
(gdb) step
main () at test_list.c:26
26      tail = tail->next;
FReD: 'step' took 0.040 seconds.
(gdb) where
#0  main () at test_list.c:26
FReD: 'where' took 0.060 seconds.
(gdb) p list_len(head)
$2 = 7
FReD: 'p list_len(head)' took 0.040 seconds.
(gdb) fred-reverse-step
FReD: 'fred-reverse-step' took 1.663 seconds.
(gdb) where
#0  newItem (i=7) at test_list.c:40
#1  0x0000000000400645 in main () at test_list.c:25
FReD: 'where' took 0.118 seconds.
(gdb) p list_len(head)
$2 = 6
FReD: 'p list_len(head)' took 0.108 seconds.
```

Demo: Part 10

```
(gdb) fred-help
Supported monitor commands follow.  Optional COUNT argument is repeat
  fred-undo <COUNT=1>:         Undo last debugger command.
  fred-reverse-next <COUNT=1>, fred-rn <COUNT=1>:  Reverse-Next Comma
  fred-reverse-step <COUNT=1>, fred-rs <COUNT=1>:  Reverse-Step Comma
  fred-checkpoint, fred-ckpt:   Request a new checkpoint to be made.
  fred-restart:                Restart from last checkpoint.
  fred-reverse-watch <EXPR>, fred-rw <EXPR:
                       Reverse execute until expression EXPR changes
  fred-debug <EXPR>:           Experts only: debug python expression
                              If no argument:  enter pdb debugger.
  fred-source <FILE>:          Read commands from source file.
  fred-list:                   List the available checkpoint files.
  fred-help:                   Display this help message.
  fred-history:          Display your command history up to this point
  fred-quit, fred-exit:         Quit FReD.
```

Questions

- DMTCP
  - `http://dmtcp.sourceforge.net` : `dmtcp-forum@lists.sourceforge.net`
  - Alias of developers: `dmtcp@ccs.neu.edu`
- Acknowledgement:
  The DMTCP project is partially supported by the National Science Foundation under grant OCI-0960978. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.