Temporal Debugging via Flexible Checkpointing: Changing the Cost Model

Gene Cooperman (presenting)
High Performance Computing Laboratory
College of Computer and Information Science
Northeastern University
Boston, Massachusetts 02115
USA
gene@ccs.neu.edu

joint with Kapil Arya, Tyler Denniston, Ana-Maria Visan

A Talk in Three Parts

- **URDB (Universal Reversible Debugger)**:
  - Good support for *single-threaded* apps
  - Supports multiple debuggers: GDB, Python/pdb, perl -d, Matlab debugger
  - Reverse expression watchpoints (analogous to GDB 'expression watchpoints', but in reverse direction)
- **FReD (Fast Reversible Debugger)**:
  - Good support for *multi-threaded* apps
  - Built on top of URDB
  - Supports Deterministic Replay of multi-threaded, multi-core applications at near native speeds
  - Sufficiently robust to run on MySQL
- **Future Work**

Background

DMTCP (Distributed MultiThreaded CheckPointing, `http://dmtcp.sourceforge.net`) is a mature open source (LGPL) checkpointing package that has been under development for seven years.

It operates entirely in user space, with no kernel modules, or modifications to the target application. If used in the simplest possible way, it works as:

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

DMTCP is contagious. Any calls to fork(), pthread_create(), or "ssh", are recognized by DMTCP, and it adds to checkpoint control all new threads, and processes (both local and remote). At checkpoint time, it also generates a script, `dmtcp_restart_script.sh`, that can restart a distributed computation.

It works transparently on most Linux non-graphics software, including: Matlab, R, Python, Perl, bash, tcsh, Open MPI, OpenMP, Cilk, GNU screen, vim, emacs, ...

URDB and FReD

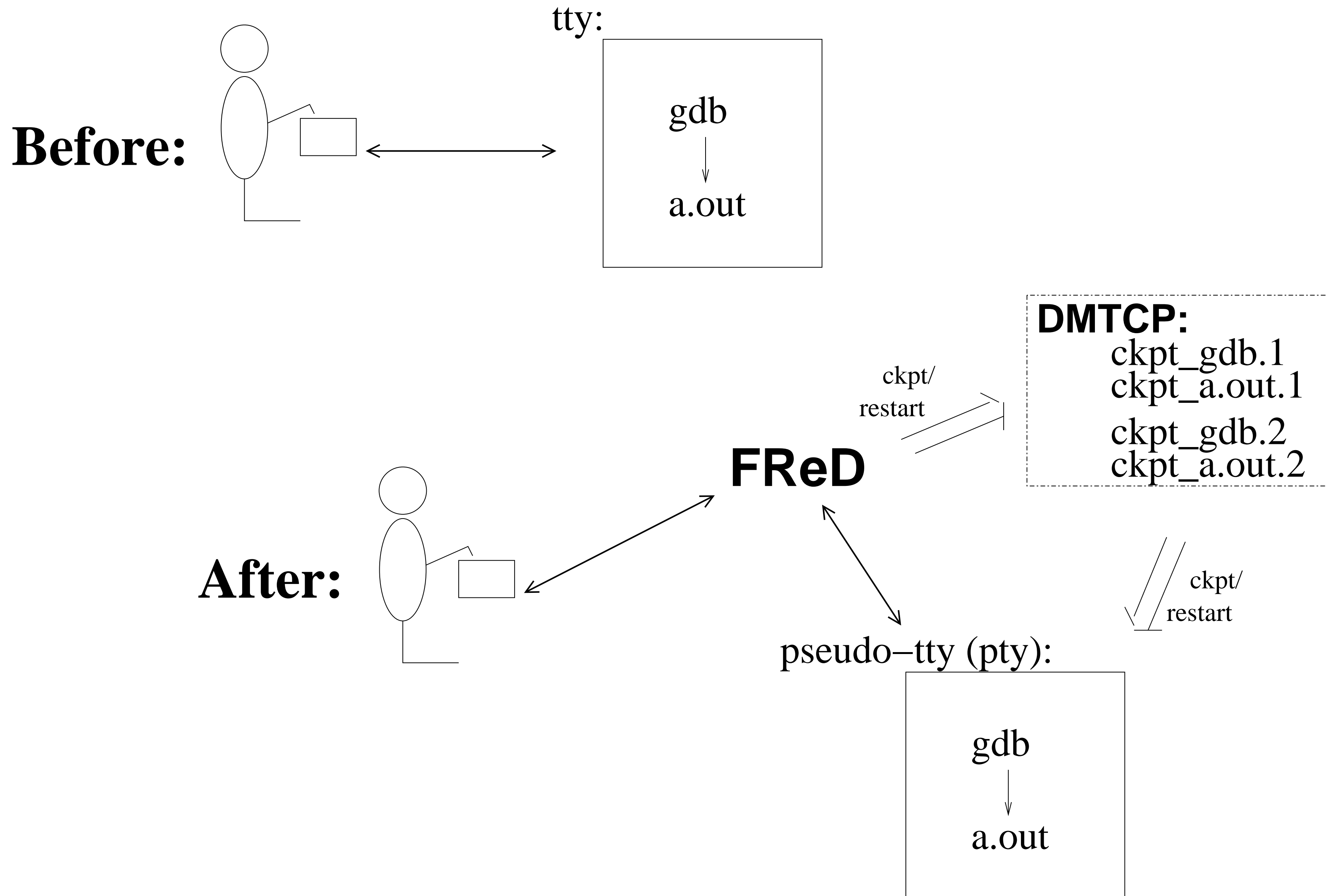
URDB (Universal Reversible Debugger)/single-threaded: Based on ability of DMTCP to checkpoint and restart a debugging session fast; works for GDB, Python, Perl, Matlab; 200 lines of python code to add a new target debugger;
freely available at `http://urdb.sourceforge.net`

WARNING: Not ready for prime-time (older technical report on URDB also available)

FReD (Fast Reversible Debugger)/multi-threaded: Handles multi-threaded, muli-core apps; good enough to reversibly debug MySQL; further bug-swatting underway to handle most major multi-threaded software;
Intended for public release in the next month or two

Note: Reversible debuggers have existed at least since 1970. The problem is to make them widely useful.

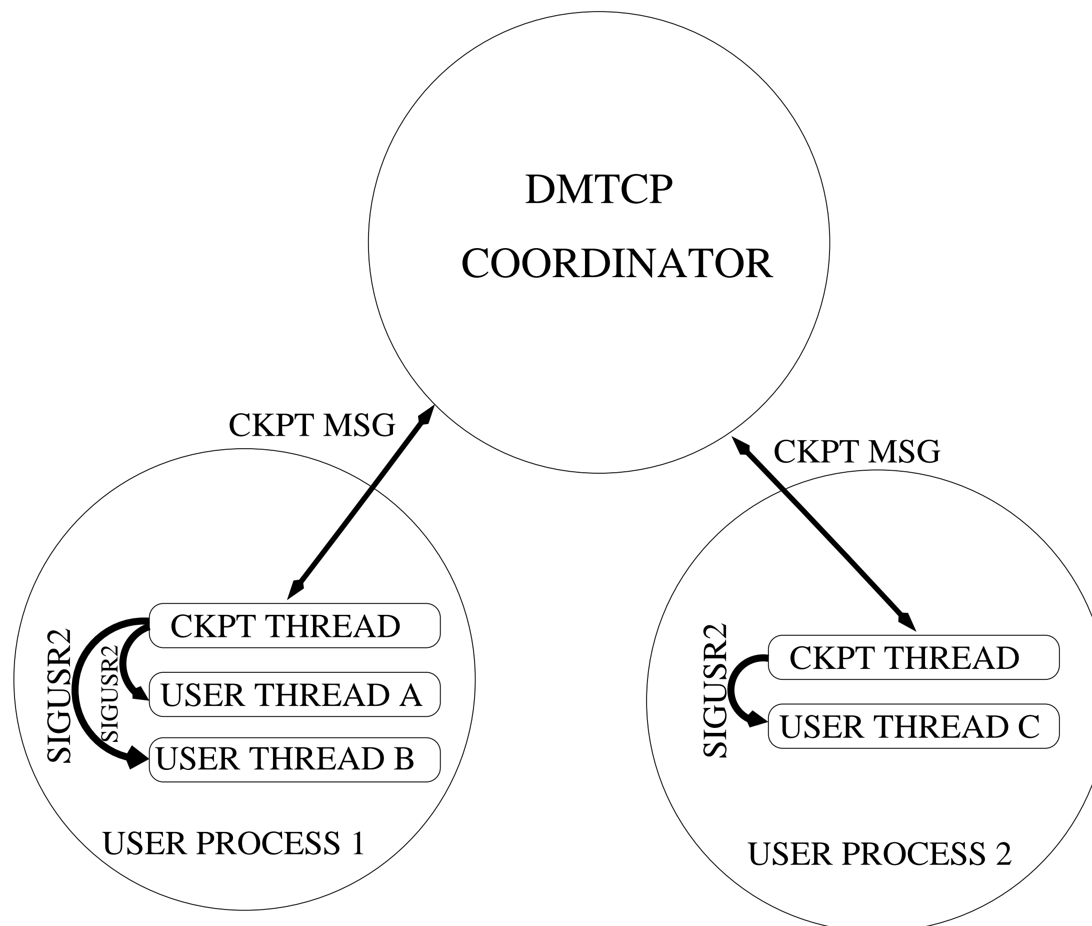
URDB Architecture



DMTCP: Distributed MultiThreaded CheckPointing

Northeastern UNIVERSITY

Provides fast checkpoint-restart (about one second)
10 MB checkpoint typical (based on footprint in RAM)
(further speed and storage optimizations planned using incremental checkpointing)



DMTCP: Wider Usage

- *Open Source:* Over 100 downloads per month from Sourceforge
- *Linux Distros:* in Debian testing; will be in OpenSuse 12.1 (Nov., 2011); submitted to Fedora;
- *Condor:* Integration with Condor as checkpointer for Standard Universe (as well as Vanilla Universe): `https://condor-wiki.cs.wisc.edu/index.cgi/wiki?p=DmtcpCondor` (Condor is a package for high throughput computing, installed on over 300,000 hosts.)
- *Open MPI:* RFC under review now (Oct., 2011) for a DMTCP module to be used by the Open MPI checkpoint-restart service — to be included in Open MPI 1.6 (to be released: Dec., 2011?) if successful.
- *Users:* Recommended to Users for these projects (among others): iPlant collaborative, `http://www.iplantcollaborative.org/grand-challenges/about-grand-challenges/curr` HOL Light Theorem Prover (`http://www.cl.cam.ac.uk/~jrh13/hol-light/`) ; NeuroDebian (neuroscience: `http://neuro.debian.net/`)

URDB Key Idea: Restart-Reexecute

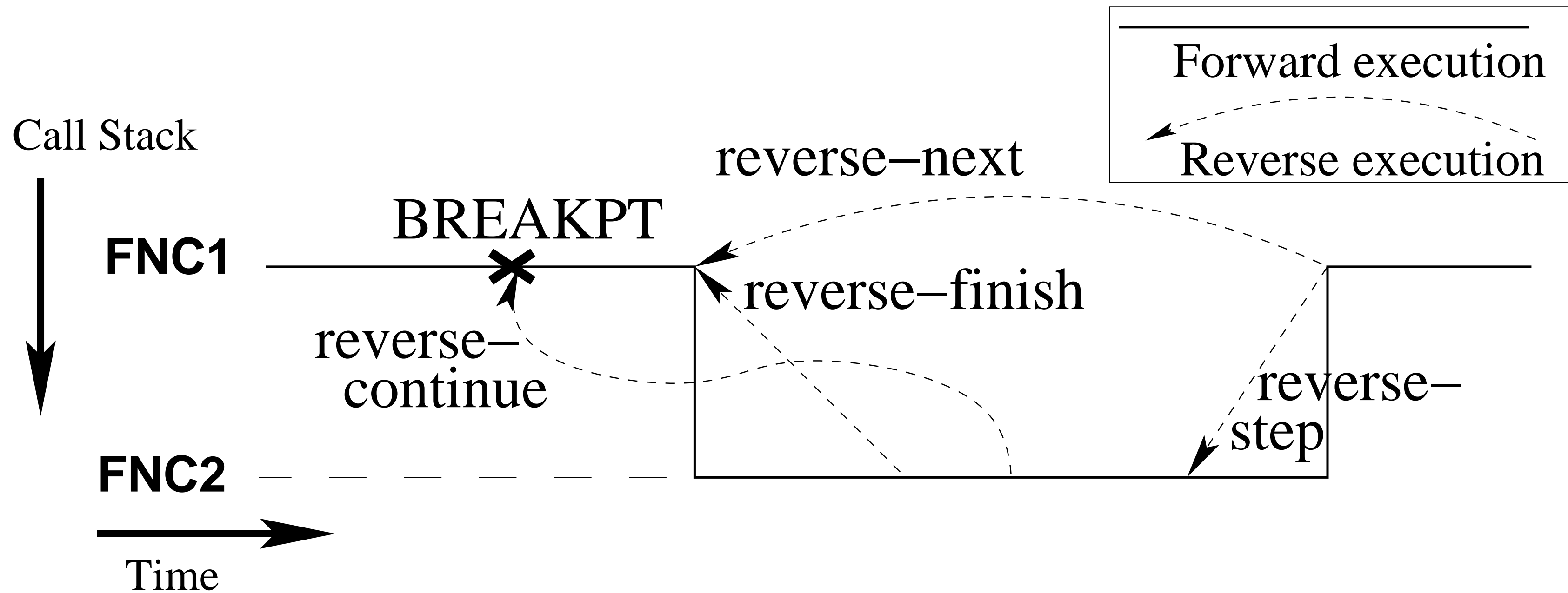
Undo: *if n commands beyond the last checkpoint, then restart and re-execute first $n-1$ commands*
(Note: for non-deterministic programs, re-execution can leave the process in a different state; more about that later)

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

Key features of DMTCP:

- fast checkpoint/restart times (about one second)
  (will be even faster in future with incremental checkpoints implemented)
- works on multiple processes
- *also works on GDB debugging sessions*
  (supports Linux ptrace system call)
- works on multiple threads
- works on distributed computations

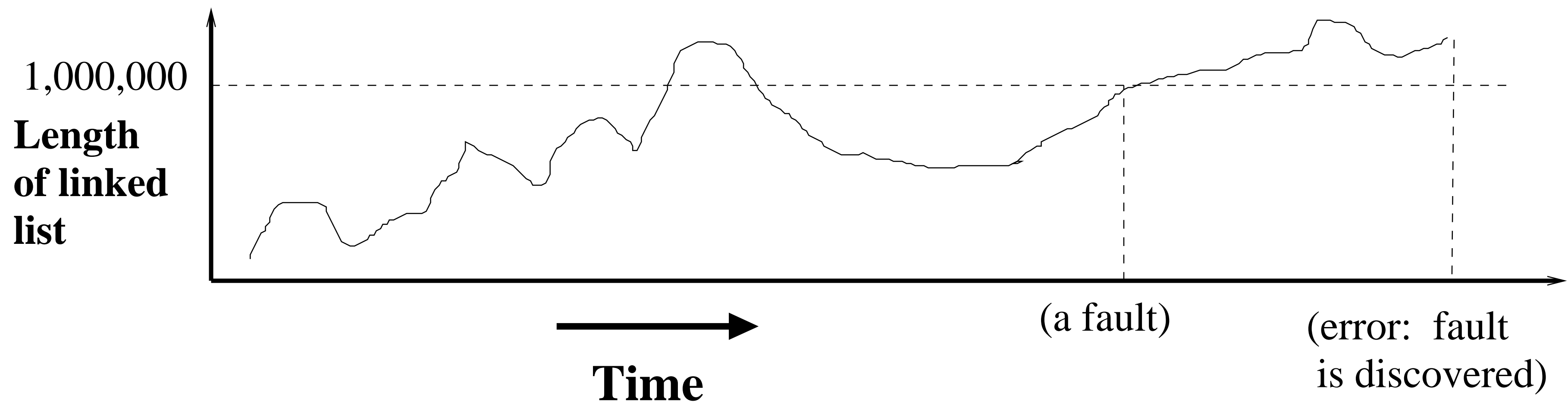
Some URDB Commands



And some more commands:

- `checkpoint`, `restart` (about a second to checkpoint ; restart is faster)
- `undo` (undo last debugger command)
- `reverse-watch <STMT>` (use binary search to find stmt where EXPR changed)

Temporal Debugging: Reverse-Watch



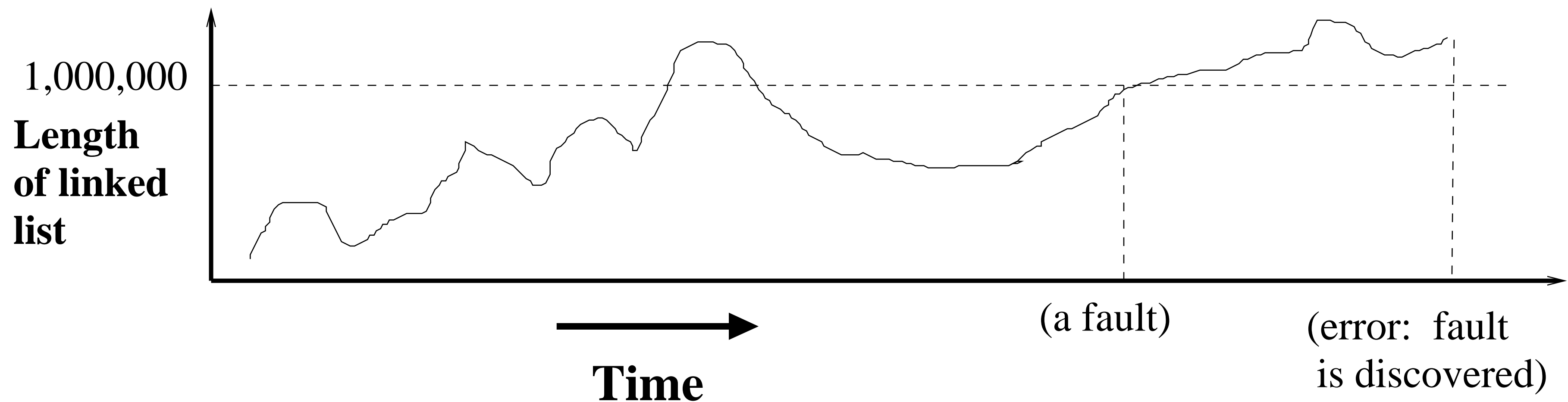
Scenario (two-point bug): Fault occurs earlier, but error manifests much later. After analyzing the error, we produce a program expression that exhibits the fault. How do we find a statement for which the expression had a "good value" prior to the statement execution and a "bad value" afterwards?

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?

Problem: The linked list is updated 1,000,000 times per second, and the error manifests only after running for an hour.

Northeastern UNIVERSITY

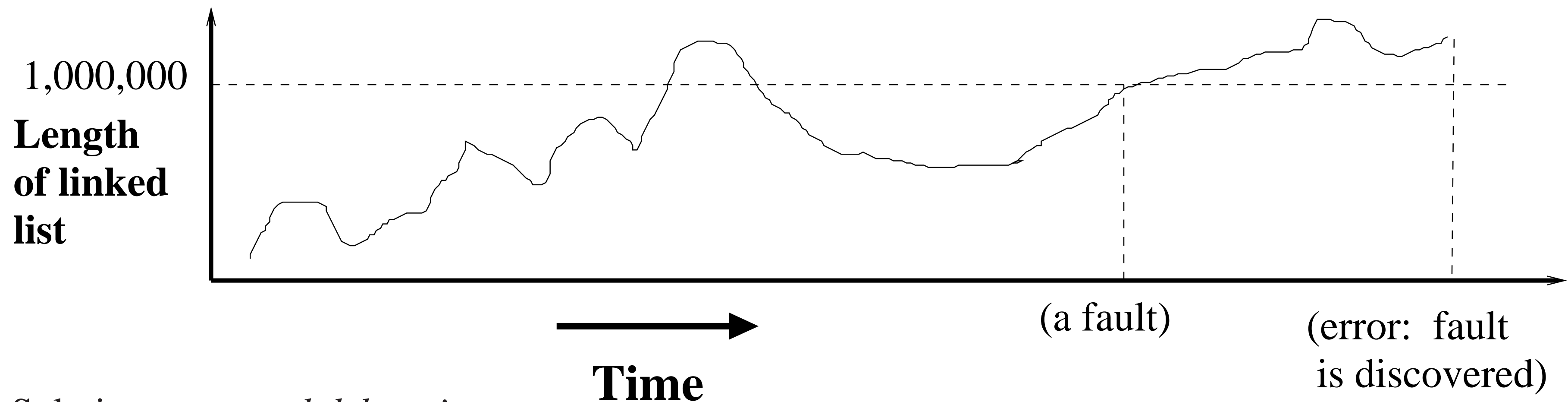
Reverse-Watch via Binary Search



Ordinary expression watchpoints are efficient *only* if the expression represents a single hardware address in memory. Standard techniques exist for this case. (Remove write permission for the page of interest, and add a segfault handler to capture a write access.)

When the expression is more complex (for example, a linked list), debuggers such as GDB just evaluate the expression at every statement. In our linked list example, this is inefficient for a program running over an hour. Checking every update to the linked list would also be inefficient: 1,000,000 links $\times$ 1,000,000 updates per second $\times$ 3600 seconds $= 3.6 \times 10^{15}$ links to traverse. The clock rate of a CPU is 4×10^9 clock cycles per second.

Reverse-Watch via Binary Search (cont.)



Solution: *temporal debugging*

Let N be the number of statements executed over the program lifetime.

Binary search over the process lifetime needs only $\log_2 N$ evaluations of the linked list length.

For most programs $N < 10^{15}$ statements, and so $\log_2 N < 50$.

- *Checkpoint and restart time:* 50 checkpoints and 50 restarts require about 100 seconds under DMTCP.
- *Running time:* Less than double the original time to run the application. (Temporary intermediate checkpoints are inserted for efficiency as needed.)
- **NOTE:** URDB/DMTCP runs at the native speed of the application when not checkpointing or restarting.

Speculation on the Future

By the end of this decade, diagnosing *run-time errors will be almost as painless as diagnosing compile-time errors.*

- Compile-time errors used to be hard to diagnose. (unmatched parentheses, mismatched types, finding definition of function/macro among many files)
- Diagnosing compile-time errors is now *usually* reasonably painless.
- Diagnosing compile-time errors requires searching in a textual or spatial dimension for which there are now excellent tools.
- URDB (and now FReD) converts the temporal dimension into a spatial dimension. (Unlike the unidirectional *arrow of time*, one can now efficiently search bidirectionally in a space-like dimension.)
- It is much easier to build tools like reverse expression watchpoint for a spatial dimension than for a temporal dimension.
- The next decade will see a proliferation of bidirectional *temporal* tools for diagnosing bugs.

Demo: Part 1

```
gene@bsn89k1:~/pthread-wrappers/fred$ ./fredapp.py --fred-demo gdb ../test-program
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

Northeastern University

```
(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;
```

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 6

```
===================== 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;
```

```
 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;
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
...
```

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

Demo: Part 7

```
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 }
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

Northeastern University

```
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

Northeastern University

```
(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 count.
  fred-undo <COUNT=1>:           Undo last debugger command.
  fred-reverse-next <COUNT=1>, fred-rn <COUNT=1>:  Reverse-Next Command
  fred-reverse-step <COUNT=1>, fred-rs <COUNT=1>:  Reverse-Step Command
  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 URDB.
```

A Calculus of Debugging Commands

```
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 1: Reverse-step*

* Ana-Maria Visan, Kapil Arya, Gene Cooperman, Tyler Denniston, "URDB: A Universal Reversible Debugger Based on Decomposing Debugging Histories", PLOS workshop, SOSSOSP-2011

Some other Interesting Approaches to Reversible Debuggers

1. *instruction logging* : long history, including most recently gdb-7.x introduced in Oct., 2009
   - gdb-7.x : Log every instruction: When executing assembly 'store', record what was previously in the memory location; When reverse executing the 'store' command, restore previous value at memory location
   - Roughly 100,000 times slower than native speed
   - Approximately 100 bytes stored for each C statement
2. *variation of logging in Standard ML* (clever language-specific use; don't garbage collect old data; one can re-use old data via continuations when going backwards in history) : Tomach, Appel (1990)
3. *record-replay* (deterministic recording and replay of events in a virtual machine; use of remote debugger outside virtual machine) : King, Dunlap, Chen (ReVirt, 2005); Lewis, Dhamdhere, Chen (VMware, 2008); Dunlap, Lucchetti, Fetterman, Chen (ReVirt, 2008)
4. *post-mortem debuggers*: Pothier, Tanter, Piquier (Omniscient Debugger, 2007: 190 bytes/event, 500,000 events/s)

Record-Reexecute Approaches via Checkpointing

1. Feldman and Brown (IGOR, 1989): specific to C (modified compiler), custom debugger; interpreter to move past checkpoint (140 times slowdown), single-threaded
2. Netzer and Weaver (1994): instrumentation of assembly code to note reads and writes on Sparc; overhead of between 1.73 and 2.15 for non-interactive programs, single-threaded
3. Boothe (bdb, 2000): specific to C (modified compiler), record only system calls (like DMTCP), custom debugger, single-threaded
4. Srinivasan, Kandula, Andrews, Zhou (Flashback, 2004): slow multi-threaded support, no multi-core support

Novel Features of URDB:

- reverse expression watchpoint (name motivated by GDB 'watch' command and 'expression watchpoint' concept; reverse analogue of gdb *software* watchpoints, but much more efficient)
- Extension of familiar debugger (GDB, Python/pdb, 'perl -d', Matlab native debugger)

FReD for Deterministic Re-execution

FReD: Fast Reversible Debugger

(Fast due to fast checkpoint-restart by DMTCP.)

- Goal 1: Multi-core reversible debugging
  - *Multi-threaded* reversible debugging exists via virtual machines (e.g. King, Dunlap, Chen (ReVirt, 2005))
  - *Multi-core* reversible debugging on commodity hardware does not exist *(However, cf. Dunlap, Lucchetti, Fetterman, Chen (ReVirt, 2008) for an approach with customized hardware)*
  - Patil et al. (PinPlay, 2010): based on PIN instrumentation of every assembly instruction. *(very slow! doesn't take advantage of multi-core)*
- Goal 2: "Gold Standard": robust enough to debug real bugs in MySQL (highly concurrent multithreading)
- Goal 3: Memory-accurate replay (same absolute addresses seen on re-execute)

Implementation: Logging of most system calls

Northeastern UNIVERSITY

Key Point: Fast lightweight loggging of *just enough* to ensure determinism: anything stronger than that would make it too slow

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

- Logging of all system calls that touch disk: *Never touch disk on re-execute. (more on that later)*
- 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. *(Note: These are not kernel system calls, but typically calls to run-time library (libc.so).)*
- Log all interrupts (Interrupt handlers could touch disk, allocate memory, etc.): Use wrappers around creation of signal handlers (around call to *signal*) to add our own signal handlers around the user signal handlers.

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
- MySQL: variable address is part of random seed

Reversible Debugging as the Ultimate Bug Report

In order to replay a bug *deterministically*, all one needs is the checkpoint file and the log.

> **NOTE:** Even if a program manipulates large files on disk (e.g. MySQL), there is no need to include those files in the bug report. System calls are replayed from the log. So, files on disk are never used during replay.

Never Touch Disk on Re-execute

All system calls that would touch the disk are replayed directly with no access to disk. This is particularly important for disk-intensive programs such as MySQL.

Consequences:

1. The data on disk always represents the most recent version, *even though we may be travelling back in time*.
2. Checkpoint and restart are much faster. (Imagine if the database files of MySQL had to be saved at checkpoint time and restored at restart time.)
3. Re-execute can be even faster than native execution for disk-intensive programs.
4. There may still be some instances when saving and restoring database files is desirable, along with accessing disk on re-execute. (What if one wishes to stop replay-mode in the middle, and change the execution path: playing "What-if?" games.)

Work in Progress

What is the overhead of execution with logging, and of re-execute?

- **Original execution:** Fine tuning still in progress, but less than 80% *on a 16-core machine*.
- **Re-execute:** *Sometimes negative overhead in MySQL!* (re-execute is faster than original execution); This is because MySQL never touches the disk on replay.

What about copying the database files of MySQL on checkpoint/restart?

- Due to wrappers around system calls, there is no access to disk on re-execute. So, we don't really use the MySQL database files on re-execute. Hence, no need to save and restore them.

What about Firefox and Apache?

- Work in Progress

Future Work (Near Term)

SPECULATION: By the end of this decade, diagnosing *run-time errors will be almost as painless as diagnosing compile-time errors.*

- A simple memory checker based on reverse expression watchpoints (see following)
- A deadlock resolver: Deadlock is detected. When was a deadlock cycle first created (in some resource graph)?

Northeastern UNIVERSITY

Future Work: Memory Checker I

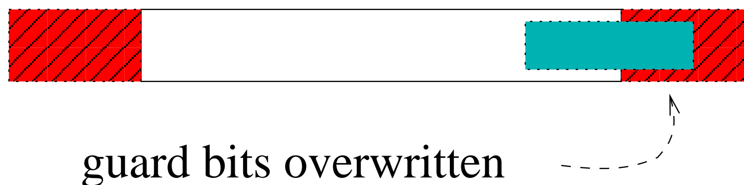
Writing past end of memory buffer: At the time of deallocation (free), the bad write is detected. When did the bad write occur?

malloc/new:

FAULT: *user overwrites guard buffer*

free/delete:

ERROR: *overwrite is discovered*



Future Work: Memory Checker II

Northeastern University

Double free: The second deallocation (free) is detected. When was the first deallocation?

malloc/new:

17

guard bits

free/delete:

FAULT: *unintended free*

42

free/delete (again):

ERROR: *previous free discovered*

91

WAS: 42 (search backwards for 42)

GENERATION

17

42

91

Future Work (Medium Term)

SPECULATION: By the end of this decade, diagnosing *run-time errors will be almost as painless as diagnosing compile-time errors.*

- Bisimulation: Many scripting languages (Python, Matlab, Javascript, etc.) have incomplete compilers. (For example, a compiler assumees a variable is an integer. The assumption is violated. The program breaks.) Bisimulation can *search backwards* for the first time where the user variables in the interpreter and compiler first diverged.
- Haskell reversible debugger: Haskell supports lazy evaluation; But a debugger that inspects the internal state will trigger lazy evaluation. This permanently changes the internal state. A reversible debugger based on checkpointing solves this.
- Reversible honey pot for malware: DMTCP may *quickly* checkpoint a virtual machine (e.g. user-space Qemu); Malware tries to hide from external observers. FReD can back up If the malware detects that it is being observed. FReD can modify external inputs (time, internet socket), and then pass the expected information to the virtual machine.
- Checking distributed invariants: When does a distributed invariant fail? (Note that DMTCP provides a ”consistent snapshot” in the sense of Lamport: freeze each process, and then take snapshot.) Error: distributed invariant has failed; Fault: when was it about to fail? (Work applies to multi-threaded, or distributed applications; Finding cause of deadlock is a simple use case.)

Future Work (Medium Term, cont.)

SPECULATION: By the end of this decade, diagnosing *run-time errors will be almost as painless as diagnosing compile-time errors.*

PROGRAM SUPPORT

- *True Exceptions:* When a program raises an exception, revert to previous program state (revert global variable values, revert open files, etc.)
- *Helpful Assert Statements:* When an assert statement triggers, call reverse-watch on the condition in the "assert" statement to find the precise point in the program when the condition was violated. *(requires running under debugger)*

Future Work (Long Term)

SPECULATION: By the end of this decade, diagnosing *run-time errors will be almost as painless as diagnosing compile-time errors.*

- *Provenance of Variable Values:* What does the variable depend on? How was this variable value created? IDEA: Go back in time to when the variable was last changed. (But there are difficulties. Suppose the variable was switching frequently between 0 and 1? Consider `if (a); x = y; else x = z;`. Does `x` depend on `y`, `z`, `a`, or all three?)

- *Algorithmic Debugging:* Algorithmic debugging is an old idea (1980s and 1990s) from functional languages. In combination with a reversible debugger, this could be revisited.
  The computation of `foo(10)` depends on the computation of `bar(20)` and `baz(100)`. The user says that the answer from `foo(10)` is wrong. So, ask the user whether the answer from `bar(20)` or `baz(100)` is wrong.

- Can some ideas from Algorithmic Debugging and from Provenance be combined to handle non-functional languages?

Ongoing Issue: Strong Determinism

- **Definition: Weak determinism:** fully synchronized programs run deterministically.
- **Definition: Strong determinism:** Unprotected access to shared variables (without a lock around the access) is also deterministic.

We don't handle strong determinism. *But*, one suggestive methodology is to detect and document all unprotected shared memory accesses (e.g. document all ad hoc thread parallelism). Then add a phantom event to be logged around each access to a shared variable, and replay deterministically from the log. Further work is needed.

Some Other Areas of Current Research

1. "Multithreaded Geant4: Semi-automatic Transformation into Scalable Thread-Parallel Software", Xin Dong, Gene Cooperman and John Apostolakis, Euro-Par 2010 (outgrowth of ten-year collaboration with the Geant4 team at CERN (high energy physics); Geant4 is a million-line program; Geant4MT (Geant4-MultiThreaded) to soon be offered in beta release)

2. Newer collaboration on MADNESS (Multiresolution ADaptive NumErical Scientific Simulation), `http://www.csm.ornl.gov/ccsg/html/projects/madness.html`, computational chemistry project led by Robert Harrison, Oak Ridge National Laboratory

   - Extension of earlier work on Roomy and Big Data:
     Roomy, `roomy.sourceforge.net`, Daniel Kunkle;
     "Solving Rubik's Cube: Disk is the New RAM" (ACM Viewpoint), Daniel Kunkle and Gene Cooperman, ACM Commmunications
   - Discussions on a distributed FReD for debugging over Infiniband clusters (DMTCP currently being extended to handle Infiniband)

QUESTIONS?

THANKS TO THE MANY STUDENTS WHO HAVE CONTRIBUTED TO DMTCP, URDB, AND FRED OVER THE LAST SEVEN YEARS:

Jason Ansel, Kapil Arya, Alex Brick, Tyler Denniston, Xin Dong, Gregory Kerr, Artem Y. Polyakov, Michael Rieker, Praveen S. Solanki, Ana-Maria Visan

QUESTIONS?