McMini and DeepDebug: Efficient Deterministic Replay of Multithreaded Bugs

Gene Cooperman
gene@ccs.neu.edu

Khoury College of Computer Sciences, Northeastern University, Boston, MA

Sept. 18, 2025

Outline of Talk

1. Part 1: Teaching Students Not To Be Afraid of Multithreaded Programs
2. Part 2: DeepDebug: In-Situ Model Checking with Long-Running Programs

Motivation for McMini: Teaching Students

We give freshman good tools for sequential programming: unit tests, input-output tests, functional tests, regression tests, debuggers.

But for multithreaded programming (the first experience of students with parallel programming), we often limit it to:

> *There are mutexes, semaphores and condition variables. Mutexes are for mutual exclusion (e.g., shared bank account: deposit/withdraw). Semaphores are for producer-consumer; Condition variables are for reader-writer programs.*

Don't depart from these three example programs or else you might have a multithreaded bug (deadlock, livelock, data race, segfault, other crashes).

McMini: Better Tools for Better Teaching

McMini (Mini-Model-Chcker):

- Free and Open Source: `https://github.com/mcminickpt/mcmini`
- Detailed Documentation: `https://mcmini-doc.readthedocs.io/`
- Catches deadlock, segfault; *and now* livelock and data races
- Easy-to-use: `mcmini ./my-multithreaded-program`
- Support for GDB debugger to replay “buggy” thread schedule

Why Teach Multithreading?

- A We live in a multicore world!
- B Multithreaded and parallel bugs are everywhere: “My Smart TV app is freezing!” – “No problem. Just turn it off and turn it on again.”
- C Developers use many-core computers to develop efficiently, and they are then surprised when users on two-core computers report bugs.

McMini example

```
void * thread_worker1(void *forks_arg) {
  for (int i = 0; i < 100; i++) {
    pthread_mutex_lock(&mutex2);
    pthread_mutex_lock(&mutex1);
    pthread_mutex_unlock(&mutex1);
    pthread_mutex_unlock(&mutex2); } }
void * thread_worker2(void *forks_arg) {
  for (int i = 0; i < 100; i++) {
    pthread_mutex_lock(&mutex1);
    pthread_mutex_lock(&mutex2);
    pthread_mutex_unlock(&mutex2);
    pthread_mutex_unlock(&mutex1);
```

McMini example: output

```
mcmini.git/mcmini --quiet -m10 ./a.out
15. thread 1: pthread_mutex_lock(mut:2)
16. thread 2: pthread_mutex_unlock(mut:1)
17. thread 2: pthread_mutex_lock(mut:1)
THREAD PENDING OPERATIONS
  thread 0: pthread_join(thr:1, _) [ Blocked ]
  thread 1: pthread_mutex_lock(mut:1) [ Blocked ]
  thread 2: pthread_mutex_lock(mut:2) [ Blocked ]
0, 0, 0, 0, 0, 1, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2,
***** Model checking completed! *****
*** DEADLOCK DETECTED ***
Number of traces: 39
```

McMini Replay in Debugger

mcmini-gdb -m15 -t'0, 0, 0, 1, 1, ' ./a.out

```
List of EXTENDED GDB commands:
  mcmini -- mcmini <TAB> : show all mcmini commands
  mcmini back -- Go back <count> transitions, by re-executin
  mcmini forward -- Execute until next transition; Accepts o
  mcmini help -- Prints help for getting started in McMini
  mcmini printPendingTransitions -- Prints the next (pending
  mcmini printTransitions -- Prints the transitions currentl
  mcmini where -- Execute where, while hiding McMini interna
(gdb)
```

Good Models for Teaching Multithreaded Programming

Resource model:

1. A mutex protects one resource: shared variable or other
2. A semaphore protects identical resources: producer slots, consumer slots, thread in a thread pool, etc.
3. A condition variable protects resources with constraint policies: no two writers at a time; priorities: writer-preferred, controller-preferred, etc.

Example programs:

- SPLASH-2 and PARSEC benchmarks
- The Little Book of Semaphores by Allen Downey: `https://greenteapress.com/wp/semaphores/`
- **We need more examples both of correct and *buggy* programs!**

McMini capabilities

SEE: `https://mcmini-doc.readthedocs.io/`

deadlock; assertion violation; segfault;

NEW: livelock; data races

Data races:

1. Compile target multithreaded program with LLVM.
2. Create an LLVM compiler plugin to interpose on access to global variables (READ/WRITE).
3. LLVM interposition calls to McMini during READ/WRITE operations
4. McMini model checker defines appropriate rules for READ/WRITE; Detects data races

Outline of Talk

1. Part 1: Teaching Students Not To Be Afraid of Multithreaded Programs

2. Part 2: DeepDebug: In-Situ Model Checking with Long-Running Programs

Combinatorial Explosion: The Achilles Heel of Model Checking

A model checkers test every thread schedule (up to isomorphism). Algorithms like DPOR (Dynamic Partial Order Reduction) prune many branches that are provably isomorphic to other branches that were tested. However, model checkers continue to suffer from combinatorial explosion. They may model a program for the first few seconds or maybe minutes, but what then?

DeepDebug is a way to get around this problem.

- DeepDebug complements the developer's existing testing strategy.
- DeepDebug does still relies on developer stress testing to find bugs.
- But DeepDebug will produce an execution trace showing how the bug occurred! (Recall the McMini execution traces.)

SOLUTION: Use Transparent Checkpointing!

The speaker has for 20 years, led a team in Transparent Checkpointing

"If you have a hammer, then everything looks like a nail."

Strategy of DeepDebug: Phase I

Phase I:

1. Run the multithreaded program under **DMTCP** (package for transparent checkpoint).
2. Checkpoint periodically: perhaps every 20 seconds
3. Upon crash, deadlock, assertion violation, *or whatever*, stop and begin Phase II.

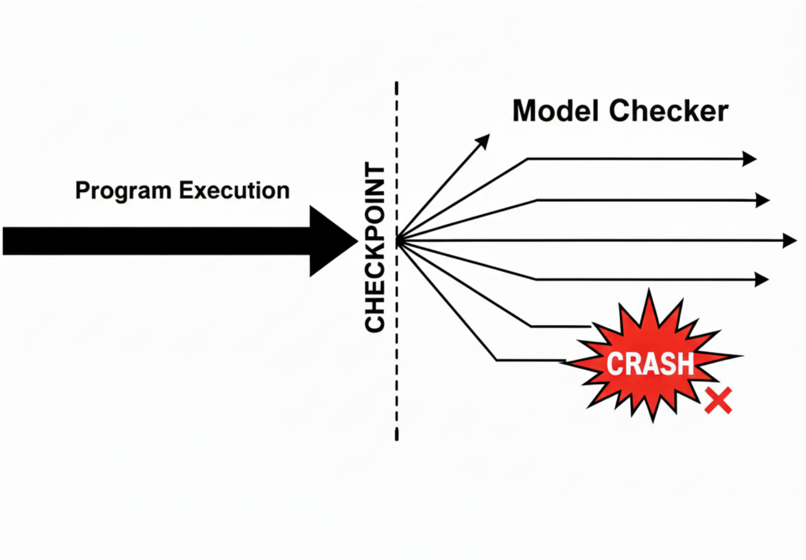
Strategy of DeepDebug: Phases II and II

Phase II:

1. Restart from the most recent checkpoint (or the one before that, if we want more context).
2. Run under a modified **McMini** (the model checker).
3. Find an execution trace that ends in crash, deadlock, assertion violation, *or whatever*.
4. Save the checkpoint file and execution trace (thread schedule) for replay debugging

Phase III: The developer traces the thread schedule with debugger.

Strategy of DeepDebug: Overview



Strategy of DeepDebug: Moral

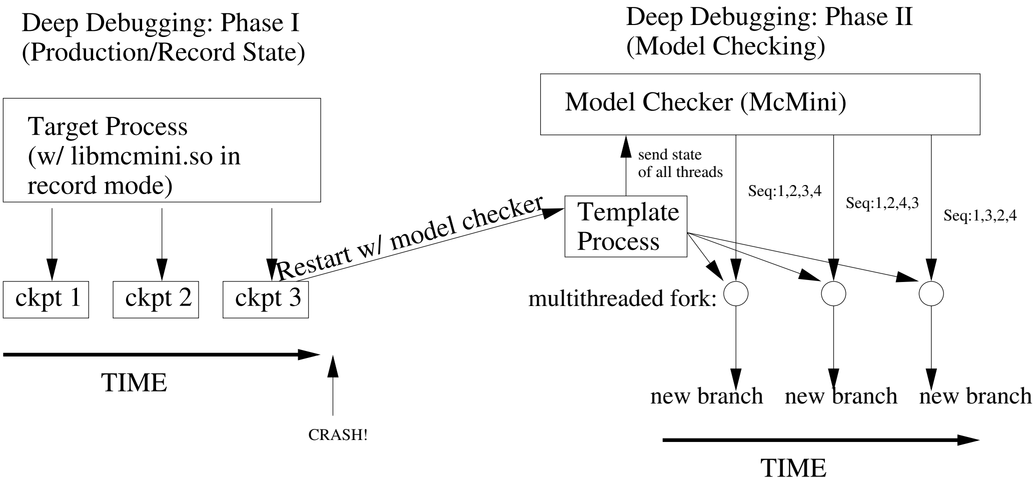
There was literature from around 2005–2015 investigating

deterministic replay

of bugs in long-running programs using logging, skeletons of the execution, etc. They tried to be faithful to the original execution of the program.

NOTE: We discard the requirement of *faithful* replay. But we do provide *deterministic replay* for *some* bug.
Fix that bug, and then come back to us if you still have another one! :-)

Internals: Overview



(Note the "multithreaded fork" for performance".)

DeepDebug: Performance

Benchmark	Native Time (s)	Phase I with Bookkeeping	Phase I with Bookkeeping +ckpt	Overhead (% (Phase I)
ABA	76.6	77.3	77.5	1.8
Dining Philosopher	77.2	78.3	79.4	2.0
Reader-Writer	74.9	75.6	76.0	1.5

ABA problem (data race: see later slide)

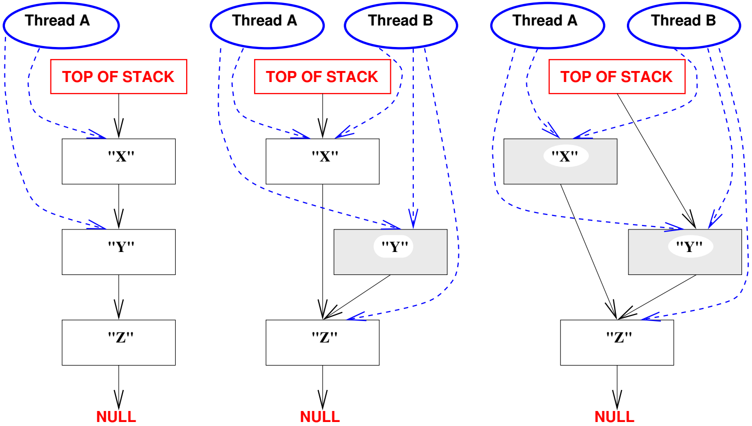
Dining Philosopher (mutex + deadlock)

Reader-Writer (condition variable with assertion failure due to bug)

DeepDebug: Performance

Benchmark	**Total # of branches in Phase II**	**Total # of buggy branches**	**branches before buggy branch**	**branches explored by McMini**
Dining Philosopher	352	283	34	612,000
ABA Problem	289	204	21	189,776
Reader-Writer	258	207	15	204,000

ABA Problem



THREAD A:
Set up for: x = pop();
(TOP OF STACK will point to "Y")

THREAD B:
x = pop();
y = pop();
push(x);

THREAD A:
Execute: x = pop();
(TOP OF STACK now points to "Y")

Questions?

QUESTIONS?