Reverse execution with persistent data structures

Omar Roth – Senior undergraduate student

Advisor: Joseph Hollingsworth – Computer Science & Software Engineering faculty member

ROSE-HULMAN
INSTITUTE OF TECHNOLOGY

Abstract/Introduction

Reversible debuggers are useful tools for developing and deploying modern applications. However, due to their high memory requirements and runtime overhead, their functionality is generally reserved for rare cases (e.g., identifying short-term memory corruption). The work included in this poster describes an alternative approach for implementing a low-overhead memory snapshotting mechanism using fully persistent data structures. Memory usage and performance analyses will be presented and compared against alternative implementations.

Background/References

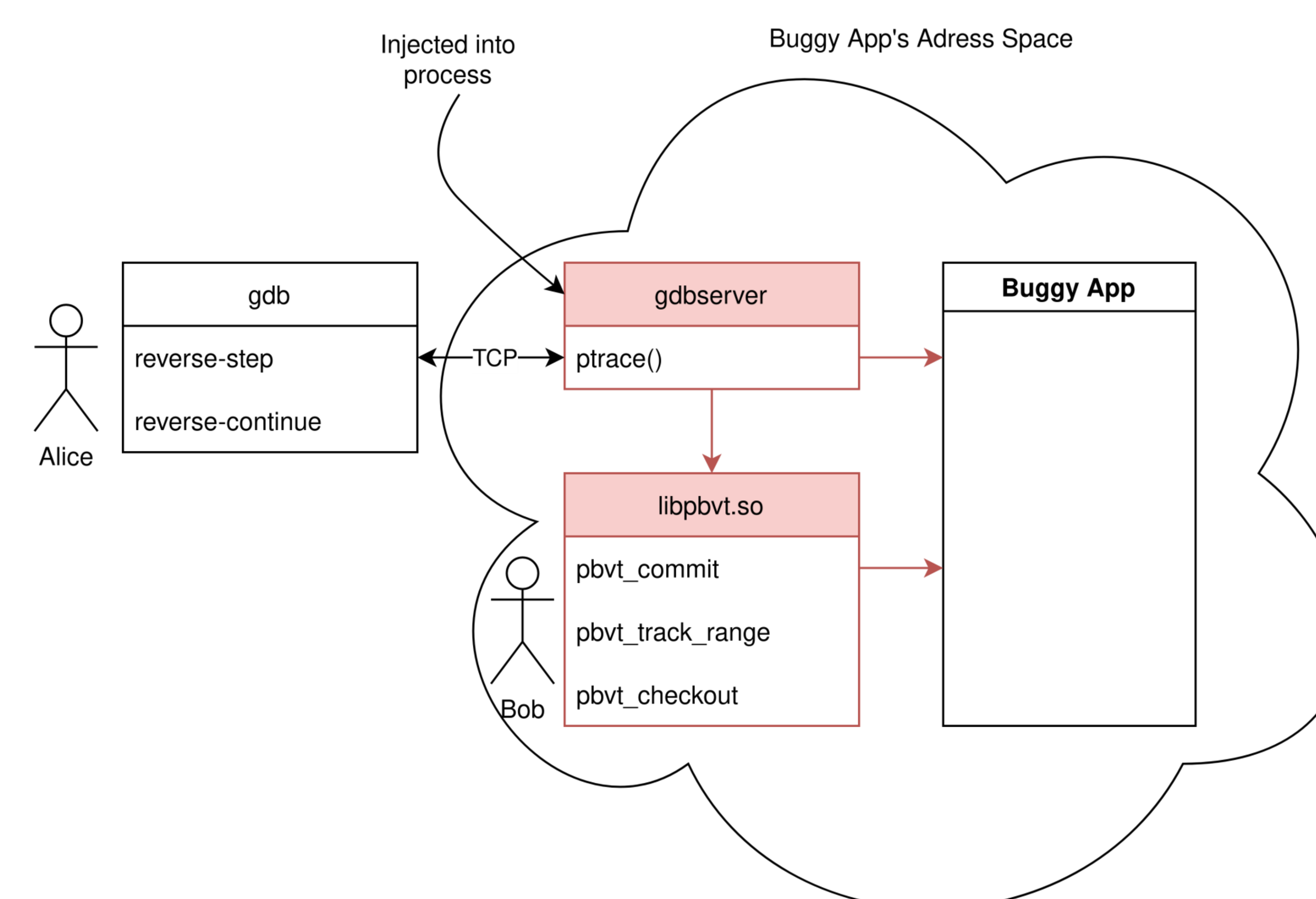
- **Fully persistent data structure**: Immutable data structure that creates and returns copy of itself when modified. All copies are stored, and each copy retains the same algorithmic complexity guarantees (i.e., older versions do not get slower).
- **Fully reversible execution**: Execution of a program that can move forwards and backwards, can be modified in a previous state, and then run forward (supports divergent re-execution).

Goal(s)/RQs

1. Can debugging overhead for Alice (a user of our reversible debugger) be pushed down to ~20% of wall-clock time while still supporting fully reversible execution for unmodified multi-threaded userspace applications?
2. Can we support fully reversible execution for Alice with less than ~20% space debugging overhead (as compared to uninstrumented execution) for long-running applications?

Methods

- Created a library (called *libpbvt*) for tracking memory, provides "version control" for arbitrary memory regions
- Created a debugger (called *rdb*) that injects into existing application, exposes a stub that connects to GDB



- Data structure of choice is persistent bit-partitioned hash trie with hash-consing. This allows us to identify duplicate sub-tries, and provides O(log32(n)) updates.
- User program writes to memory are tracked through page-faults, are "compacted" and set to a transient (writable) state until the next commit.

(a) Sample listing of memory writes, where $c_1 = c_3$.

(b) Our implementation recognizes duplicate subtrees and can compact them up to the root.

Results

- Our debugger currently supports reverse breakpoints, reverse-step, reverse-continue, as well as normal forward debugging supported by GDB
- After reverse-step, the state of the program can be modified and re-executed.
- Performance results are forthcoming and will include comparisons against other existing (but less featureful) debuggers that support reverse execution.

Future Work/Acknowledgments

- Extending our debugger to support multithreaded applications
- Integrate into QEMU: This approach would provide full-system reversible debugging
- Can our library be used for other applications besides debugging, e.g., multi-shot continuations, transactional memory, backtracking search?