Summarized Trace Indexing and Querying for Scalable Back-in-Time Debugging

Guillaume Pothier and Éric Tanter

PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile – Chile
www.pleiad.cl

Abstract. Back-in-time debuggers offer an interactive exploration interface to execution traces. However, maintaining a good level of interactivity with large execution traces is challenging. Current approaches either maintain execution traces in memory, which limits scalability, or perform exhaustive on-disk indexing, which is not efficient enough.

We present a novel scalable disk-based approach that supports efficient capture, indexing, and interactive navigation of arbitrarily large execution traces. In particular, our approach provides strong guarantees in terms of query processing time, ensuring an interactive debugging experience. The execution trace is divided into bounded-size execution blocks about which summary information is indexed. Blocks themselves are discarded, and retrieved as needed through partial deterministic replay. For querying, the index provides coarse answers at the level of execution blocks, which are then replayed to find the exact answer. Benchmarks on a prototype for Java show that the system is fast in practice, and outperforms existing back-in-time debuggers.

1 Introduction

Execution traces are a valuable aid in program understanding and debugging. Most research is centered on the capture of execution traces for offline automatic analysis [7][17][20]. However, there has been a recent surge of interest in interactive trace analysis through back-in-time, or omniscient, debuggers [5][8][9][10][11][12][13]. Such debuggers allow forward and backward stepping and can directly answer questions such as “why does variable x have value y at this point in time?”, thus greatly facilitating the analysis of causality relationships in programs.

The navigation operations provided by back-in-time debuggers are based on a small set of conceptually very simple queries. To achieve interactive navigation, those queries must execute extremely quickly, regardless of the size of the execution trace. It is therefore necessary to build and use indexes, otherwise queries would require scanning arbitrarily large portions of the execution trace. Interactive navigation in large execution traces requires an efficient indexing scheme tailored to the core set of back-in-time debugging queries:
**Bidirectional stepping.** These queries are similar to the usual stepping operations provided by traditional debuggers, with the added benefit of being able to perform them both forward and backward in time. Step into operations are very simple, as they consist in navigating to the next or previous event in the trace. Step over and step out operations, on the other hand, are more complex, as they require to skip all the events that occurred inside a method call. As the number of events to skip is potentially huge, it is not efficient to just perform a linear scan of the trace.

**Memory inspection.** Back-in-time debuggers support the inspection of the values of memory locations (such as object fields and local variables) at any point in time. To retrieve the value of a location at a particular point in time, the query to process consists in determining the last write operation to that location before the currently observed point. Again, as the last write can have happened much before the current observation point, it is not efficient to linearly scan the trace.

**Causality links.** Back-in-time debuggers support navigating via causality links, e.g. by instantly jumping to the point in time where a memory location was assigned its currently observed value. The corresponding query is actually the same as the one used to perform memory inspection: the last write operation to the location gives both the written value and the point in time at which it was written.

Interactive navigation in large execution traces is challenging: memory-based approaches allow fast navigation, but do not scale past a few hundred megabytes of trace data and therefore must discard older data [8,11]. To handle larger traces without losing information, a disk-based solution is mandatory [13], but this typically reduces the efficiency of the system. In addition, most back-in-time debuggers rely on directly capturing exhaustive executions traces [5,8,11,13]. Unfortunately, this incurs a significant runtime overhead on the debugged program, which is problematic for two reasons: (a) it makes the system less practical to use because of long execution times, and (b) the probe effect can perturb the execution enough that the behaviors to examine do not occur.

An alternative technique to avoid capturing exhaustive traces that alleviates the above issues is deterministic replay [1,3,15,16,19]. It consists in capturing only a minimal trace of non-deterministic events during the initial execution of a program. The minimal trace can then be deterministically replayed to obtain the exhaustive trace without affecting the execution of the debugged program. This is much cheaper than capturing an exhaustive trace, and thus greatly reduces the probe effect. Non-deterministic events are typically related to external inputs and system calls. However, another source of non-determinism is thread scheduling, something that is not properly supported in several deterministic replay systems.

Some deterministic replay systems support restarting the replay in the middle of the trace through snapshots that capture the state of the program at given points in time [15,16]. However these snapshots are heavyweight because they represent the full state of the heap. These snapshots can be produced efficiently by combining process forks and OS-level copy-on-write mechanisms, but they are