Reverse Execution With Constraint Solving

Raluca Sauciuc
George Necula

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2011-67
http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-67.html

May 25, 2011

Reverse Execution With Constraint Solving

Raluca Sauciuc
University of California, Berkeley
sauciuc@cs.berkeley.edu

George Necula
University of California, Berkeley
necula@cs.berkeley.edu

Abstract

The typical debugging experience is an iterative process of setting breakpoints, running the program and inspecting the state until the source of the bug is identified. When re-executing the program is not guaranteed to reproduce the faulty execution path, the final state and the system logs can be used to mentally construct a model of the execution. By contrast, debugging by going backward in time, i.e. the ability to reverse-execute the program, can be considered the most intuitive and easy-to-use alternative.

We show an efficient approach to reverse execution, which sacrifices the full replay ability in favor of portability and a lighter memory footprint. We treat the execution trace as a set of constraints, and use an SMT solver to navigate the trace and restore data values. Depending on how the solver performs on constraint sets corresponding to multiple test runs, we decide where to instrument the code to save intermediate values, for a faster replay.

1. Introduction

Consider the classic scenario of running into an exception and generating a crash report. The contents of the stack trace are somewhat informative with respect to the program path leading to the exception, but there is no information about the methods whose execution already finished. The problem is exacerbated in an event-driven application, where the stack ends at the dispatcher method which called the event handler; there is no information about the execution of previous event handlers. The stack trace can be accompanied by a partial or full heap dump, providing the values that some of the program variables held when the crash occurred. If logging was enabled, the relevant system logs can also be attached. In a nutshell, the crash report provides clues about the execution, in terms of both control- and data-flow, leaving the developer to perform detective work in order to identify the exact cause of the crash. In many cases, this collection of clues is not sufficient to pinpoint the problem.

Ideally, we would like to perform a post-mortem analysis to recreate the program state at the crash, and then to step through the execution, move both backward and forward, insert breakpoints and be able to inspect the data values at different points in the execution. Consequently, we need a systematic way to collect the clues, and an automated technique to regenerate and navigate through that particular execution. Both deterministic replay and reverse execution promise to deliver at least a subset of these goals, and provide a very intuitive approach to debugging. Deterministic replay collects program inputs as clues, while reverse execution collects many intermediate values and the exact control-flow information. Both allow navigating the execution trace, optimizing for one preferred direction.

When we consider the current practice of debugging JavaScript or mobile applications (for Android or the iPhone), we see that the manual, detective work, is still the norm. Developers look at stack traces, heap dumps, and insert a flurry of print statements throughout the code to later analyze the logs. These long-running, event-driven applications raise new challenges in the way of automated techniques. Their interface is not the low-level OS syscall, but a complex API such as the DOM, where objects can cross boundaries. They rely heavily on library code, most of which the developer has little knowledge about and cares less to trace through. They run on a variety of platforms, both in terms of software (different browsers, different third-party libraries) and hardware. Most replay systems are restricted to a subset of the system call interface, or particularly tuned toward certain sets of libraries such as MPI, or tied to a checkpointing library that works only on some platforms. Long-running, interactive applications tend to generate massive amounts of logged data, which makes reverse execution as provided by `gdb7` or UndoDB [3] unfeasible. Even the log of the last two seconds of execution might only contain unuseful information, such as the repaints triggered by a timer function in the GUI.

We advocate a lightweight application-level tracing and reverse execution technique for Java. We currently target stand-alone applications, but we plan to extend the support to Android applications in the future. We do not promise faithful replay, but offer the exact, maybe truncated, control-flow information and partial data-flow information. The trace can be stepped through both backward and forward. At each point in the trace, the developer can inspect the object graph starting from the roots, and check the values corresponding to the program state at that particular moment in the execution. For efficiency reasons, some of these values will be regenerated on-the-fly with the help of an SMT solver, and we guarantee that our replay is path-deterministic, i.e. the computation will end in the same final state. Our technique will log the necessary data in the background. When an exception is being raised, or simply when the user chooses to report an incorrect execution, the trace can be transmitted to the developer and analyzed offline.

To summarize, our contributions are as follows:

- we show how to use an SMT solver to efficiently regenerate the intermediate values on-the-fly, based on the final state; unlike other reverse execution methods, we can handle non-primitive types such as arrays and objects.
- we provide insights into what types of analyses can be used to identify which data values, if saved, can make replay more faithful, i.e. how to automatically choose the clues.
- we assess the effectiveness of CVC3, Yices and STP for our benchmarks and our approach in general; we show how an in-

strumented SAT solver can be used to guide the developer to insert state saving code, in a way that is amenable for automation.

2. Related Work

When summarizing the huge body of work related to reverse execution, we will consider the following central question: how is a destructive assignment statement `x := e` handled? The previous value of `x` can be restored from the log, or it can be recomputed either from earlier or future states. Checkpointing is an orthogonal aspect that can be used to optimize the running time or the amount of saved state.

State Saving The straightforward approach to reverse execution is to save every bit of data that is overwritten. The GNU debugger `gdb` [1] implements this technique for its Process Record target. For each executed statement, gdb saves the program counter and the previous values of the affected registers into an execution log, which can be later saved to disk. The total size of the log is limited by default at 200,000 instructions, after which it behaves like a truncating log, but it can also be set to never truncate. Because of its simplicity, this scheme works very well in practice, even in the presence of multithreading. A reverse step is instantaneous because the log provides the values to be restored. The developer has some control over the size of the execution log, which can be set to never grow too large (if, say, only the last million instructions are to be reversed). Multithreading is handled by remembering the thread switches inside the execution log; when reverse stepping, it is enough to observe that a thread switch happened and to switch the thread context and proceed with restoring older values inside this context.

In a similar vein, there have been many attempts at providing application-level reverse execution via state saving [10, 11, 13, 25]. Some of them recognize easily-invertible statements such as increments and decrements and use them for restoring previous values. The advantages of working at the source level include being able to eliminate libraries from the tracing thus saving less in the execution log. Also, a source-to-source translation can be used to derive a "reverse" program that conceptually goes through the statements backwards, by reading and restoring older values from the log. This program can be used with a regular debugger to provide the effect of reverse execution debugging. However, targeting a higher-level language such as Java makes the problem harder, since objects now have to be marshalled into the execution log. Using reflection is not enough, as some of the fields may be marked private, so hooks have to be inserted into the JVM to allow full object inspection and restoration. Only by instrumenting the JVM can thread switches be observed properly. In effect, if one doesn't want to record and replay at such a low level as gdb or the JVM bytecode, no language-level solution is general enough.

The major drawback of state saving is the amount of memory used for the execution log. A pure replay solution would save all the inputs to make the computation forward-deterministic, whereas state saving will save many more intermediate values along the way and the trace, to make the computation also backward-deterministic. The inefficiency stems from the fact that most programs are not reversible and it is not clear how much information is needed to achieve backward-determinism.

Generating Reversible Code A radically different approach to the central question is to eliminate destructive updates altogether. Motivated by the seminal papers of Bennett [8, 9], reversible languages such as Janus [34, 35] have been designed. They guarantee backward-determinism by construction and offer an efficient inverse semantics, since all the assignments and control constructs are purely reversible [36]. An `if` statement has an exit assertion, which behaves just as a conditional when going backwards (to determine which branch to take). A `for` loop has an assertion at the beginning of the loop, which becomes the test to determine when to exit the loop when reverse-executing. A procedure has to specify the values of its locals both on entry and exit, etc. It becomes evident that the burden of reverse execution has been shifted to the developer, who has to design the code to be reversible and provide all the extra assertions. There is still one precious insight here: by assuming that program inversion is a local property, syntax-driven (i.e., the inverse program mostly has the inverse of the statements, in reverse order), and by using a constraint solver or theorem prover to fill in some of the extra assertions needed, program inversion can be automated in simple cases [16, 17, 30].

Hybrid methods attempt to generate reverse code whenever possible, and revert to state saving when the problem becomes too hard. Generating the reverse code can be done either statically [4, 5] or dynamically [21]. The static approach is intraprocedural, and attempts a path-sensitive analysis to discover the data dependencies and control dependencies. The value of a variable can be restored by finding the nearest reaching definition, then recursively restoring the values it depends on. Or, in some cases, an inverse function can be used to restore values ahead of time: an assignment `x := y + 1` can be used to restore the value of `y` by applying the inverse function `f(x) = x - 1` to the current value of `x`. These assumptions only hold for simple sequential code, otherwise the control flow graph has to include all the possible interleavings, and the chances of finding the correct reaching definition are minimal. It is also not clear how well this static analysis would perform in the presence of arrays, objects and aliasing. The dynamic approach is essentially applying the same techniques over the execution trace, generating the reverse program on the fly. But keeping the reverse program in memory when tracing is expensive, and as we will show, unnecessary, since most of the work can be performed offline by a constraint solver.

Replay Deterministic replay is arguably the most memory-efficient technique: it only saves the inputs of the computation. When used for reverse execution, it is very inefficient unless coupled with checkpointing, because it has to re-execute the entire program. The log can't be truncated without resorting to checkpointing and losing portability, especially for application-level replay.

Previous library-based tools such as liblog [15] and Jockey [28] choose to interpose a uniform, low-level interface (syscalls), which results in massive amounts of data being logged, but still can't guarantee faithful replay. Irrespective of the chosen interface, replay can never be truly faithful due to the external world: file and network I/O, connections to an X server, references to state kept in other processes, etc. In fact, reverse debugging an OS requires a time-traveling virtual machine [20]. R2 [18], operating at the application level, lets developers choose the replay interface in the hope of minimizing the amount of saved state and getting a more faithful replay. Part of the burden is thus shifted into manually annotating the replay interface. Mugshot [22] chooses a narrow interface for JavaScript replay, but silently ignores the logging overheads of rich AJAX applications.

We borrow ideas from recent work, which combines constraint solving with a relaxed notion of replay [6, 24, 26] to explore equivalent executions. Techniques such as ODR [6] only save "breadcrumbs" of the state in the execution log and recover the rest offline.

Checkpointing Traditionally, checkpointing has been taking advantage of copy-on-write semantics, whether for continuation objects as in the case of the ML debugger [31] or for virtual memory pages as in the case of UndoDB [3]. Otherwise, smart incremental techniques can be applied to achieve the same effect [23, 29].

```
// positive inputs a, b
c = a + b;
d = a - b;
p = c * d;
s = c + d;
if (p == 19)
    if (s == 20)
        ERROR(p,s);
```

Figure 1. Simple Arithmetic

```
1  public static int pow(int a, int b) {
2    if (b == 0) return 1;
3    if (a == 0) return 0;
4    int newa = a * a;
5    int newb = b / 2;
6    if (b % 2 == 0)
7        return pow(newa, newb);
8    else
9        return a * pow(newa, newb);
10 }
```

Figure 2. Power Computation

(1) $a_1 = a_0$	(1) $b_1 = b_0$	(2) $b_1 \neq 0$
(3) $a_1 \neq 0$	(4) $newa_1 = a_1 * a_1$	(5) $newb_1 = b_1/2$
(6) $b_1 \% 2 = 0$	(1) $a_2 = newa_1$	(1) $b_2 = newb_1$
(2) $b_2 \neq 0$	(3) $a_2 \neq 0$	(4) $newa_2 = a_2 * a_2$
(5) $newb_2 = b_2/2$	(6) $b_2 \% 2 \neq 0$	(1) $a_3 = newa_2$
(1) $b_3 = newb_2$	(2) $b_3 = 0$	(2) $pow_1 = 1$
(9) $pow_2 = a_2 * pow_1$	(7) $pow_3 = pow_2$	(*) $pow_3 = 9$

Figure 3. Power Constraints

One important observation from [29] is that only around 10% of the executed instructions are STOREs, and knowing those values allows all the other temporaries to be recalculated. Checkpointing works well with replay, minimizing the execution time by leaving an exponentially increasing series of intervals behind the current location [12].

However, checkpoints are very brittle if performed at the OS level as most systems expect. They are not portable across machines (imagine how you would replay from a checkpoint taken on a mobile phone), so the basic, limiting assumption is that debugging happens on the same machine as testing, at the same time.

3. Traces for Reverse Execution

This section attempts to summarize our proposed approach. We first show how constraint solving can be used for reverse execution on simple code examples in Section 3.1. We discuss the motivation for using an SMT solver in Section 3.2, and how we can also use the solver to guide the saving of intermediate values in Section 3.3. Techniques for compacting the execution trace are discussed in Section 3.4.

3.1 Motivating Examples

We begin with the very simple code snippet in Figure 1. a and b are inputs, and let's assume their values are such that when we run the program the ERROR is triggered. We further assume c and d are temporaries, hence dead after the definitions of p and s, so that we only have the values of p and s as clues to why we have reached the error state. We want to recover the values of variables a, b, c and d that led the computation on this particular path.

At first, it might seem impossible to recover all these values without saving extra information, especially since there are no direct ways to invert the assignments and infer the values backwards. All previous reverse execution techniques would resort to state saving. There is however a way to recover these values, and we essentially imitate the work of a constraint solver.

We collect the constraints following the execution path, each conditional or assignment statement generating one constraint, and we get the set $c = a + b$, $d = a - b$, $p = c * d$, $s = c + d$, $p = 19$, $s = 20$. Recovering the values of a, b, c and d amounts to solving this set of constraints, and we try it by hand as follows. We start by substituting p and s by their known values, so our working set becomes $c = a + b$, $d = a - b$, $19 = c * d$, $20 = c + d$. We pick c and substitute all its occurrences by $a + b$ and get $d = a - b$, $19 = (a + b) * d$, $20 = a + b + d$, then pick d and substitute all its occurrences by $a - b$ and get $19 = (a + b) * (a - b)$, $20 = a + b + a - b$. Simplifying the last constraint, we have $20 = a + a$ from which we infer $a = 10$, so we can rewrite the other constraint as $19 = (10 + b) * (10 - b)$. We now find $b = 9$ as a solution and finally substitute the values of a and b to get $c = 19$ and $d = 1$. We have the values of all variables in the trace, including the problematic inputs.

We move to a slightly more involved example in Figure 2 computing a^b in logarithmic time. There are a few interesting aspects in this example: the use of non-linear arithmetic (notice the squaring of the first argument on both branches) and recursion. Static approaches to generating the reverse code would save values at method boundaries, effectively saving all temporary values in `pow`: `b%2, a*a, b/2`. While this simple example would benefit from an iterative implementation, generally it is very hard to motivate code refactoring and inlining for static analysis to work. The non-linear arithmetic cannot be avoided, and all existing techniques would save temporary values. There is no inverse for `b/2`, since this operation loses one bit of information. But is that bit lost? Well, no, its value can be recovered if we knew which branch on the conditional we were on. This fact is however too deep for ad-hoc techniques to discover.

Let's assume we invoke the function with $a = 3$ and $b = 2$ and it returns the value 9. We will describe the algorithm for generating the constraints in a later section, so for now we can look at Figure 3 and visually inspect how they follow the execution path for `pow(3,2)`. For each assignment statement we generate a "fresh" name for the left-hand variable. We use subscripts to keep a single-assignment property for the constraint set, and to index through the re-definitions. The original inputs are hence denoted by a_0 and b_0. pow_i denotes the return value for each invocation of the method `pow`. The number next to each constraint indicates the corresponding line number in the source code. The final constraint $pow_3 = 9$ asserts the return value is the one observed, and this is the only constraint generated from data values. Suppose we now want to inspect the execution in reverse.

We start backwards: from $pow_3 = 9$ we can infer that $pow_2 = pow_3 = 9$ by chaining the corresponding constraints, and substituting $pow_1 = 1$ we also get that $9 = 1 * a_2$. But $a_2 = newa_1 = a_1 * a_1$, and $a_1 = a_0$, so finally $9 = a_0 * a_0$.

On the other hand, we know that $b_3 = 0$, and $b_3 = newb_2 = b_2/2$ so $b_2/2 = 0$. Also, $b_2 \% 2 \neq 0$, so we can recover that $b_2 = 1$. Substituting this value, $b_2 = newb_1 = b_1/2$ so $b_1/2 = 1$. And $b_1 \% 2 = 0$ thus $b_1 = 2$. Finally $b_0 = b_1 = 2$. We are able to recover the value of b by going mostly backwards in the trace.

```
int data[] = new int[n];
//... initialize data

for (int i = 0; i < data.length; i++)
    for (int j = 0; j < i; j++)
        if (data[i] < data[j]) {
            int tmp = data[i];
            data[i] = data[j];
            data[j] = tmp;
        }
```

Figure 4. Bubble Sort

The constraint $9 = a_0 * a_0$ can be solved by bit-blasting, i.e. modeling each integer as a 32-bit vector. Most constraint solvers would then find $a_0 = 3$ hence successfully regenerating all the values on the trace. The developer can back-step through the execution and observe the correct values.

If, however, we were to save the value of a_0, by adding the constraint $a_0 = 3$ to the set, the constraint solver would have had a much easier job of simulating the execution forward: $a_1 = a_0 = 3$, then $a_2 = newa_1 = a_1 * a_1 = 9$, then $pow_2 = a_2 * 1 = 9$. This would have simply required round after round of simplifications. In effect, this is what we aim for: an effective combination of forward and backward execution, namely the simplifications and substitutions performed by the constraint solver. Some values can be easily recovered just by going backwards, while others are best recomputed from initial values.

Let's consider a final example in Figure 4, of a simple bubble-sort implementation. Intuitively, every operation performed is reversible, even the swap in the inner for loop. So if we know the final sorted array and the full trace, we should be able to determine the original, un-sorted array. Again, static or dynamic methods for generating reverse code will run into a host of problems: modeling arrays, indexing and updating their elements, and also three apparently destructive updates in the inner for loop. Of course, we can alleviate the latter by introducing a special swap function – and this is exactly what reversible languages have to provide, in order for this kind of programs to be possible to write. Nevertheless, bubble-sort would require saving a lot of intermediate values.

With our approach, there is no need to save any such values. The constraint solver will be able to forward-execute through the increments of the i's and j's (since their initial values are always 0), such that all array indexing will be made with constant values. Swaps can then be simplified away since the indices are all known, so by going backwards in the trace, we can restore the previous versions of the $data$ array, one swap at a time.

The examples we looked at are all deterministic computations, and classical replay would only save their inputs and restore everything else. Conveniently enough, the inputs are all integers, and replay would definitely perform best. On the other hand, we would require saving enough information to reconstruct the trace, and some final values, which is clearly more. This clear distinction in efficiency is blurred in the context of long-running, event-driven applications, where saving all the sources of non-determinism is still very inefficient, where inputs are much larger than simple integers and sometimes only parts of them are processed directly by the application code. Replay needs all these values to *be able to reproduce the trace*; we attempt to augment the trace with values that may or may not be those observed during the execution, but which are definitely taking the execution down the same path.

3.2 Relational Semantics

Semantically, the inversion of a destructive assignment statement can be modeled with angelic non-determinism (the refinement calculus of [32]):

$$(x := e)^{-1} = \bigvee_v (\{x = e[v/x]\}; < x := v >)$$

based on the strongest postcondition formulation:

$$sp(x := e, Q) = \exists v. x = e[v/x] \wedge Q[v/x]$$

The value of x before the assignment is a non-deterministic choice over all possible values v (note the corresponding existential quantifier in sp). The inverse statement is composed of an assumption, which terminates all executions in which the equality between the new value of x and the value of expression e when substituting x with v doesn't hold, followed by the restoration of value v in x. The state-saving method has to restore every value v from the log. The reverse-code generation looks at e and tries to determine its inverse: if it exists, then v can be inferred from x, otherwise restore v from the log just as before.

Our first insight is that the constraint $x = e[v/x]$, which appears in both formulations, doesn't specify a direction. A theorem prover or constraint solver may be able to infer v if given x, or vice-versa. Instead of choosing a fixed evaluation direction ahead-of-time, we can let the constraint solver figure out the evaluation order and the missing values. For example, given the constraint $x = y + z$, every value can be uniquely inferred given the other two. Instead of saving the old value of x before the assignment, or looking for its previous definition, we have many more options for reconstruction as the old value of x can even be inferred from one of its uses.

In a more formal setting, our technique can be described as operating over a relational semantics, which is obtained from the operational semantics (after the trace is reconstructed, the constraint generation algorithm acts as an interpreter, "executing" each statement in the trace by converting it to a constraint). As a consequence of this directionless technique, we could perform replay, i.e. regular, forward execution, if provided with just the input values. We could also compute the weakest preconditions over the trace, with respect to the final state. These can be used to generate tests. Our focus however is on how close we can get to faithful replay by saving a combination of input, intermediate and final values, and on obtaining an empirical understanding of the requirements.

A related approach to relation semantics is taken by Ross [27]. The paper uses an iterative version of the power algorithm in Figure 2 as an example to show how a Prolog-style inference can be used to invert the computation. Arguably, Prolog itself is not fixing the evaluation direction and the trace can be navigated in any way. However, there is one error in the paper, in assuming that $x' := x/2$ is invertible when knowing the value of x', aside from the vagueness of inverting assignments with Prolog. Nevertheless, an intriguing implication of the translation is that instead of using an SMT solver over the execution trace, a theorem prover could be used on the source code to prove that when instrumented to save some of the values of its variables, the program code makes it possible to properly reconstruct its inputs and obtain the reverse execution trace for all executions. We leave such insights for future work.

3.3 SMT and SAT Solvers

While manually solving the constraint sets in Section 3.1, we encountered a few hard, non-linear constraints. Naturally, a question arises of whether our technique can work on large, realistic traces, where such constraints might be frequent. To address it, we begin by looking at how these solvers are implemented.

Both SMT and SAT solvers can simulate the easily-invertible operations naturally. For SAT solvers, this basic unit of inference is called *unit propagation*: a clause becomes *unit* when all literals except one are false, and thus the remaining literal must be set to true. This can result in more unit clauses, and more literals being set to true in an iterative process. Similarly, an SMT solver will have specialized rules for each theory to discover such facts and propagate them through rounds of simplifications and substitutions. Essentially, this efficient process corresponds to a mix of forward execution when the right-hand operands are known, and backward execution when the left-hand value is known, and enough of the right-hand operands are known to make the operation invertible.

When no such inference is possible, the solver has to search by *case-splitting* and backtracking. At each step, an assumption is made, and it is followed by a round of inference. If a conflict is discovered, the assumption and all subsequent inferences are retracted, and the backtracking process continues with another assumption. Case-splitting by itself is not necessarily inefficient, if the solver is able to "guess" the correct values without backtracking. This corresponds to a value not being uniquely determined in our constraint set, making the replay approximate – we have the choice of not saving every such value in the log. However, if the solver is backtracking on a certain variable, saving its value in the log can provide not only a more faithful replay, but valuable pruning of the solver's search space.

Therefore, our second insight is that saving some data values along the execution trace helps the constraint solver in being more efficient and also, when desired, in recovering the exact values of the particular execution being debugged. We propose an iterative process of observing the execution of the solver, and inserting state-saving code to eliminate as much of the backtracking as possible. The trade-off is between memory and an exponential solving time. One extreme is to eagerly save all inputs, yielding a linear solving time equivalent to deterministic replay. The other extreme is to only save the final values, yielding an expensive weakest-precondition computation with no guarantees on how close the computed values are from the original ones. We attempt to find a more balanced choice.

3.4 Compacting the Trace

Having a compact representation of the execution trace is vital, yet orthogonal to our current focus. We plan to integrate the Ball-Larus algorithm [7] for effective tracing. In a nutshell, based on the control-flow graph of each method, a maximum spanning tree is determined and witnesses are placed over all the un-covered edges. Each method would have a separate set of witnesses. While this may sound inefficient when compared to storing one bit per conditional, it has the major advantage of being bi-directional: we can truncate the beginning of the trace yet still be able to determine the remaining part. We also hope that frequently-observed intraprocedural paths can enable some compression of the witnesses. Multithreading will introduce a complication, by saving each thread switch into the log. We have to assume a concurrent setting, such as many event-driven frameworks provide, with no full parallelism between threads, otherwise the notion of execution trace has to be redefined.

To summarize, we propose a more relaxed version of reverse execution, which combines the benefits of earlier work. Our technique operates over the execution trace, which can be truncated if running with limited memory or developer-specified constraints – such as only keeping the last million instructions. We log enough "breadcrumbs" to reconstruct the precise control flow (the execution path), and we log some data values. At replay time, we generate a set of constraints corresponding to the observed execution (or fragment of), and we restore the values of the program variables at each point in the trace. This costly operation is performed offline, and only once; the results are then available for inspection and each back-step through the trace is still instantaneous. The amount of memory consumed by the log is far less than that of typical reverse execution solutions that rely heavily on state-saving (either with or without checkpointing). Since we operate on the trace, we do not incur the imprecision of static analysis, thus being able to restore objects in the presence of aliasing. We trade-off the goal of full replay, as most typical replay systems do, for increased efficiency. By not relying on checkpointing, our solution is truly portable, as any application-level debugging mechanism should be.

4. Implementation

Our prototype implementation currently targets Java code, and uses the Soot framework [2] for instrumentation. In the first pass the code is instrumented, adding hooks into a library which handles the constraint generation. For simplicity, we currently generate the constraints in tandem with code execution, but of course a more efficient implementation would only save breadcrumbs for the compacted trace and perform the latter task offline. The set of constraints is then solved (we know it is satisfiable), and the model obtained represents the valuation of all variables of interest. We keep line number information in the set of constraints, so a debugger-like tool can walk the trace and allow the user to inspect the values at each line of code in the source.

Our technique works at the application level, leaving the developer in control over which classes are instrumented, i.e. delimiting the interface to library code. A consequence of not instrumenting a particular class is that the objects of that type will be treated as "opaque", there won't be any constraints on their fields in the trace, and the trace will not include any of their executed methods. We have observed in practice that many of the objects created by library code are not meant to be inspected by the caller: sockets, file handles, GUI objects, etc. can all be inspected with a debugger such as `jdb`, but their private fields rarely help pinpoint a problem in the application code. Unless the developer has very good knowledge of their internals, libraries can be omitted from tracing. We also note that when running with memory constraints, omitting library code is essential in keeping the trace compact and still useful. For example, in a simple event-driven application, if we decide to only recover the last million instructions, we might discover that most of them are part of the refresh/repaint cycles triggered automatically by the UI library, while the history of the useful event-handlers has been lost! In such cases, we would probably want the trace to disregard the UI library altogether.

4.1 Constraint Solvers

A major decision in the design process was the choice of an SMT solver. We decided to target three of them, in order to see how well they perform on our benchmarks. We didn't want to fine-tune our constraints to the heuristics of a particular solver right from the start. Most solvers handle bitvector arithmetic, even the non-linear cases (since precision is finite). We are not handling floating-point arithmetic, and we currently don't encode strings – although we could integrate existing decision procedures such as [19] for STP.

CVC3 and Yices have a similar architecture, broadcasting newly-discovered facts between the many built-in theories, simplifying, backtracking and case-splitting as necessary. STP only provides the theory of bitvectors and arrays, and its approach consists of a first phase of simplifications and linear solving, followed by bit-blasting and SAT solving. Even though CVC3 is not as competitive as the others, since we didn't have the source code of Yices, in the cases when it timed out on simple examples we ran CVC3 with logging enabled to figure out the source of non-linear behavior.

For STP, we instrumented the code to see exactly how the backtracking worked and which bits of the input were chosen for case-splitting. We also look at what the other choices were on each level, since MiniSAT [14] uses heuristics to assign SAT variables involved in recent conflicts a higher priority and its backtracking is conflict-driven. This means eagerly saving the first bits chosen for branching might not yield the best results, since the solver "learns" at a later time a more profitable set of bits to explore.

We generate constraints for all three targets, following the same algorithm. We take advantage of the typed world of Yices and CVC3 and map objects to records, whereas for STP we erase types and encode everything as arrays of bitvectors.

4.2 Generating Constraints

We first collect typing information from the instrumented classes. Each class is mapped to a user-defined type, based on the fields it declares. For each instrumented class C that declares instance fields $f_1, f_2, \ldots, f_n$ and static fields $g_1, g_2, \ldots, g_m$, we assign two record types. First, the instance type has the following signature (in CVC language):

$$C : TYPE = [\#id : INT, class : INT, f_1 :< Type(f_1) >,$$

$$f_2 :< Type(f_2) >, \ldots, f_n :< Type(f_n) > \#]$$

Second, the class type has the following signature:

$$C\_class : TYPE = [\#id : INT, g_1 :< Type(g_1) >,$$

$$g_2 :< Type(g_2) >, \ldots, g_m :< Type(g_m) > \#]$$

The meaning of $Type(f_k)$ is the following. If f_k is a primitive type, such as integer or byte, then $Type(f_k)$ is the respective type; otherwise, the signature uses INT to denote a reference to another object. Primitive types are encoded as bitvectors based on their length, so for instance an `int` in Java would be mapped to the $integer$ type:

$$integer : TYPE = BITVECTOR(32);$$

Each object has a unique id, which is given when the object is newly allocated. The fact that a field f_k refers to another object is therefore encoded as f_k having the type INT and the associated record referring to the id of that object. This allows the referred object to be functionally updated without the referrer losing its reference to it. Each class is also given a unique id, and an object refers to its class through its $class$ field. We must represent the class instances due to static fields.

Arrays are also encoded as objects, meaning each instance has its own id. The $length$ field and indexes are represented as bitvectors. For instance, an array of type $C[]$ would have the following signature, whatever the definition of C is:

$$C\_V : TYPE = [\#id : INT, length : integer,$$

$$data : (ARRAY\ integer\ OFC)\#];$$

Uninstrumented classes give rise to opaque definitions, which only have id and $class$ fields.

Having collected the type information, we pass through the trace and emit constraints. We act as an interpreter, keeping an internal stack of symbol tables to be able to uniquely identify each lvalue. This is because each assignment has to generate a "fresh" name for its lvalue while also correctly referring to any rvalues. A fresh name is obtained by appending the current value of the counter for that name. The rvalues are looked for in the symbol table for the current method. We also have to emit a type definition for each fresh lvalue. For method calls, we model the parameter passing through a stack, the caller pushing the actual arguments on the stack and the callee popping them. Returns are handled similarly, the callee pushing the returned value on the stack and the caller popping it. Library calls, i.e. calls for which there is no corresponding callee, are treated as uninterpreted functions with CVC3 and Yices.

$x = y$	$x' = y$
$x = y\ op\ z$	$x' = y\ op\ z$
$x = y.f$	$\begin{cases} x' = select(y, f) & \text{y.f primitive} \\ select(x', id) = select(y, f) & \text{otherwise} \end{cases}$
$x.f = y$	$\begin{cases} x' = update(x, f, y) & \text{y primitive} \\ x' = update(x, f, select(y, id)) & \text{otherwise} \end{cases}$
$x = y[i]$	$\begin{cases} x' = select(y, i) & \text{y[i] primitive} \\ select(x', id) = select(y, i) & \text{otherwise} \end{cases}$
$x[i] = y$	$\begin{cases} x' = update(x, i, y) & \text{y primitive} \\ x' = update(x, i, select(y, id)) & \text{otherwise} \end{cases}$

Figure 5. Translation Rules for CVC

$x = y$	$x' = y$
$x = y\ op\ z$	$x' = y\ op\ z$
$x = y.f$	$x' = select(M_{Class(y)}, y, f)$
$x.f = y$	$M'_{Class(x)} = update(M_{Class(x)}, x', f, y)$
$x = y[i]$	$x' = select(M_{Class(y)}, y, i)$
$x[i] = y$	$M'_{Class(x)} = update(M_{Class(x)}, x', i, y)$

Figure 6. Translation Rules for STP

Figure 5 provides an overview of the translation, with the most interesting cases. For simplicity, instead of showing the full CVC notation, we used the classic select/update functions from the theories of arrays and records. x' denotes a fresh version of x. The type information is used to determine whether we are writing a primitive value or just a reference into an object field or array element. To maintain soundness in the presence of aliasing, we add axioms for each type:

$$FORALL(o1 : C, o2 : C) : o1.id = o2.id => o1 = o2;$$

Unlike Yices and CVC3, STP only offers arrays of bitvectors. In this case, each object is denoted by a numerical identifier, and for each type we represent the memory as a two-dimensional array indexed first by the numerical identifier and then by the field f. More specifically, for each type C:

$$M_C : ARRAY\ BITVECTOR(32)\ OF$$

$$BITVECTOR(< Width(C) >);$$

where the object id's are 32-bit integers and $Width(C)$ denotes the number of bits required to represent the type C, packed as a concatenatation of its instance fields. We have to perform an extra step of assigning each field to an integer range of bits, such that field f of object o is denoted by the bit sequence $b_f..b_{f+Width(f)}$ of $M_C[o]$. Arrays are represented similarly. Figure 6 summarizes our translation for STP; for clarity, we hide the details of bit extraction and concatenation and use the two-dimensional array concept.

Example Let's consider the code snippet in Figure 7. We construct a linked list with two elements, o_1 and o_2, link them together, then update the value in o_2. Figure 8 shows the output of our prototype in CVC format. Notice that each assignment generates a fresh name for o_1 or o_2, however the identity of the objects is not lost: the object pointed to by o_1 has $id = 2$, and the object pointed to by o_2 has $id = 3$, and there are only 2 Node objects throughout the execution. If we ask for the satisfying model, the actual output is tabulated in Figure 9. Suppose we want to go back just one step, revert the last assignment, and inspect the state from o_1. At that time o_1, in its latest revision as $o1\_\_2$, was pointing to an object (with

```
class Node {
    public int value;
    public Node next;
}

Node o1 = new Node();
o1.value = 5;
Node o2 = new Node();
o2.value = 13;
o2.next = null;
o1.next = o2;

o2.value = 14;
```

Figure 7. List Example

$id = 2$) having the value field of 5, and the next field pointing to an object with $id = 3$. Looking at the most recent version of all the records for $id = 3$, we find an old o_2 which had the value field of 13.

In the general setting, the SMT solver's output can be parsed by a debugger tool which can reconstruct the history of all the program states, similarly to the demonic memory concept from [33]. The most useful part is the automatic reconstruction of the object graphs, such that at each point in time the developer can navigate any reachable objects and see the correct values. Also, even though the SMT solvers can't provide an estimate to the number of models available, we can use a little trick to determine whether each recovered value is unique or not: simply assert it can't be equal to the proposed value, check for satisfiability again, then retract the assertion. The debugger can then mark which values are identical to those from the original run, and which values might be off (unconstrained). Moreover, a notion similar to coverage can be defined, so that the developer can consider whether having the exact copy of a variable is worth saving extra data in the trace, or it can be left as a best-effort guess.

5. Evaluation

We start by evaluating our prototype implementation against a suite of micro-benchmarks, all of which require traditional reverse execution implementations to save intermediate values in addition to the execution trace. fact_10 represents the factorial computation for $n = 10$. fib_30_iter computes the 30th Fibonacci number iteratively and fib_10_rec computes the 10th Fibonacci number recursively. list_5 is creating a sorted linked list of length 5 and inserting another element into the list while maintaining the sorted property. bubble_<n> represents a bubble sort for an array of length n. Finally, pow_<x>_<y> computes x^y as shown in the first code example. All of these examples are invertible without requiring any saved data. We provide the final values and check that the constraint solvers are able to recover the inputs and all the intermediate values appearing in the traces.

A final benchmark that we considered, snake, is more closely related to the long-running, event-driven applications of today. We implemented a simplified version of the Snake game available on most mobile platforms. In a nutshell, the game is played on a grid, the snake occupying a contiguous set of cells and having one end designated as the "head" and an initial direction set to South. The user can change the direction by pressing a key. At each time tick, the snake advances in the current direction, by adding the adjacent cell to its body and removing the cell corresponding to its tail. The game is lost when the snake collides with itself or tries to go out of the grid. In our version, all the events are stubbed out as inputs, and we omitted the scoring. We tested the code on various

```
integer : TYPE = BITVECTOR(32);
Node : TYPE = [# id: INT, class: INT,
  value: integer, next: INT #];
Node_class : TYPE = [# id: INT #];

ASSERT FORALL (o1:Node, o2:Node):
  o1.id = o2.id => o1 = o2;

Node__class__0: Node_class;
ASSERT Node__class__0 = (# id := 1 #);
o1__0: Node;
ASSERT o1__0.class = 1;
ASSERT o1__0.id = 2;
o1__1: Node;
ASSERT o1__1 = o1__0 WITH .value := 0hex00000005;
o2__0: Node;
ASSERT o2__0.class = 1;
ASSERT o2__0.id = 3;
o2__1: Node;
ASSERT o2__1 = o2__0 WITH .value := 0hex0000000d;
o2__2: Node;
ASSERT o2__2 = o2__1 WITH .next := 0;
o1__2: Node;
ASSERT o1__2 = o1__1 WITH .next := o2__2.id;
o2__3: Node;
ASSERT o2__3 = o2__2 WITH .value := 0hex0000000e;

QUERY FALSE;
COUNTERMODEL;
```

Figure 8. List Example Constraints

Name	id	class	value	next
o1__0	2	1	0	0
o1__1	2	1	5	0
o2__0	3	1	0	0
o2__2	3	1	13	0
o1__2	2	1	5	3
o2__3	3	1	14	0
Node__class	1	-	-	-

Figure 9. List Example Output

sequences of events, and the table shows three sample runs for event sequences of length 6, 10 and 45. More interesting, we were able to recover the intermediate values, as STP managed to simplify away the formula even before invoking the SAT solver. Hence the running times account for the linear solving phase and printing the satisfying model. This is encouraging, as it confirms the importance of the final state in recovering previous states, especially for long-running applications such as games which "accumulate" the inputs.

All the tests were run on a 64-bit Intel Core 2 Quad machine with 4Gb of RAM. We used Yices 1.0.29 and we checked out the latest revisions of CVC3 and STP. With the exception of snake, we ran the benchmarks on all three constraint solvers, in order to assess their effectiveness with respect to our approach. Table 1 summarizes the results, with the running times for each solver, averaged over three runs.

We show the approximate number of constraints generated in each case (there are minor differences for each solver), and the number of constraints corresponding to branches. Effectively the first number represents the length of the trace, while the second represents an estimate of the amount of memory required for the log, using a compact yet naive representation which would take 1 bit of storage per branch. In our benchmarks the percentage of

branches over all instructions varies between 20% and 40%, so without compression, a rough estimate of the number of bytes used to encode a trace of N statements would be $N/20$. We see two opportunities to improve this result: using compression over the Ball-Larus method of encoding the trace, as well as trying to identify parts of the trace for which control-flow can be fully determined from data values that are either saved or restored by the constraint solver.

We made a distinction between the iterative and recursive implementations for Fibonacci as the latter stresses out the importance of an interprocedural technique. The recursive implementation computes a lot of redundant data which is saved naively by `gdb`, ending up with 6070 instructions in its recording buffer and about 6Mb of (unnecessarily) saved data. But as we can see from the table, all the constraint solvers handle both cases with ease. We marked the running time for CVC3 with an asterisk because in those particular runs, CVC3 had trouble generating the full model, and we could only get a counterexample omitting some of the intermediate values. We suspect this behavior is due to a bug in CVC3, since it could handle bigger and more complex examples. The factorial computation and the list insertion run smoothly as all the necessary data is already in the trace and no case-splitting is required.

The pow benchmarks show that CVC3 is hopelessly inefficient with bitvector arithmetic, especially when case-splitting is required. The bubble-sort benchmarks also take Yices out of the competition, as it times out performing case-splitting when having to invert 25 or 100 swaps of array elements. This is clearly an inefficiency stemming from not performing enough simplifications with the theory of arrays. STP is so fast because it manages to simplify away all the constraints in its first phase, such that the SAT solver is always given the formula "TRUE". In effect, STP is the only reasonable choice for us, since we rely so heavily on simplifying away many operations on objects and arrays.

5.1 Case Study: Smoothing

Having checked that we are able to reverse-execute when all the data values can be recovered from the trace and the final state, we move to a more interesting example which requires some amount of state saving. Our intuition is that the decision for saving a particular value should be based on the behavior of the constraint solver, i.e. we should probably save those values which entail case-splitting.

Let's consider a simplified "smoothing" operation over an array, such that at each iteration, the value of the current element is computed as the average of its neighbors' values (integer division). This is of course an over-simplification of many image-processing algorithms using convolution kernel masks, in which each output pixel is altered by contributions from a number of adjoining input pixels. At each iteration, data is lost as these operations are not fully reversible.

Figure 10 shows the snippet of code that performs the smoothing in m iterations over an array of size n. We ran the instrumented code over a series of tests, varying both m and n, with random data in the array. For each test, we looked at the bits picked by the SAT solver. Figure 11 attempts to summarize these results, where $a[i]$ denotes an input and the subscripts denote the bits chosen.

Interestingly enough, depending on how many iterations are run, the last m bits of the initial values are usually chosen. This corresponds to the intuition that at each iteration we lose the least significant bit of each value due to the integer division. The table also shows some higher-order bits; it may well be the case that the SAT solver had to choose from a set of equally optimal splits. We believe this is case because we manually added the last m bits of the inputs to the traces, and verified that the solutions are identical to the original inputs and no case-splitting is performed. If later we run the code and observe weird final values, we can trace back

```
for (int k = 0; k < m; k++) {
  for (int i = 0; i < n; i++) {
    int sum = 0;
    if (i > 0) sum += a[i-1];
    if (i < n-1) sum += a[i+1];
    b[i] = sum / 2;
  }
  for (int i = 0; i < n; i++)
    a[i] = b[i];
}
```

Figure 10. Smoothing Kernel

m	n	bits
1	2	$a[1]_0$
1	4	$a[3]_2, a[2]_0, a[1]_0$
1	6	$a[5]_{31:10,4:0}, a[4]_0, a[3]_0, a[2]_0, a[1]_0$
2	2	$a[1]_{1:0}, a[0]_{1:0}$
2	4	$a[3]_{1:0}, a[2]_{1:0}, a[1]_{3,1}$
2	6	$a[5], a[4]_{2:1}, a[3]_4, a[2]_1, a[1]_{5:4}, a[0]_0$
3	2	$a[1]_{2:0}, a[0]_{1:0}$
3	4	$a[3]_{5,3,1:0}, a[2]_{23:4,0}$

Figure 11. Chosen Bits

the computation and figure out a subtle overflow for the addition operations on variable sum.

This example suggests a possible technique for optimizing the amount of saved state, when we want to reconstruct as much of the original values as possible. By saving more data, we are constraining the generated model to be closer to the original execution. The developer can run the code against a test suite, and collect a profile of the input bits which are most frequently chosen for case-splitting. Then the code can be instrumented to save some of these bits. This process might be iterative (as any optimization), since it is not clear whether the profiling data is painting a complete picture of what needs to be saved. But the results are encouraging. In our example, any local analysis would save the lost bit in each division and one of the summands, to make the operations fully reversible. Saving just the last m bits of the inputs is a very deep fact. Essentially, no bit needs to be saved unless it is derived from an input bit, and the SAT solver can answer this question.

5.2 Case Study: Hashing

For our next experiment, we implemented the Rabin-Karp rolling hash function:

$$H = (a_0 * b^k + \ldots + a_{k-1} * b + a_k)\%n$$

where $a_0, \ldots, a_k$ are input values, and b and n are prime. We chose this type of computation to get more insights into the trade-offs between saving more values for an accurate replay and minimizing the constraint solving time. We consider a random set of four integers a_3, a_2, a_1, a_0 as input, and iteratively compute the hash H_4, where H_i denotes the hash value after iteration i:

$$H_0 = 0$$

$$H_i = (H_{i-1} * b + a_i)\%n$$

First, we attempt to reverse-execute by using only the final hash value. As expected, the solution we get is the trivial one, setting all inputs to 0 except the final one which is set to H. The behavior of the solver is almost linear, making 126 case splits but not backtracking even once – because all the decisions were setting input bits to 0. Unsatisfied with the solution, we now provide one of

Benchmark	Constraints	Branches	CVC3	Yices	STP
fact_10	40	10	0.61s	0.015s	0.022s
fib_30_iter	157	32	1.76s*	0.04s	0.01s
fib_10_rec	681	322	1.50s	0.27s	0.03s
list_5	44	12	0.4s*	0.04s	0.09s
pow_3_3	17	7	2.79s	0.050s	0.11s
pow_5_13	30	13	19.58s	0.115s	0.28s
pow_2_30	37	16	20.55s	0.21s	0.34s
bubble_3	44	21	0.55s	0.55s	0.005s
bubble_5	177	43	1.44s	Timeout	0.013s
bubble_10	457	133	4.63s	Timeout	0.02s
snake_6	400	81	-	-	0.021s
snake_10	563	145	-	-	0.031s
snake_45	2354	979	-	-	0.111s

Table 1. Benchmarks

Inputs	Case Splits	Input Splits	Conflicts	Time
-	126	126	0	0.053s
a_3	265	96	38	0.054s
a_2, a_3	2757	84	701	0.123s
a_1, a_2, a_3	5202	41	1292	0.207s
$a_0, \ldots, a_3$	2	0	0	0.006s
H_3	151	123	11	0.052s
H_3, H_2	967	115	73	0.063s
H_3, H_2, H_1	160	97	13	0.048s
H_3, H_2, H_1, a_0	147	71	15	0.038s

Table 2. SAT Solver Stats

the input values to the solver, the first 32 bits used for case splitting earlier, to further constrain the search space.

Table 2 summarizes this process. 'Inputs' denotes what values (32-bit integers) we provide to the solver besides H, 'Case Splits' counts the total number of decisions while 'Input Splits' only counts decisions made on bits belonging to the input values, 'Conflicts' counts the number of times backtracking was involved in the search. When varying the amount of saved input values from $a_0, \ldots, a_3$ we observe an exponential blow-up in solving time, corresponding to our constraints becoming harder and more backtracking being part of the search. However we were not able to recreate the original set of inputs even when providing all-but-one of them, which means that for a perfectly accurate replay in this case, we would have to save all the inputs. The fifth line in the table corresponds to this instance of forward-execution, in which the solver simplifies everything away by unit propagation. In effect, unless we want this perfect replay, there is little value in saving one or more of the input values, since we don't get any closer to the original inputs and only burden the solver. Our reasoning can be automated by trying combinations of saved inputs and observing the solver's (degrading) performance, until finally choosing a local minimum, such as not saving any input values.

We were intrigued by the fact that most of the case splits were not performed on input bits. MiniSAT uses a clause learning strategy in which for each detected conflict a new clause forbidding the conflicting literal assignment is added to the set, and all the variables involved are promoted for case splits. This means bits holding intermediate values in the computation can be more useful than input bits, especially to the solver's performance. One problem is that not all bits from the SAT solver have a corresponding name in the execution trace. Currently STP does not propagate the necessary mapping information to the SAT solver. However, we guessed that the intermediate H_i's would be good candidates, and sent their values to the solver as well. As can be seen from the table, the performance is much better, although we still can't get the original input values. In this example, no other 4-value set except $a_0, \ldots, a_3$ can.

We tested this assumption also on the code in Figure 1. Without providing any additional values, there are 1653 case splits and 313 conflicts. When providing the value of b we get 2 conflicts and 77 case splits – b is not that useful, the first 29 case splits are for a's bits. When providing the value of a we get 1 conflict and 31 case splits corresponding to the bits of b. However, for either c or d no conflicts occur and the solver exhibits a linear behavior. In this case, the temporary values are worth more than the input values in terms of savings, especially since a and b can be linearly solved from c and d.

To summarize the findings of this study, we envision an automated technique to minimize the solver's running time by iteratively trying combinations of values to save. Simply trying to eliminate backtracking as soon as possible might fail to converge, as in the previous case study – the last m bits weren't necessarily picked first when case-splitting, however they were sufficient. Another obstacle is the loss of source-code information in the SAT solver, which combined with heuristics such as MiniSAT's dynamic variable ordering (based on learned conflicts) can result in case-splitting and backtracking that provides little guidance for our technique.

6. Conclusions and Future Work

We have described a lightweight and flexible technique for reverse execution. We operate at the application level, and only trace through classes designated by the developer, ignoring libraries. We generate a lot less data in the log, and are able to trace through more user code, when having a hard limit on storage space. We are not platform-dependent as we do not generate checkpoints, and we can replay the code in isolation, without repeating the calls to problematic, non-idempotent library functions. The precision of our method can be increased by adding more data values to the trace; we have a basic mechanism for marking which values are recovered exactly, and a profiling session can inform the developer of how faithful the replay is overall. A worst-case scenario is to save all the input values, rendering our technique equivalent to deterministic replay.

Future work will explore how to automatically drive the code instrumentation to perform more state saving, by closing a feedback loop around the process we manually attempted in the case studies. We plan to modify STP to keep detailed information about the mapping between the variables in the trace and the bits on which the SAT solver is backtracking. We also plan to implement the trace minimization technique from [7] and to explore the interplay between control-flow and data-flow information.

References

[1] http://sourceware.org/gdb/wiki/ReverseDebug.

[2] http://www.sable.mcgill.ca/soot/.

[3] http://undo-software.com/.

[4] T. Akgul and V. J. Mooney III. Assembly instruction level reverse execution for debugging. *ACM Transactions on Software Engineering and Methodology*, 13:149–198, April 2004.

[5] T. Akgul, V. J. Mooney III, and S. Pande. A fast assembly level reverse execution method via dynamic slicing. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 522–531, 2004.

[6] G. Altekar and I. Stoica. Odr: output-deterministic replay for multi-core debugging. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 193–206, 2009. ISBN 978-1-60558-752-3.

[7] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Trans. Program. Lang. Syst.*, 16:1319–1360, July 1994.

[8] C. H. Bennett. Logical reversibility of computation. *Ibm Journal of Research and Development*, 17:525–532, 1973.

[9] C. H. Bennett. Time/space trade-offs for reversible computation. *Siam Journal on Computing*, 18:766–776, 1989.

[10] B. Biswas and R. Mall. Reverse execution of programs. *SIGPLAN Not.*, 34:61–69, April 1999.

[11] S. P. Booth and S. B. Jones. Walk backwards to happiness - debugging by time travel. In *in Automated and Algorithmic Debugging*, pages 171–183, 1997.

[12] B. Boothe. Efficient algorithms for bidirectional debugging. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 299–310, 2000.

[13] J. J. Cook. Reverse execution of java bytecode. *The Computer Journal*, 45:(6)608–619, 2002.

[14] N. Eén and N. Sörensson. An Extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *Theory and Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes in Computer Science*, chapter 37, pages 333–336. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2004.

[15] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay debugging for distributed applications. In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, pages 27–27, 2006.

[16] R. Glück and M. Kawabe. A program inverter for a functional language with equality and constructors. In *APLAS '03*, pages 246–264, 2003.

[17] R. Glück and M. Kawabe. Revisiting an automatic program inverter for lisp. *SIGPLAN Not.*, 40:8–17, May 2005.

[18] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: an application-level kernel for record and replay. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 193–208, 2008.

[19] A. Kiezun, V. Ganesh, P. J. Guo, M. D. Ernst, P. Hooimeijer, V. Ganesh, P. J. Guo, and M. D. Ernst. Hampi: A solver for string constraints. In *In International Symposium on Software Testing and Analysis*, 2009.

[20] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 1–1, 2005.

[21] J. Lee. Dynamic reverse code generation for backward execution. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 174:37–54, May 2007.

[22] J. Mickens, J. Elson, and J. Howell. Mugshot: deterministic capture and replay for javascript applications. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 11–11, 2010.

[23] R. H. B. Netzer and M. H. Weaver. Optimal tracing and incremental reexecution for debugging long-running programs. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, PLDI '94, pages 313–325, 1994.

[24] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. Pres: probabilistic replay with execution sketching on multiprocessors. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 177–192, 2009.

[25] K. S. Perumalla and R. M. Fujimoto. Source-code transformations for efficient reversibility. *Technical Report GIT-CC-99-21, College of Computing, Georgia Tech*, 1999.

[26] J. Pool, I. S. K. Wong, and D. Lie. Relaxed determinism: making redundant execution on multiprocessors practical. In *HOTOS'07: Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, pages 1–6, 2007.

[27] B. J. Ross. Running programs backwards: The logical inversion of imperative computation. *Formal Aspects of Computing*, 9:331–348, 1998.

[28] Y. Saito. Jockey: a user-space library for record-replay debugging. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging*, AADEBUG'05, pages 69–76, 2005. ISBN 1-59593-050-7.

[29] R. Sosič. History cache: hardware support for reverse execution. *SIGARCH Comput. Archit. News*, 22:11–18, December 1994.

[30] S. Srivastava, S. Gulwani, J. S. Foster, and S. Chaudhuri. Path-based Inductive Synthesis for Program Inversion. *PLDI '11: Proceedings of the 2011 ACM SIGPLAN conference on Programming language design and implementation*, 2011.

[31] A. P. Tolmach and A. W. Appel. Debugging standard ml without reverse engineering. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, LFP '90, pages 1–12, 1990.

[32] J. von Wright. Program inversion in the refinement calculus. *Information Processing letters*, 37, 1990.

[33] P. R. Wilson and T. G. Moher. Demonic memory for process histories. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, PLDI '89, pages 330–343, 1989.

[34] T. Yokoyama. Reversible computation and reversible programming languages. *Electronic Notes in Theoretical Computer Science*, 253(6): 71 – 81, 2010.

[35] T. Yokoyama, H. B. Axelsen, and R. Glück. Principles of a reversible programming language. In *Proceedings of the 5th conference on Computing frontiers*, CF '08, pages 43–54, 2008.

[36] T. Yokoyama, H. B. Axelsen, and R. Glück. Reversible flowchart languages and the structured reversible program theorem. In *Proceedings of the 35th international colloquium on Automata, Languages and Programming, Part II*, ICALP '08, pages 258–270. Springer-Verlag, 2008.