A Case for Dynamic Reverse-code Generation to Debug Non-deterministic Programs

Jooyong Yi
School of Computing
National University of Singapore
jooyong@comp.nus.edu.sg

Backtracking (i.e., reverse execution) helps the user of a debugger to naturally think backwards along the execution path of a program, and thinking backwards makes it easy to locate the origin of a bug. So far backtracking has been implemented mostly by state saving or by checkpointing. These implementations, however, inherently do not scale. Meanwhile, a more recent backtracking method based on reverse-code generation seems promising because executing reverse code can restore the previous states of a program without state saving. In the literature, there can be found two methods that generate reverse code: (a) static reverse-code generation that pre-generates reverse code through static analysis before starting a debugging session, and (b) dynamic reverse-code generation that generates reverse code by applying dynamic analysis on the fly during a debugging session. In particular, we espoused the latter one in our previous work to accommodate non-determinism of a program caused by e.g., multi-threading. To demonstrate the usefulness of our dynamic reverse-code generation, this article presents a case study of various backtracking methods including ours. We compare the memory usage of various backtracking methods in a simple but nontrivial example, a bounded-buffer program. In the case of non-deterministic programs such as this bounded-buffer program, our dynamic reverse-code generation outperforms the existing backtracking methods in terms of memory efficiency.

1 Introduction

When something goes wrong, we people tend to think backwards to find the cause of an observed problem. As a simple example, if you lost something valuable, you would go back to the places you visited during the day, one by one starting from the last place you visited. When a program goes wrong, it is tempting to do the same for debugging. One would wish to trace backwards an erroneous execution path to find the cause of unexpected behavior of a program.

However, alas, programs run only forwards. As a result, traditional debugging methods are by and large speculation-based. Developers first guess program points that are potentially problematic, and observe program states at those program points by inserting print commands or using breakpoints. Such speculation-based debugging is often time-consuming, and requires high expertise in the program being debugged.

Indeed, empirical studies of Ko and Myers [11, 12] suggest that backward reasoning, which is supported through their custom debugger, makes a debugging process more efficient as compared to the traditional speculation-based methods; in their user experiments, users could complete more debugging tasks successfully in a shorter time when backward reasoning is facilitated than when a traditional breakpoint-based method is used.

In fact, the idea of supporting backtracking in debugging is not new. There have been numerous studies that make use of backtracking for debugging [1, 2, 3, 4, 5, 6, 7, 10, 11, 12]. However, all the backtracking methods used in those studies share a common problem; there is a limit in the length of an execution path that can be traced backwards. Program states beyond that limit cannot be restored

A. Banerjee, O. Danvy, K.-G. Doh, J. Hatcliff (Eds):
David A. Schmidt's 60th Birthday Festschrift
EPTCS 129, 2013, pp. 419–428, doi:10.4204/EPTCS.129.27

using those backtracking methods. This is because all backtracking methods rely on state saving or checkpointing (i.e., periodic state saving) where changes of program states are saved in the memory. Thus, as a program runs longer, more memory is consumed in general.

However, one backtracking method does not entirely depend on state saving. One can restore the previous program states by running special separate code. We call such special code *reverse code*. As a simple example, an assignment command `x:=x+1` can be traced backward by executing reverse code `x:=x-1`. Such reverse code can be generated through program analysis as evidenced by Akgul and Mooney [3] and in our previous work [14]. As one can imagine, it is impossible to obtain such reverse code for every command that is executed. Only in such cases, state saving is used. Such less dependence on state saving results in more applicability of backtracking.

Reverse code can be generated through static analysis [3] or dynamic analysis [14]. We refer to these two methods as static reverse-code generation and dynamic reverse-code generation, respectively. The former method prepares reverse code in advance before starting a debugging session, whereas the latter method generates reverse code, which works only for the current execution path, on the fly during a debugging session. While the difference between these two methods will be detailed in Section 2, we first point out that non-determinism caused by multi-threading has a different impact on each method. Non-determinism makes almost no impact on dynamic reverse-code generation; reverse code can be generated as long as the current execution path can be obtained. Meanwhile, it is challenging to accommodate non-determinism to static reverse-code generation, essentially due to the inherent limitations of static analysis. Indeed, the state-of-the-art method of Akgul and Mooney [3] does not support non-deterministic programs.

Overall, we believe that dynamic reverse-code generation can widen the applicability of reverse code to non-deterministic programs, while keeping memory-efficiency of reverse-code-based backtracking. To support our belief, we present a case study of various backtracking methods, including dynamic reverse-code generation (Section 3). In particular, we illustrate the usefulness of dynamic reverse-code generation by (1) considering the standard example of a multi-threaded program managing a bounded buffer and (2) measuring its memory usage for each existing backtracking method and for dynamic reverse-code generation. As will be shown, our dynamic reverse-code generation method consumes the least amount of memory among all the backtracking methods considered. Surely, memory efficiency, while important, is not the only factor that affects the usefulness of a backtracking method. We consider in Section 4 other factors that deserve consideration such as the response time of backtracking and memory overhead of storing reverse code.

2 Backtracking Methods

Four kinds of backtracking methods are found in the literature: state saving, checkpointing, static reverse-code generation and dynamic reverse-code generation. Notably, the method for *replay* [9, 13, 15], that guarantees to reproduce the same program execution as in the past, is excluded from the list. It is infeasible to backtrack with the replay method alone since a given program must be restarted from the beginning of its execution.

2.1 State saving

This method saves previous program points and previous data values while running a program forwards and restores them in a LIFO (Last In, First Out) manner when backtracking [2, 4, 10, 16]. The naive approach is to save a whole state-vector consisting of global variables, local variables and program point, whenever the current state of a program changes (*basic state saving*). Alternatively, one can save only modified values (*incremental state saving*). State saving is the easiest way to perform backtracking.

$z \in \mathbb{Z} \quad x \in \textbf{Variable} \quad E \in \textbf{Expression} \quad C \in \textbf{Command}$

$E ::= z \mid x \mid E+E \mid E-E \mid E \times E \mid E/E \mid E^b \qquad E^b ::= \mathsf{true} \mid \mathsf{false} \mid E{==}E \mid E{>}E \mid !E^b$

$C ::= x{:=}E \mid \mathsf{skip} \mid C;C \mid \mathsf{if}\ E^b\ \mathsf{then}\ \{C\}\ \mathsf{else}\ \{C\} \mid \mathsf{while}\ E^b\ \mathsf{do}\ \{C\}$

(a) Our minimal programming language

$$\mathtt{g} := \mathtt{d}+1 \longrightarrow \mathtt{e} := \mathtt{g}\times 2 \longrightarrow \mathtt{g} := \mathtt{e}-1$$

(b) An example execution path

$$\mathtt{g} := \mathtt{d}+1 \longrightarrow \mathtt{e} := \mathtt{g}\times 2 \longrightarrow \mathtt{g} := \mathtt{e}-1 \longrightarrow \mathtt{d} := \mathtt{g}\times 3$$

(c) An example execution path

Figure 1: Our programming language and example execution paths

However its clear drawback is that memory is consumed for every statement that changes the state of a program, e.g., an assignment. Therefore, memory consumed for backtracking is directly proportional to the length of the execution path.

2.2 Checkpointing

This method saves previous program points and previous data values only at predefined *checkpoints* (hence 'checkpointing'), not on every statement. When backtracking, one restores the state saved at the previous checkpoint (thus the program point goes back to the previous checkpoint), and runs the program forward to the desired previous program point [1, 7]. As in the case of state saving, it is more economical to only save changes between adjacent checkpoints than to save a state-vector at a checkpoint. Checkpointing can consume less memory than state saving. When the same variable is modified several times between adjacent checkpoints, only the variable value at the previous checkpoint is saved. However, checkpointing is in a broad sense still a state saving because states are saved periodically. Therefore, memory consumed for backtracking is still directly proportional to the length of the execution path.

2.3 Reverse-code generation

This method first generates reverse code. Executing this reverse code restores the previous states of a program, and as a result, a program backtracks. As a simple example, the reverse code of $x := x+1$ is $x := x-1$. Such simple reverse code for self-defined assignments (i.e., assignments where the same variable is used in both sides of the assignments) are used in various reverse execution methods [5, 8]. To handle more general cases beyond self-defined assignments, the combination of the following three techniques are used by Akgul and Mooney [3] and by the author [14]: the redefine technique, the extract-from-use technique, and lastly the aforementioned state-saving technique that is used only if neither of the previous two techniques can be applied.

2.3.1 Reverse-code generation techniques

Notations. To explain reverse-code generation techniques, we assume a minimal imperative programming language shown in Figure 1(a). We also use several notations. We refer to an execution path as π. An execution path consists of assignment commands, and we refer to the assignment command at n-th position as $\pi(n)$. Lastly, we refer to the left-hand-side variable of an assignment C as $lhs(C)$, and the set of variables appearing in right-hand side of C as $rhs(C)$.

Now, consider an n-length execution path π whose last assignment command (i.e., $\pi(n)$) is about to be reverse-executed. Then, each of the technique works as follows.

(1) The redefine technique. This technique looks for the reaching definition of $lhs(\pi(n))$ in π, and re-execute that reaching definition rd if no variable in $rhs(rd)$ is modified between rd and $\pi(n)$. As a result, a variable equivalent to the $lhs(\pi(n))$ reverts to its previous value. Reverse code, in this case, is the reaching definition rd. Meanwhile, if some variables in $rhs(rd)$ are modified between rd and $\pi(n)$, then those modified variables should be restored beforehand through the three restoration techniques of reverse-code generation. Reverse code, in this case, is a sequence of (i) reverse code for those modified variables and (ii) rd.

As an example, consider the execution path shown in Figure 1(b). Note that in the end of the path the value of variable `g` is modified. The reaching definition of the last command is `g:=d+1`, and variable `d` is not modified behind that reaching definition. Thus, reverse code to reverse-execute the last command is simply `g:=d+1`.

(2) The extract-from-use technique. Consider an execution path that ends with $x_2 := x_1 + 1$; $x_1 := 0$. To reverse-execute the last assignment and restore the previous value of x_1, one can execute reverse code $x_1 := x_2 - 1$ that is obtained from the previous assignment, $x_2 := x_1 + 1$.

More generally, this technique looks for an assignment command C where $rhs(C)$ contains $lhs(\pi(n))$. In addition, such C must be located behind the reaching definition of $lhs(\pi(n))$. If there is no such C, this technique cannot be used. Suppose $x_2 := f(x_1)$ is the C that is found, and function f is invertible. If x_2 is not modified between C and $\pi(n)$, then $lhs(\pi(n))$, which is x_1 in this case, can be restored by executing reverse code $x_1 := f^{-1}(x_2)$. Meanwhile, if x_2 is modified, the above reverse code should be preceded by additional reverse code to restore x_2.

The above function f does not have to involve only one variable. An expression like $x_1 + x_2$ can be viewed as $f(x_1)$ by replacing x_2 with its value at that program point. If this is the case, a use of f^{-1} in reverse code should be preceded by code that makes sure the value of x_2 is restored beforehand.

As an example, consider the execution path in Figure 1(c) that in the end modifies the value of variable `d`. Since `d` is used in the right-hand side of the first statement (i.e., `g:=d+1`) before `d` is re-defined at the end of the execution path, the first candidate of the reverse code for `d` is `d:=g-1`. However, `g` is redefined after the first statement by `g:=e-1`, so we need to recover `g` beforehand. In this case, from `e:=g`×`2`, we get `g:=e/2`. Putting them together, we can restore the previous value of `d` through `d:=e/2-1`.

Reverse code can be pre-generated before starting a debugging session. Or it can also be generated on the fly during a debugging session. We refer to the former as static reverse-code generation, and the latter as dynamic reverse-code generation. We describe the differences between them in the following two subsections.

(3) The state-saving technique. As mentioned, if neither of the previous two techniques can be applied, then state saving is used as the last resort. As an example, suppose that $lhs(\pi(n))$ is variable x_1, and thus the previous value of x_1 needs to be restored to backtrack. If x_1 is assigned its previous value through user input, the redefine technique cannot be used. Although the extract-from-use technique can be used instead in some cases, this is not always the case. For example, an assignment command $x_2 := x_1 \times 0$ that is executed between the reaching definition of x_1 and $lhs(\pi(n))$ does not help with applying the extract-from-use technique because the function comprised of the right-hand side of the assignment is not invertible.

2.3.2 Static reverse-code generation

Akgul and Mooney employed a path-sensitive static analysis to pre-generate reverse code before starting a debugging session [3]. To limit the number of paths that should be considered, they unroll each loop a few times during the analysis. Despite that, it is occasionally possible to generate a reverse loop (i.e., a loop whose execution reverse-executes the original loop of a source program) consisting of non-state-saving code. This is particularly true when iterations of a loop form a regular pattern such as 1, 3, 5, ... for an integer variable.

To distinguish different paths in reverse code, conditionals are used. As an example, suppose that the reaching definition of x_1 is rd_1 if a boolean expression E^b holds, and rd_2 if $!E^b$ holds. Then, possible reverse code is `if` E^b `then` $\{C_1; rd_1\}$ `else` $\{C_2; rd_2\}$, assuming that (i) the variables used in E^b are properly restored beforehand through a preceding reverse code fragment, and (ii) C_1 and C_2 restore the values of the variable in $rhs(rd_1)$ and $rhs(rd_2)$, respectively. The boolean expressions used in conditionals of reverse code are also obtained through static analysis of the source program.

While by using reverse code one can avoid saving every value change, there is a memory overhead associated with storing reverse code. In the case of static reverse-code generation, the size of reverse code is pre-defined before starting a debugging session. In general, this memory overhead for reverse code is offset as program execution goes on because less memory is used for state saving.

However, non-determinism caused by multi-threading changes the dynamics. There are too many execution paths to consider due to interleaving between threads. It is difficult to pre-generate reverse code of reasonable size that works for those many execution paths. In addition, it is difficult to make static reverse code to infer dynamic non-deterministic choices (i.e., context switches between threads), if not impossible. Indeed, Akgul and Mooney [3] considered only deterministic programs.

2.3.3 Dynamic reverse-code generation

We introduced in the author's previous work [14] a method that generates reverse code on the fly during a debugging session. This method applies the three techniques of reverse-code generation directly to the current execution path. Since an execution path to consider is fixed, it is tractable to generate reverse code through the three reverse-code generation techniques we earlier explained.

It is our thesis that this dynamic method is useful for backtracking in non-deterministic programs. In the next section, we substantiate this thesis by comparing the memory consumed by the backtracking methods described earlier. In the end, we will show that our dynamic reverse-code generation consumes the least amount of memory.

3 The Case of a Bounded Buffer

In this section, we introduce a bounded-buffer program for which we measure the memory usage for each of the backtracking methods presented in the previous section. Figure 2 shows a Java-like program manipulating a bounded buffer shared between multiple threads. This bounded-buffer program consists of producers, consumers and a finite buffer shared between producers and consumers. Producers put data into the buffer unless the buffer is full, and the data in the buffer is fed to consumers unless the buffer is empty. For the sake of brevity, we consider the simple case where only one pair of producer and consumer exists. We assume that the size of the buffer is `M`. Elements in the source array (denoted by `src[N]`) of the producer are copied to the destination array (denoted by `dst[N]`) of the consumer in order after incrementing the value by one. For example, if `src[N]` contains $\{10, 20, 30\}$ in the beginning, then in the end, `dst[N]` is filled with $\{11, 21, 31\}$. Semaphores are used to control concurrency. Two semaphore procedures `wait` and `signal` (assuming call-by-reference) are shown in Lines 41-48 of Figure 2. In the definition of `wait`, command `await(s>0)` blocks the execution if condition `s>0` does not hold. Therefore,

```
/***************************
  * Bounded-Buffer Example *
  **************************/
int buf[M]; // buffer size M
int g:=0; // experimental code
int empty:=M;
int full:=0;

thread Producer {
  int src[N]; // source array
  int p:=0;
  int rear:=0;
  int d:=0; // experimental code
  while (p < N) {
    wait(empty);
    buf[rear]:=src[p];
    p:=p+1;
    rear:=rear+1;
    rear:=rear % N;
    signal(full);
    g:=d+1; // experimental code
    d:=g×3; // experimental code
  }
}
thread Consumer {
  int dst[N]; // destination array
  int c:=0;
  int front:=0;
  int e:=0;
  while (c < N) {
    wait(full);
    dst[c]:=buf[front]+1;
    c:=c+1;
    front:=front+1;
    front:=front % N;
    signal(empty);
    e:=g×2; // experimental code
    g:=e-1; // experimental code
  }
}
// Two semaphore procedures
procedure wait(int s) {
  // atomic action
  ⟨await(s>0); s:=s-1;⟩
}
procedure signal(int s) {
  ⟨s:=s+1;⟩ // atomic action
}
```

Figure 2: The bounded-buffer program.

if `s` becomes zero indicating that the buffer is full, then the producer thread waits until the contents of the buffer are consumed by the consumer thread, and `s` becomes positive as a result. Once `await(s>0)` is passed successfully, the next command, `s:=s-1`, is immediately executed without being interfered by another thread as usual in semaphore.

To ensure the linearity of execution paths, we simply assume that threads are interleaved during a debugging session. Note that this does not mean that a debugger that supports backtracking should not allow parallel (i.e. non-interleaved) execution of threads. The linearity of execution paths can still be ensured by employing a more sophisticated method, e.g., logging the orders of access to unprotected regions of code (i.e. regions that can cause a data race).

When interleaving threads, we assume the assignment-command-level granularity, not the usual instruction-level granularity. In other words, interleaving cannot take place while an assignment command, e.g., `p:=p+1`, is being executed. (Meanwhile, when the instruction-level granularity is used, an instruction for a read-access to `p` and an instruction for a write access to `p` can be interspersed with other instructions executed by other threads.) Such simplification is merely to avoid excessive complications in this case study. All the backtracking methods explained earlier can be used at both the command level and the instruction level. For example, Akgul and Mooney perform their analysis for static reverse-code generation at the instruction level [3].

In order to run a program backwards, it is necessary to restore the previous program points and data values. First, to restore the previous program points in a non-deterministic program, it is necessary to remember what choice was made (e.g., which thread became active) and when that choice was made (e.g., when the thread context was switched). Such information can be inferred by looking at the following logs: the order of (1) the accesses to the entries of basic blocks, and (2) program points where the thread context is switched. We assume that those logs are recorded in every backtracking method.

Meanwhile, to restore the data values, each backtracking method presented in the previous section induces a different memory-usage pattern. In the following, we show how many integer memory units (we assume that one integer value costs I bytes) are consumed in each of the backtracking methods to support backtracking in our running example.

3.1 Basic state saving

This method saves a state vector whenever the state of a program changes. A state vector of the program consists of global variables and local variables. In the running example, we have nine integer variables (`p`, `c`, `front`, `real`, `empty`, `full`, `g`, `d`, `e`), one M-length integer array variable (`buf`) and two N-length integer array variables (`src,dst`). The state of a program changes eight (8) times in the loop body of `Producer` and `Consumer` respectively. Until the program terminates, `Producer` and `Consumer` respectively execute their loop body N times. So overall, basic state saving costs $8(9+M+2N)I \times N \times 2$ units of memory.

3.2 Incremental state saving

This method saves only modified values instead of state-vectors. As mentioned above, the state of a program changes eight times in the loop body of `Producer` and `Consumer` respectively. All the modified values (including `buf[rear]` and `dst[c]`) take integer space. The loop body of each thread iterates N times, as pointed out previously. Therefore incremental state saving costs $8I \times N \times 2$ units of memory, which is certainly less than the memory usage in basic state saving.

3.3 Checkpointing

This method saves states periodically at predefined checkpoints. The memory requirement typically depends on how densely checkpoints are set. The more coarsely they are set, the less memory is likely to be consumed but the more runtime is required to reach the previous program point from the nearest checkpoint. In our example, let us suppose that two checkpoints are set at Lines 15 and 31.

Since state vectors are expensive, the incremental-state-saving idea is also often used in checkpointing; only data values modified during the previous period are saved. For example, if Lines 15-20 are executed before reaching the checkpoint at Line 31, `buf[rear],p,rear` and `full` are saved.

Note that `rear` is saved only once. In incremental state saving, `rear` is saved twice (one after Line 18, the other after Line 19). Incremental checkpointing consumes less memory than incremental state saving when the same variable is modified several times between adjacent checkpoints. In our example `rear` and `front` are modified twice, hence we can save $2I$ units of memory. If Lines 21 and 38 are executed consecutively so that `g` is modified twice in a row, we can also economize one more unit of memory. In this optimal case, checkpointing costs $(16-3)I \times N$ units of memory, which is less than the memory usage in incremental state saving.

3.4 Static reverse-code generation

This method performs backtracking by running pre-generated reverse code. We earlier pointed out in Section 2 that the current static reverse-code generation does not help much in supporting backtracking of multi-threaded programs due to its inherent non-determinism. Only self-defined assignments can still be used to generate non-state-saving reverse code. Thus, in our running example, the commands that can be restored through non-state-saving reverse code are `p:=p+1` at line 17, `rear:=rear+1` at line 18, `c:=c+1` at line 33, `front:=front+1` at line 34, and `wait`/`signal` commands. The remaining eight commands (four in `Producer` and another four in `Consumer`) are state-saved. Since the loop body of each thread iterates N times, overall, the static reverse-code generation costs $4I \times N \times 2$ units of memory, which is less than the memory usage in checkpointing.

3.5 Dynamic reverse-code generation

This method performs backtracking by running reverse code generated on the fly. We earlier mentioned that reverse code can be generated from the current execution path. Recall that to restore the previous program points, we log the order of accessed basic-block entries and context-switch points. It is possible to obtain the current execution path consisting of executed commands from that logged information and the program source code.

Different reverse code is dynamically generated depending on the current execution path. For example, the following shows four possible scenarios among many that modifies variable `d` at line 22.

(a) $21 : \mathsf{g} := \mathsf{d}+1 \longrightarrow 22 : \mathsf{d} := \mathsf{g} \times 3$

(b) $21 : \mathsf{g} := \mathsf{d}+1 \longrightarrow 37 : \mathsf{e} := \mathsf{g} \times 2 \longrightarrow 38 : \mathsf{g} := \mathsf{e} - 1 \longrightarrow 22 : \mathsf{d} := \mathsf{g} \times 3$

(c) $21 : \mathsf{g} := \mathsf{d}+1 \longrightarrow 37 : \mathsf{e} := \mathsf{g} \times 2 \longrightarrow 22 : \mathsf{d} := \mathsf{g} \times 3$

(d) $21 : \mathsf{g} := \mathsf{d}+1 \longrightarrow 38 : \mathsf{g} := \mathsf{e} - 1 \longrightarrow 22 : \mathsf{d} := \mathsf{g} \times 3$

In case of scenario (b), the thread context switches after line 21 and after line 38. In all the four scenarios, line 22 can be backtracked without state saving. In the first scenario, dynamic reverse code, `d:=g-1`, can restore the previous value of `d`. The second scenario was already considered in Figure 1(c). The remaining two scenarios can also be handled similarly to generate dynamic reverse code.

We can also recover array elements similarly by dynamic reverse code. Consider the following scenario of an execution path (array indices are replaced with their runtime values) that in the end modifies `buf[0]`.

$$32 : \mathtt{dst}[0] := \mathtt{buf}[0]+1 \longrightarrow 33 : \mathsf{c} := \mathsf{c}+1 \longrightarrow 34 : \mathtt{front} := \mathtt{front}+1 \longrightarrow 35 : \mathtt{front} := \mathtt{front}\%\mathtt{N}$$
$$36 : \mathtt{signal(empty)} \longrightarrow 15 : \mathtt{wait(empty)} \longrightarrow 16 : \mathtt{buf}[0] := \mathtt{src}[2]$$

Possible reverse code in this case is `buf[0]:=dst[0]-1`. The previous value of `buf[0]` can be restored by executing this reverse code. Note that the original command at line 32 is `dst[c]:=buf[front]+1`. The fact that `c` and `front` had value 0 can be obtained by separate reverse code for `c` and `front`.

Overall, for the above execution scenarios, we only need to save the value of `rear` at line 19 and the value of `front` at line 35. Since the loop body of each thread iterates N times, overall, dynamic reverse-code generation costs $I \times N \times 2$ units of memory, which is less than the memory usage in static reverse-code generation. It is also noteworthy that the gap between the dynamic reverse-code generation method and the other ones becomes wider as the number of threads (i.e., `Producer` and `Consumer`) increases.

4 Discussion

The comparison of Section 3 is focused on the memory usage of each backtracking method. Meanwhile, the response time of backtracking is also important. It is apparent that the dynamic reverse-code generation method requires more CPU cycles during a debugging session than the other methods. Thus, it is necessary to put engineering efforts to lower response time. One possible way is to use both state saving and dynamic reverse-code generation altogether in a complementing way. That is, while running a program, state saving can be used initially to guarantee a quick response time. At the same time, a background process can replace stored data with reverse code one by one. Then when backtracking, reverse code is executed if it is ready; otherwise, stored data are used instead.

Another point worth considering is a memory overhead associated with storing reverse code. The case for static reverse-code generation was earlier considered. The size of static reverse code is predefined before starting a debugging session. Meanwhile, the situation in dynamic reverse-code generation is more sophisticated. In one extreme, memory overhead can be entirely avoided if reverse code is generated only when necessary. As pointed out earlier, however, this will increase the response time of backtracking. In the opposite extreme, reverse code can be continuously generated and kept in the memory as the program runs. While this will decrease the response time, the memory overhead in this case is proportional to the length of the current execution path. We believe that it is wise to make a trade-off between these two extremes. That is, it is possible to keep in the memory only the reverse code for the last n commands of the current execution path where n is a certain trade-off number. As an execution path gets longer, the oldest fractions of reverse code in the memory are deleted from the memory. Those deleted fractions of reverse code can be re-generated in the background while the latest fractions of reverse code are executed to backtrack.

5 Conclusion

We have shown how much memory is consumed in various backtracking methods when a bounded buffer is accessed concurrently. In particular, we have pointed out why the existing static reverse-code generation does not work well on non-deterministic programs such as multi-threaded ones and why the dynamic reverse-code generation does. Finally, we have illustrated that our dynamic reverse-code generation can use less memory than the existing backtracking methods when applied to non-deterministic programs.

Acknowledgments: I am grateful to Dave Schmidt for his kindness. I admire him for his devotion to research, education, and humanity. I am also grateful to Olivier Danvy for encouraging me to write this paper. Finally, my thanks also go to anonymous reviewers.

References

[1] Hiralal Agrawal, Richard A. DeMillo & Eugene H. Spafford (1991): *An Execution-Backtracking Approach to Debugging*. *IEEE Software* 8(3), pp. 21–26, doi:10.1109/52.88940.

[2] Hiralal Agrawal, Richard A. DeMillo & Eugene H. Spafford (1993): *Debugging with Dynamic Slicing and Backtracking*. *Software - Practice and Experience* 23(6), pp. 589–616, doi:10.1002/spe.4380230603.

[3] Tankut Akgul & Vincent J. Mooney III (2004): *Assembly Instruction Level Reverse Execution for Debugging*. *ACM Transactions on Software Engineering and Methodology* 13(2), pp. 149–198, doi:10.1145/1018210.1018211.

[4] Robert M. Balzer (1969): *EXDAMS–EXtendable Debugging and Monitoring System*. In: *Proceedings of AFIPS Spring Joint Computer Conference*, 34, AFIPS Press, pp. 567–580, doi:10.1145/1476793.1476881.

[5] Bitan Biswas & R. Mall (1999): *Reverse Execution of Programs*. *SIGPLAN Notices* 34(4), pp. 61–69, doi:10.1145/312009.312079.

[6] Simon P. Booth & Simon B. Jones (1997): *Walk Backwards to Happiness – Debugging by Time Travel*. In: *Proceedings of the Workshop on Automated and Algorithmic Debugging*, pp. 171–183.

[7] Bob Boothe (2000): *Efficient algorithms for bidirectional debugging*. In: *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 299–310, doi:10.1145/349299.349339.

[8] Christopher D. Carothers, Kalyan S. Perumalla & Richard M. Fujimoto (1999): *Efficient Optimistic Parallel Simulations Using Reverse Computation*. In: *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*, pp. 126–135, doi:10.1145/347823.347828.

[9] Jong-Deok Choi & Harini Srinivasan (1998): *Deterministic replay of Java multithreaded applications*. In: *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, ACM Press, pp. 48–59, doi:10.1145/281035.281041.

[10] Jonathan J. Cook (2002): *Reverse Execution of Java Bytecode*. *The Computer Journal* 45(6), pp. 608–619, doi:10.1093/comjnl/45.6.608.

[11] Andrew Ko & Brad A. Myers (2009): *Finding causes of program output with the Java Whyline*. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'09)*, pp. 1569–1578, doi:10.1145/1518701.1518942.

[12] Andrew J. Ko & Brad A. Myers (2008): *Debugging reinvented: asking and answering why and why not questions about program behavior*. In: *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pp. 301–310, doi:10.1145/1368088.1368130.

[13] Thomas J. Leblanc & John M. Mellor-Crummey (1987): *Debugging parallel programs with instant replay*. *IEEE Transactions on Computers* 36(4), pp. 471–482, doi:10.1109/TC.1987.1676929.

[14] Jooyong Lee (2007): *Dynamic Reverse Code Generation for Backward Execution*. *Electronic Notes in Theoretical Computer Science* 174(4), pp. 37–54, doi:10.1016/j.entcs.2006.12.028.

[15] Mark Russinovich & Bryce Cogswell (1996): *Replay for concurrent non-deterministic shared-memory applications*. In: *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'96)*, ACM Press, pp. 258–266, doi:10.1145/231379.231432.

[16] Marvin V. Zelkowitz (1971): *Reversible Execution as a Diagnostic Tool*. Ph.D. thesis, Department of Computer Science, Cornell University.