An Efficient and Generic Reversible Debugger using the Virtual Machine based Approach

Toshihiko Koju
Graduate School of Science and Technology, Keio University
k-ju@doi.ics.keio.ac.jp

Shingo Takada
Graduate School of Science and Technology, Keio University
michigan@ics.keio.ac.jp

Norihisa Doi
Faculty of Science and Engineering, Chuo University
doi@ise.chuo-u.ac.jp

ABSTRACT

The reverse execution of programs is a function where programs are executed backward in time. A reversible debugger is a debugger that provides such a functionality. In this paper, we propose a novel reversible debugger that enables reverse execution of programs written in the C language. Our approach takes the *virtual machine based approach*. In this approach, the target program is executed on a special virtual machine. Our contribution in this paper is two-fold. First, we propose an approach that can address problems of (1) compatibility and (2) efficiency that exist in previous works. By compatibility, we mean that previous debuggers are not generic, i.e., they support only a special language or special intermediate code. Second, our approach provides two execution modes: the native mode, where the debuggee is directly executed on a real CPU, and the virtual machine mode, where the debuggee is executed on a virtual machine. Currently, our debugger provides four types of trade-off settings (designated by unit and optimization) to consider trade-offs between granularity, accuracy, overhead and memory requirement. The user can choose the appropriate setting flexibly during debugging without finishing and restarting the debuggee.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*

General Terms

Reliability

Keywords

Debugger, Reverse Execution, Virtual Machine

VEE'05, June 11-12, 2005, Chicago, Illinois, USA.

1. INTRODUCTION

The reverse execution of programs is a function where programs are executed backward in time. In other words, it is a function that "undoes" effects of previously executed instructions, lines, and procedures. A reversible debugger is a debugger that provides such a functionality.

Reverse execution is very useful for locating the causes of software failures. In general, software does not fail immediately after the cause of the failure is executed. Instead, the software may fail some time after the cause of the failure has been executed. In such cases, a programmer may want to backtrack from the location where the failure occurred to the location where its cause exists. But most debuggers cannot perform reverse execution. If the programmer wants to investigate the state of the program at the location immediately before the failure occurs, he/she will need to restart the program and set a breakpoint at that location to stop it. If the programmer wants to go further back in the program, he/she will again need to restart the program. The programmer will need to repeat this tedious process until the cause of the failure is successfully located.

In this paper, we propose a novel reversible debugger that enables reverse execution of programs written in the C language. Our approach takes the *virtual machine based approach* [9, 12]. In this approach, the target program is executed on a special virtual machine, instead of directly on a real CPU. Our contribution in this paper is two-fold. First, we propose an approach that can address problems of (1) compatibility and (2) efficiency that exist in previous works using the virtual machine based approach. Second, our approach provides two execution modes: the native mode, where the debuggee is directly executed on a real CPU, and the virtual machine mode, where the debuggee is executed on a virtual machine [1].

In the rest of this paper, we first describe related work. Section 3 presents the design of our reversible debugger. Then, section 4 describes the implementation of our reversible debugger. Section 5 gives an evaluation, and finally section 6 makes concluding remarks.

2. RELATED WORK

There have been many attempts to realize reverse execution. The most common approach is to create a history log of changes to program states during execution [1, 3, 4, 6, 8,

[1]Note that our debugger deals with only applications and not entire systems (that include hardware and OS).

9, 11, 12]. For example, the contents of memory are saved before the program tries to overwrite it. Reverse execution can then be realized by reconstructing the original memory contents using the saved information.

PROVIDE [12] is a visualization and debugging environment for a subset of the C Language. PROVIDE executes a program on its interpreter. The interpreter records state transitions to a process history database. Spyder [1] is a debugging tool for the C Language. Spyder uses a structured-backtracking approach which identifies and saves a change-set of a statement. A change-set consists of all variables whose values are modified by a statement. ZStep94 [11] is a source code stepper for Lisp. ZStep94 uses a special interpreter, and keeps a history of evaluations. In addition, it maintains a consistent view of the user interface, and can backtrack graphical outputs. Chen et al. [6] proposed an approach based on instrumentation of assembly code. Their approach keeps all states that are changed by assembly instructions. Akgul et al. [3] also proposed an approach that enables assembly instruction level reverse execution. It distinguishes states that are reconstructable from other states. An inverse version of the program is generated to avoid saving reconstructable states. So, it can reduce storage requirements. This approach was improved by using dynamic slicing [4]. Cook [8] proposed a reversible debugger for Java. It exploits Kaffe's interpreter so that it can capture state changes caused by Java bytecode. It also supports multi-threading. Leonardo Virtual Machine (LVM) [9] is a portable virtual machine which has the function of reverse execution. It uses a transaction for a unit of reverse execution. The length of each transaction can be freely changed during execution. Executables for LVM can be produced by using the Leonardo C Compiler.

There are two main issues for approaches which create history logs: what kinds of execution states need to be saved, and how to save the execution states.

The former is a very difficult issue because there are trade-offs between forward and reverse execution speed, memory requirement, granularity of a reverse execution unit, analysis cost, etc. For example, [9] showed that larger granularity can realize faster execution and smaller memory requirement. Another example would be the difference in the states that are saved in a loop in [1] and [6]. In [1], only variables which could be modified during execution of an entire loop are saved before entering the loop. On the other hand, in [6], every state changed by each instruction in the loop body are saved. So, [1] can achieve faster execution and smaller memory requirement. But it cannot backtrack each loop iteration. [6] is the exact opposite. It can backtrack each loop iteration or each instruction in the loop body, but it has higher overhead and needs larger memory. [3] showed that faster execution and smaller memory requirement can be achieved in exchange for possibly large analysis cost. Futhermore, [4] showed that faster reverse execution along dynamic slices can be achieved with extra analysis cost. From the above consideration, it is very important not to limit users to a particular trade-off setting. A reversible debugger should provide multiple types of trade-off settings to consider related trade-offs and be able to switch among them flexibly during debugging. But most of the above works do not have such a capability.

There are two software based approaches focusing on the latter issue, i.e. how to save execution states. One is the *static instrumentation approach* [6] and the other is the *virtual machine based approach* [8, 9, 11, 12]. In the static instrumentation approach, code is inserted to save states that are changed during execution. This insertion (or translation of the program) is done before the target program is started. In the virtual machine based approach, the target program is executed on a special virtual machine, instead of directly on a real CPU. The virtual machine is responsible for saving states that are changed during execution. As mentioned above, a reversible debugger should provide multiple types of trade-off settings and be able to switch among them. So, the virtual machine based approach is preferable rather than the static instrumentation approach. This is because the virtual machine based approach is much more flexible compared to the static instrumentation approach. In the static instrumentation approach, all translation occurs before the program starts. So, users need to consider trade-offs before starting the program. If the user wants to make changes during execution, the user must finish the program, re-translate it, and restart it. This is a very tedious process. In contrast, the virtual machine based approach does not have such limitations. The behavior of a virtual machine (including settings for trade-offs) can be changed freely during execution. So, the virtual machine based approach can provide the above capability much more easily.

When we focus on reversible debuggers for the C language, PROVIDE [12] and LVM [9] are examples that take the virtual machine based approach. But they have two serious problems: (1) compatibility and (2) efficiency. Both works have a compatibility problem with existing development environments. PROVIDE [12] supports only a subset of the C language, PROVIDE-C. There are many limitations, e.g., only integers, characters and one-dimensional arrays are supported. On the other hand, LVM provides the Leonardo C Compiler, which supports the ISO C89 standards. But it is not sufficient because LVM uses special intermediate code, and libraries of existing development environments (native machine code) cannot be used without recompiling the libraries using the Leonardo C Compiler. In other words, the source code of the libraries are necessary. So, to use PROVIDE or LVM, developers may first need to port their programs and/or libraries to the environments. This greatly reduces the usefulness of these environments.

The other problem, efficiency, is caused by the interpretive execution in both PROVIDE and LVM. Generally, C programs are compiled to native machine code, and directly executed on a real CPU. So, the slowdown factors in these environments are very significant.

3. REVERSIBLE DEBUGGER USING THE VM BASED APPROACH

As described in section 2, existing works which use the virtual machine based approach have two serious problems: (1) compatibility and (2) efficiency. So, we propose a new reversible debugger using the virtual machine based approach, which overcomes these problems:

(1) Compatibility Existing works have a compatibility problem because they support only a special language or special intermediate code. In our reversible debugger, we avoid using such a language or intermediate code. We choose to use native machine code as the target of our reversible debugger, and execute native machine

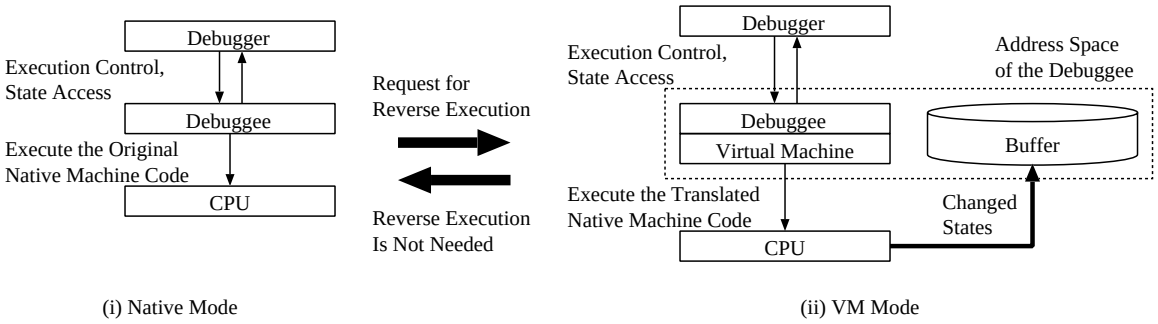


Figure 1: Overview of our reversible debugger.

code on our virtual machine. This makes it possible to use compilers, linkers, libraries, etc. of existing development environments along with our reversible debugger. There is no need to modify user programs and the user's development environment. Thus, we assure complete compatibility with existing development environments.

(2) Efficiency Existing works have an efficiency problem because they exploit the interpretive execution approach. In our reversible debugger, the virtual machine makes use of an approach called *dynamic translation* [5, 7, 10, 13]. Dynamic translation is an approach that performs code translation within a virtual machine. Our virtual machine translates native machine code of the target program by inserting code for saving states that are changed during execution. Code generated by this translation is also native machine code. Then, our virtual machine executes the generated code on the real CPU. This makes it possible to both execute the target program and save changed states directly by native machine code. Thus, we realize very fast execution compared with the interpretive execution approach.

In the following sections, we give an overview and then describe functionalities of our reversible debugger.

3.1 Overview

Fig. 1 shows an overview of our reversible debugger. We provide two main components: the reversible debugger itself and the virtual machine. The reversible debugger is responsible for providing its functionalities to users through its user interface. The virtual machine is responsible for saving states that are changed during execution of the target program.

As mentioned above, we use native machine code as the target of our reversible debugger to assure compatibility. In addition, our reversible debugger can use debugging information in the target program's executable file. Debugging information includes mapping between native machine code and source code. This information can easily be generated with existing compilers by turning on the debugging option. Thus, we can say that our reversible debugger offers its functionalities at both the native machine code level and the source code level.

One novel characteristic of our reversible debugger is its two execution modes. In our debugger, the target program is executed directly on the real CPU when reverse execution is not needed (Fig. 1 (i)). So, the execution does not incur any overhead. We call this the *native mode*. The target program is executed on our virtual machine only when reverse execution is needed (Fig. 1 (ii)). We call this the *VM mode*. As mentioned above, our virtual machine employs *dynamic translation* to realize very fast execution.

Providing these two modes offers two significant advantages. First, even though our virtual machine employs dynamic translation, execution on our virtual machine always incurs some degree of overhead, even if no state saving is performed[2]. In such a case, this overhead is completely unnecessary. By offering the native mode, we can avoid such overhead and execute the target program at full speed when reverse execution is not needed. Second, the capability of switching between the native mode and the VM mode enables *attaching* and *detaching* of already running programs. This is because our debugger does not require the target program to be entirely executed on our virtual machine. Our debugger can attach to an already running program on a real CPU, enter native mode and then switch to VM mode, or vice versa. "Attach" and "Detach" are basic functionalities of conventional debuggers, and essential for debugging of multi-processing programs.

Native Mode In this mode, users cannot reverse execute the target program, and functionalities are limited to those provided by traditional source code level debuggers, such as those given in Table 1 (b) and (c).

VM Mode In our reversible debugger, details of execution on our virtual machine are hidden from the users. So, the users can perform debugging activities without any attention to details of the virtual machine.

In this mode, users can use functionalities specific to reverse execution. Our virtual machine translates the native machine code of the debuggee (the target program of the debugger) by inserting code that will save states that are changed during execution. So, as the execution of the debuggee on our virtual machine proceeds, the changed states are saved in a buffer in the debuggee's address space (Fig. 1 (ii)). Later, the debugger performs reverse execution by reconstructing the original states using the information stored in the buffer.

[2]As will be described in section 5.1, the average was a slowdown of 1.7 times.

Table 1: Main functionalities of our reversible debugger.

(a)	Start	Start new program under debugger's control.
	Attach	Attach already running program to obtain its control.
	Kill	Terminate program.
	Detach	Detach program from debugger's control.
(b)	Breakpoint	Set and delete breakpoints.
	Continue	Continue execution.
	Step	Continue execution but for only one execution unit.
(c)	Access to registers	Read and write register values.
	Access to memory	Read and write contents of memory.
(d)	Enable reverse execution	Switch to VM mode.
	Disable reverse execution	Switch to native mode.
	Change trade-off setting	Choose appropriate trade-off setting
	Reverse execution	Perform reverse execution.

3.2 Functionality

In this section, we briefly describe functionalities of our reversible debugger from the user's viewpoint. Table 1 shows the main functionalities of our reversible debugger. Functionalities (a), (b), and (c) in Table 1 are very similar to conventional debuggers and can be used in the same way.

Functionalities (d) indicate those related to reverse execution. With "Enable reverse execution", the debuggee will be executed on our virtual machine. With "Disable reverse execution", the debuggee will be executed on the real CPU. With "Change trade-off setting", users can choose the appropriate trade-off setting to consider trade-offs between granularity, accuracy, overhead and memory requirement. Details are described later in section 4.1.1.1. Finally, with "Reverse execution", the user can reverse execute the debuggee.

From the user's viewpoint, "Enable reverse execution", "Disable reverse execution", and "Change trade-off setting" cause no side-effects to the execution of the debuggee, and functionalities (b) and (c) can be used in the same way regardless of the mode. So, even if a debuggee is executed on our virtual machine, it can be seen as executing directly on the real CPU by the user.

4. IMPLEMENTATION

In this section, we describe the implementation of our reversible debugger. We implemented our debugger for the Intel x86 architecture and Linux operating system. The debuggees need to be compiled with debugging information, e.g. with the -g option in GCC [3]. In the following sections, we describe the implementation of our virtual machine and then our reversible debugger.

4.1 Virtual Machine

Our virtual machine makes use of dynamic translation. Fig. 2 shows an overview of the execution of the debuggee along with dynamic translation on our virtual mchine:

(i) Save the register values and change the stack area to our virtual machine's own stack area.

(ii) Fetch a code fragment of the debuggee that should be executed next, and insert code for saving states that are changed during execution of that fragment.

[3]This is required to identify units of reverse execution (lines and procedures in source code).

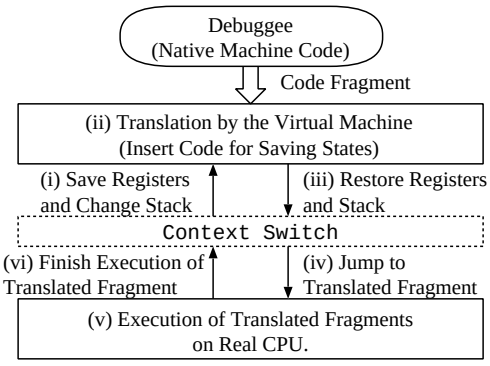


Figure 2: Execution on our virtual machine.

(iii) Restore the register values and the stack area backed up in (i).

(iv) Jump to the top of the translated fragment.

(v) Execute the translated fragment directly on the real CPU.

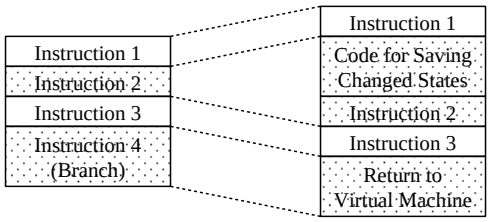
(vi) After the translated fragment finishes executing, return to step (i).

Note that we use the term *"context switch"* to indicate the switching between the target program and our virtual machine that happen in the same debuggee process. In the following sections, we describe code translation carried out in step (ii) in more detail.

4.1.1 Code Translation

Fig. 3 shows an overview of code translation in our virtual machine. A unit of translation in our virtual machine is a *code fragment*. A code fragment begins with an instruction that resides in the next execution point, and ends with the first branch instruction. Basically, our virtual machine performs two types of translation:

Insert Code for Saving Changed States Our virtual machine inserts code for saving states before the states are changed during execution of that fragment. For example, in Fig. 3, such a code is inserted between *instruction 1* and *instruction 2*, assuming that some state(s) are changed in *instruction 2* but not in *instructions 1*,



(a) Original Fragment (b) Translated Fragment

Figure 3: Overview of code translation.

```
; Increment memory content
; pointed by the register eax.
(1) inc dword [eax]

; Jump to ''somewhere''.
jmp somewhere
```

```
(2) Backup the registers.
Save the data type.
Save the address.
Save the contents.
(3) Restore the registers.
; The original instruction.
(1) inc dword [eax]

Save address of somewhere.
; Jump to the virtual machine.
jmp VM
```

(a) Original Fragment. (b) Translated Fragment.

Figure 4: Example of code translation.

3 and *4*. Currently, we provide four types of code for saving changed states based on the trade-off settings.

Change Destinations of Branches Our virtual machine changes destinations of branch instructions to let translated fragments return control to our virtual machine after finishing executing them on the real CPU. This corresponds to step (vi) in section 4.1. *Instruction 4* in Fig. 3 is an example.

Fig. 4 shows an example of the code translation. In Fig. 4, *italic* typeface indicates pseudo-code. In the original code (Fig. 4 (a)), the first instruction increments the memory content pointed by the register eax, and thus overwrites the memory content. The second instruction is a branch instruction that jumps to "somewhere".

In the translated code (Fig. 4 (b)), code for saving the changed state is inserted before the first instruction of the original code. In this case, the data type (e.g., the size of the content), the address (the value of eax), and the content (the value of dword [eax]) is saved. Also, translation is done to change the destination in the second instruction of the original code. First, the original destination address ("somewhere") is saved to the appropriate data area in our virtual machine. Then, the destination address is changed so that control will jump to our virtual machine.

Note that we do not destroy registers and memory locations in inserted code (excluding dead registers; see section 4.1.1.1). Thus, instructions in translated fragments corresponding to the original instructions can use the exact same registers and memory locations as the original instructions (Fig. 4 (1)). This means live registers and memory locations are kept "in the clear" for the original execution. This makes the switchings between the native mode and the VM mode very simple. In fact, the switchings can be done by just adjusting the current execution position as will be described in section 4.2.4.

In addition, our virtual machine performs *caching* and *linking* of translated fragments [13]. With caching, once original fragments of the debuggee are translated, the translated fragments are registered to our virtual machine's cache buffer. This makes it unnecessary to re-translate the same fragments. With linking, translated fragments in the cache buffer are directly linked with branch instructions. So, context switches in Fig. 2 can be avoided between linked fragments. These approaches make the execution speed of our virtual machine extremely fast. But for indirect branch instructions, we cannot apply linking because their destinations are not fixed. Instead, we execute small assembly code that checks a sub-cache of translated fragments each time they are executed. In the following sections, we describe the code inserted for saving changed states and the circular buffer used to save the changed states.

4.1.1.1 Code for Saving Changed States.

The execution states of programs consist of internal states and external states. Internal states are the contents of registers and memory. External states are any other states that are mainly managed by the operating system, such as file systems. To realize reverse execution, we must save the differences between the execution states when they change. For internal states, we need to save the contents of registers and memory before the change. For external states, we need to save enough information to reconstruct the original external states, such as the contents and/or the I/O pointer of a file before the change.

Currently, our reversible debugger provides four types of trade-off settings to consider trade-offs between granularity, accuracy, overhead and memory requirement. The user can choose the appropriate setting by designating *unit* and *optimization*.

In our reversible debugger, the user can designate the *unit* of reverse execution to be line or procedure of C source code. We avoid saving the same registers many times during execution of instructions corresponding to a single line or procedure. The code for saving registers are gathered together for such instructions, and we only save registers once per execution of a line or a procedure. For any other states, such as contents of memory and external states, we insert code for saving states immediately before they change. So, they may be saved multiple times in one construct.

The unit of reverse execution offers a trade-off between granularity, (forward) execution speed, and memory requirement. A line is of smaller granularity than a procedure. But using a line as a unit will incur higher overhead and memory requirement than with a procedure. This is because registers are saved more frequently when lines are used as the units of reverse execution.

In addition, the user can enable or disable *optimization*. Some inserted code for saving changed states may need to use temporary registers. The purpose of optimization is to omit saving and restoring these temporary registers at the beginning and the end of inserted code, i.e. spills of registers (Fig. 4 (2) and (3)). If optimization is disabled, spills are performed on all temporary registers. On the other hand, if optimization is enabled, our virtual machine analyzes dead registers [2] in the original code fragments. Dead registers are registers whose values are no longer used by later in-

structions. If available, we make use of such dead registers as temporary registers, and no spills are performed on them.

Optimization offers a trade-off between (forward) execution speed and accuracy against the original execution. The user can improve (forward) execution speed by enabling optimization since spills of some temporary registers will be omitted. But at the same time, since some temporary registers are not spilled in inserted code, their values may differ from the original execution. So, accuracy against the original execution is lower if optimization is enabled.

In sum, the user can select the unit of reverse execution from line or procedure and the optimization from enable or disable. These settings for the unit and optimization can of course be combined. Thus, our reversible debugger offers a grand total of four types of trade-off settings. The settings offer trade-offs between granularity, accuracy, overhead and memory requirement. The user can choose the appropriate setting flexibly during debugging without finishing and restarting the debuggee. Section 5 will show a detailed comparison between these four types of trade-off settings.

4.1.1.2 Circular Buffer.

We save states changed during execution in a circular buffer, similar to [6]. We make use of the Unix system call *mprotect* to reduce the overhead incurred by boundary checking. We protect the last page of the circular buffer and perform no boundary checking in the translated code. So, when translated code tries to save changed states to a protected page in the circular buffer, a memory violation error occurs, i.e. the buffer is full. Our debugger is given the task of detecting such a violation error. When such an error is detected, our debugger performs a wraparound of the circular buffer.

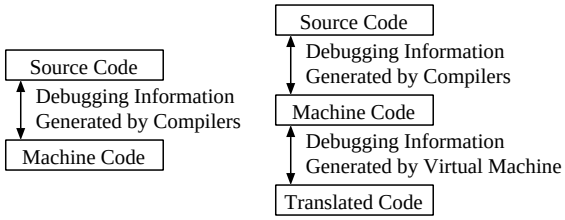
4.2 Reversible Debugger

Our reversible debugger manipulates the debuggee and our virtual machine using *ptrace* system call and */proc* file system. They are very popular debugging constructs in Unix-like operating systems, and offer very raw primitives for debugging at the native machine code level. In the following sections, we briefly describe the implementation of the functionalities of our reversible debugger according to the classification in Table 1. Our approach is very similar to traditional source code level debuggers. So, we describe mainly the parts that are different from traditional source code level debuggers.

4.2.1 Debugging Session (Table 1 (a))

Basically, our reversible debugger is the same as traditional debuggers in terms of the implementation of these functionalities. But our reversible debugger must perform the additional tasks of loading and unloading our virtual machine. When "Start" or "Attach" is used to start a new debugging session, our reversible debugger must load (or inject) our virtual machine to the debuggee's address space. Similarly, when "Kill" or "Detach" is used to finish the debugging session, our reversible debugger must unload our virtual machine from the debuggee's address space.

To achieve this, we make use of a very small *stub library* and an environment variable *LD_PRELOAD*. This stub library contains only one procedure each for loading and unloading our virtual machine and is set to *LD_PRELOAD* in advance. This indicates the system to automatically load the stub library to the programs' address space. Thus, loading or unloading our virtual machine can be easily achieved by just executing the procedures of the stub library in the debuggee's address space.



(a) Traditional Debugger

(b) Our Debugger (VM mode)

Figure 5: Debugging information.

4.2.2 Forward Execution (Table 1 (b))

As mentioned in section 3.2, these functionalities can be used in both the native mode and the VM mode. In the native mode, these functionalities can be implemented in exactly the same way as traditional debuggers. But in the VM mode, there is a problem since our reversible debugger must handle translated code in addition to native machine code and source code of debuggees. Thus, our reversible debugger introduces *extra debugging information*.

Traditional debuggers use only one type of debugging information. This information is generated by compilers and contains mapping between source code and native machine code (Fig. 5 (a)). Our reversible debugger also makes use of extra debugging information, which is generated by our virtual machine and contains mapping between native machine code and translated code (Fig. 5 (b)).

In the VM mode, machine code level manipulation over translated code is realized using information generated by our virtual machine. Then, source code level manipulation over machine code is realized using information generated by compilers. The latter (source code over machine code) is basically the same as in traditional debuggers. We thus now describe the former, i.e., machine code over translated code:

Breakpoint In our virtual machine, the entry to a translated fragment is restricted to the first instruction of that translated fragment. This means the same instruction can be translated multiple times. Fig. 6 shows an example. In Fig. 6 (a), we assume *instructions 1*, *2*, and *3* are destinations of branch instructions not shown in the figure. When the branch instruction whose destination is *instruction 1* is first executed, our virtual machine translates the fragment into one that begins with *instruction 1* (Fig. 6 (b)). Later, when the branch instruction whose destination is *instruction 2* is first executed, our virtual machine translates the fragment to produce another fragment that begins with *instruction 2* (Fig. 6 (c)), instead of letting the branch instruction jump to the middle of the *translated fragment 1* which resides in the cache buffer. Thus, in this case, *instructions 2*, *3*, and *4* are translated twice and appear in two different translated fragments.

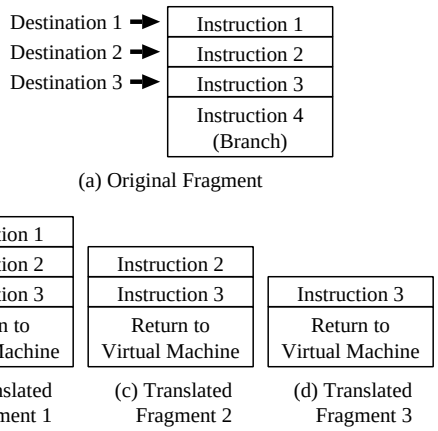


Figure 6: Multiple translated fragments.

So, when the user sets (or deletes) a breakpoint to one such (original) instruction, our virtual machine sets (or deletes) breakpoints to all code corresponding to the (original) instruction scattered in different translated fragments. For example, in the above case, when the user sets a breakpoint to *instruction 3*, our virtual machine sets breakpoints to both *instructions 3* in *translated fragments 1* and *2*. Such correspondences can be retrieved from the extra debugging information generated by our virtual machine.

But this is not enough because after setting a breakpoint to an (original) instruction and resuming the execution, the (original) instruction may be re-translated. For example, after resuming the execution in the above case, when the branch instruction whose destination is *instruction 3* is first executed, our virtual machine produces yet another translated fragment which begins with *instruction 3* (Fig. 6 (d)). In this case, there is already a breakpoint set to *instruction 3*. We need to make sure that the breakpoint is still set to *instruction 3* of *translated fragment 3*. This is done by our virtual machine at the time of translation. Our debugger passes information on the breakpoints that have been set to our virtual machine before the translation occurs.

Continue There is no special processing for "Continue" in VM mode. Just resuming execution is enough, and execution of the debuggee on our virtual machine proceeds as before.

Step Our reversible debugger performs step execution by executing translated code corresponding to one instruction of the original code. Such correspondences can be retrieved from extra debugging information generated by our virtual machine.

But when step execution is performed on a branch instruction, it may return control to our virtual machine (as described in section 4.1.1). In such a case, our reversible debugger continues execution of our virtual machine to the top of the next translated fragment (steps (i) to (iv) in section 4.1).

4.2.3 State Access (Table 1 (c))

Special handling for accessing registers and memory locations are not needed. This is because we do not *simulate* registers or memory locations of the original code with different ones in translated code. So, the mapping between original code and translated code is not needed, and we can access registers and memory locations directly.

4.2.4 Reverse Execution (Table 1 (d))

The functionalities concerned with reverse execution are as follows:

Enabling and Disabling Reverse Execution These functionalities entail switching between the native mode and the VM mode. When switching to the VM mode, we save the current execution position (PC) of the debuggee to the appropriate data area of our virtual machine. Then, we adjust the PC of the debuggee and let our virtual machine execute steps (i) to (iv) in section 4.1. Switching to the native mode is much easier. This can be done by just recovering the PC to indicate the corresponding position in the original code.

Changing Trade-off Setting When changing the trade-off setting, we first reset our reversible debugger and our virtual machine in the following manner. We let our virtual machine flush the cache buffer which contains translated code of the debuggee. The circular buffer is also flushed. Then, our reversible debugger discards all extra debugging information generated by our virtual machine. After our debugger and our virtual machine has been reset, we set our virtual machine to the new chosen trade-off setting, and let our virtual machine execute steps (i) to (iv) in section 4.1.

Reverse Execution The process of reverse execution is very straightforward. We can perform reverse execution of the debuggee by sequentially accessing the circular buffer and reconstructing the original states using information saved by our virtual machine.

But this process needs operations to the debuggee's address space, reading from the circular buffer and reconstructing the original states. These operations are repeated many times until the reverse execution is completed. Operations to other process's address space incur high overhead because they need support from the operating systems. So, if our reversible debugger performs the process of reverse execution by directly manipulating the debuggee's address space, it may incur significant and unacceptable overhead.

Thus, we prepare code that performs the process of reverse execution in our virtual machine. Reverse execution is then realized by excuting that code in the debuggee's address space. In this way, our debugger performs reverse execution avoiding direct manipulations to the debuggee's address space as much as possible.

4.3 Limitations

Currently, our reversible debugger has several limitations. First, our reversible debugger supports only frequently used system calls, such as those related to memory management and file I/O. We use an approach similar to [8] for handling file I/O.

Next, our reversible debugger does not support multi-threaded programs. Multi-threading proposes a very difficult problem since just saving changed states in each thread (such as is done in [8]) is insufficient. Such an approach allows only separate reverse execution of each thread, and the user is responsible for synchronizing backtracked threads. Separate reverse execution of each thread is interesting and useful in some cases, but in most cases, manual synchronization of backtracked threads is a tedious and erroneous process.

To backtrack the whole program consistently, we need to retrieve the precise order of each state change that happens in different threads. The simplest way is to use a single circular buffer shared by all threads. But this approach needs synchronizations of accesses to the buffer each time a state changes, and can be extremely slow. Thus, we are currently investigating the introduction of a user level threading scheme to our virtual machine. This enables complete control over the scheduling of threads. We expect that such a user level threading scheme makes arranging and deriving the order of state changes much easier.

5. EVALUATION

This section describes the results of experiments we conducted. All measurements were done on a computer with Pentium4 2.4 GHz and 512 MBytes memory. The main components of the system we used were as follows: Linux Kernel 2.4.27, gcc 2.95.4, and glibc 2.3.2.

5.1 Comparison with LVM

As described in section 2, existing works which use the virtual machine based approach have two serious problems: (1) compatibility and (2) efficiency. The compatibility problem is solved by using native machine code as debuggees. In this section, we give evaluation results related to the efficiency problem.

We compare our debugger with LVM [9] using variants of the Stanford integer benchmark suite. We modified a very small amount of benchmark code related to standard I/O and memory allocations for the evaluation of LVM, due to LVM's compatibility problem. But effects on the overall execution time can be ignored. Since trade-offs provided by LVM and our debugger are not the same, we compare the base execution speeds of the virtual machines without saving the changed states.

Table 2 shows the results. Overhead is the slowdown factor of the base execution speed without saving the changed states over the direct execution speed on the real CPU. As Table 2 shows, execution on LVM even without saving changed states (i.e. normal execution for LVM) incurs on average a slowdown of 80.8 times. In contrast, our virtual machine incurs only a slowdown of 1.7 times. Thus, we achieve a speed up of 47.5 times over LVM. This is because our virtual machine performs execution of debuggees and saving of changed states directly by native machine code.

Note that we compared the base execution speeds without saving changed states. Execution speed when saving changed states incurs an overhead of several times more in both LVM and our debugger. These slowdowns are affected by many other factors, such as granularity, accuracy, and memory requirement. We show this in the next section.

Table 2: Comparison with LVM [times].

Overhead		Speed up
LVM	Our VM	of our VM
80.8	1.7	47.5

5.2 Comparison of Trade-offs

As described in section 4.1.1.1, users of our reversible debugger can consider trade-offs between granularity, accuracy, overhead and memory requirement. The user can choose the appropriate trade-off setting by designating settings for *unit* and *optimization*. In this section, we compare trade-offs provided by these settings.

To evaluate overhead and memory requirement, we used SPEC CPU2000 benchmarks [14]. We selected eight integer benchmarks (CINT), *gzip, vpr, mcf, crafty, parser, gap, vortex, bzip2*, and four floating point benchmarks (CFP), *mesa, art, equake, ammp*. We excluded benchmarks written in C++ and Fortran. We also excluded 3 benchmarks which use system calls unsupported in our debugger. We ran every benchmark using input data of training size. Table 3 shows the results.

In the first column, "line" and "proc" denote that line or procedure was selected as the unit of reverse execution. In the second column, "opt" and "noopt" denote that enabling or disabling was selected for optimization. Overhead is the slowdown factor of execution speed with saving changed states on our virtual machine vs execution speed on the real CPU. Memory requirement is the required storage area per line if the unit is line or per procedure if the unit is procedure. For the case of unit of procedure, we also denote normalized memory requirement per line in parentheses.

As Table 3 shows, by changing the unit from line to procedure, the overhead of CINT is reduced from 28.4 times to 13.3 times, and the overhead of CFP is reduced from 23.8 times to 7.1 times when optimization is disabled. Also, the overhead of CINT is reduced from 24.8 times to 9.0 times and the overhead of CFP is reduced from 22.2 times to 6.1 times when optimization is enabled. Thus, execution speeds up by two to three times. In addition, the memory requirement of CINT is reduced from 59.5 bytes to 16.3 bytes, and the memory requirement of CFP is reduced from 114.9 bytes to 27.0 bytes. Thus, memory requirement is reduced by three to four times. But these improvements are at the consequence of granularity.

As Table 3 shows, by enabling optimization, the overhead of CINT is reduced from 28.4 times to 24.8 times, and the overhead of CFP is reduced from 23.8 times to 22.2 times when the unit is line. Also, the overhead of CINT is reduced from 13.3 times to 9.0 times, and the overhead of CFP is reduced from 7.1 times to 6.1 times when the unit is procedure. Thus, execution speeds up by 7.2% - 47.7%. But these improvements sacrifice accuracy.

As shown above, there are very difficult trade-off problems between granularity, accuracy, overhead and memory requirement. One of the advantage of the virtual machine based approach is its flexibility. In our reversible debugger, the user can choose the appropriate trade-off setting flexibly during debugging without finishing and restarting the debuggee. For example, users can first use the unit of procedures, and later change to the unit of lines when the parts of code suspected to have bugs are narrowed down. Users

Table 3: Comparison of trade-offs.

Trade-off Setting				Overhead [times]		Memory Requirement [bytes]	
Unit	Optimization	Granularity	Accuracy	CINT	CFP	CINT (per Line)	CFP (per Line)
line	noopt	small	high	28.4	23.8	59.5	114.9
line	opt	small	low	24.8	22.2	59.5	114.9
proc	noopt	large	high	13.3	7.1	414.7 (16.3)	4943 (27.0)
proc	opt	large	low	9.0	6.1	414.7 (16.3)	4943 (27.0)

can also normally enable optimization, but for code where dead register values are important (such as when registers are allocated to variables in source code), they can disable optimization. Since reverse execution has a very large cost, such flexibility to consider trade-offs is very important.

5.3 Overhead due to the Virtual Machine

In our reversible debugger, there are two main sources of overhead, the generated code runs slower than the original code (step (v) in section 4.1), and the cost of code generation (steps (i), (ii), (iii), (iv), and (vi) in section 4.1). We call the former *execution overhead* and the latter *translation overhead*. The main problem of the virtual machine based approach compared to the static instrumentation approach is the translation overhead. We can consider that the execution overhead is about the same as the entire overhead of the static instrumentation approach, if the same trade-off is chosen. So, the overhead due to the virtual machine based approach itself is mainly caused by the translation overhead. We consider the percentage that the total translation overhead occupies within the entire execution time, assuming that the total translation overhead is constant regardless of the trade-off setting chosen.

We calculate the translation overhead by first considering the entire execution time of our virtual machine when no state saving is performed. This execution time consists of the execution time of the generated code (the execution time of the original code plus the execution overhead), and the translation overhead. Since no state saving is performed in the generated code, we can assume that the execution time of the generated code is the same as the execution time when execution is performed directly on the real CPU. Thus, we can approximate the translation overhead by subtracting the execution time when execution is performed directly on the real CPU from the entire execution time of our virtual machine when state saving is not performed.

Table 4 shows the percentage of the translation overhead within the entire execution time for each trade-off setting. As Table 4 shows, the percentages are low: under 5% when lines are used as the units, and under 11% when procedures are used as units [4]. This is because our virtual machine performs *caching* and *linking* of the translated fragments [13] as described in section 4.1.1. They cache translated fragments in the internal buffer, and link between translated fragments directly by branch instructions. So, as execution proceeds, the number of times steps (i), (ii), (iii), (iv) and (vi) in section 4.1 happen is greatly reduced. Thus, their contribution to the entire execution time becomes very small as shown in the above results. Therefore, we can consider that the overhead incurred by our virtual machine itself (the translation overhead) to be unproblematic.

[4]Note that the percentages are higher when procedures are used only because its entire execution time is smaller.

Table 4: Overhead of our VM [%]

Benchmark	line/noopt	line/opt	proc/noopt	proc/opt
CINT	3.8	4.5	7.5	10.7
CFP	1.1	1.3	3.2	3.7

6. CONCLUSIONS

We proposed a novel reversible debugger using the virtual machine based approach. Our approach addressed the problems of (1) compatibility and (2) efficiency that existed in previous works. We used native machine code as the target of our reversible debugger and made use of dynamic translation. In addition, our approach provides two execution modes: the native mode and the VM mode. This is made possible by not destroying live registers or memory locations in our virtual machine and keeping them "in the clear" for the original execution.

Currently, our debugger provides four types of trade-off settings (designated by unit and optimization) to consider trade-offs between granularity, accuracy, overhead and memory requirement. The user can flexibly choose the appropriate setting during debugging while considering the trade-offs of the choices, without finishing and restarting the debuggee.

Future works include support of multi-threaded programs as mentioned in section 4.3. Behaviors of multi-threaded programs are much more complex than single-threaded programs. We believe that supporting multi-threaded programs in reversible debuggers can be a great help to programmers.

7. ACKNOWLEDGMENTS

This study was supported by the Special Coordination Funds of the Ministry of Education, Culture, Sports, Science and Technology of the Japanese Government.

8. REFERENCES

[1] H. Agrawal, R. A. DeMillo, and E. H. Spafford. An execution-backtracking approach to debugging. *IEEE Software*, 8(3):21–26, 1991.

[2] A. V. Aho, R. Sethi, and J. D. Ulman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[3] T. Akgul and V. J. Mooney. Instruction-level reverse execution for debugging. In *Proc. of Workshop on Program Analysis for Software Tools and Engineering*, pages 18–25, 2002.

[4] T. Akgul, V. J. Mooney, and S. Pande. A fast assembly level reverse execution method via dynamic slicing. In *Proc. of 26th International Conference on Software Engineering*, pages 522–531, 2004.

[5] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proc. of*

Conference on Programming Language Design and Implementation, pages 1–12, 2000.

[6] S. Chen, W. K. Fuchs, and J. Chung. Reversible debugging using program instrumentation. *IEEE Trans. on Software Engineering*, 27(8):715–727, 2001.

[7] B. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. In *Proc. of Conference on Measurement and Modeling of Computer Systems*, pages 128–137, 1994.

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

[9] C. Demetrescu and I. Finocchi. A portable virtual machine for program debugging and directing. In *Proc. of Symposium on Applied Computing*, pages 1524–1530, 2004.

[10] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proc. of Symposium on Principles of Programming Languages*, pages 297–302, 1984.

[11] H. Lieberman and C. Fly. Bridging the gulf between code and behavior in programming. In *Proc. of Conference on Human Factors in Computing Systems*, pages 480–486, 1995.

[12] T. Moher. Provide: A process visualization and debugging environment. *IEEE Trans. on Software Engineering*, 14(6):849–857, 1988.

[13] K. Scott, N. S. Velusamy, B. Childers, J. Davidson, and M. L. Soffa. Retargetable and reconfigurable software dynamic translation. In *Proc. of International Symposium on Code Generation and Optimization*, pages 36–47, 2003.

[14] The Standard Performance Evaluation Corporation (SPEC). http://www.specbench.org.