Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging *

Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews and Yuanyuan Zhou
{smsriniv, kandula, crandrws, yyzhou}@cs.uiuc.edu

Department of Computer Science
University of Illinois, Urbana-Champaign
Urbana, IL 61801

ABSTRACT

Software robustness has significant impact on system availability. Unfortunately, finding software bugs is a very challenging task because many bugs are hard to reproduce. While debugging a program, it would be very useful to rollback a crashed program to a previous execution point and deterministically re-execute the "buggy" code region. However, most previous work on rollback and replay support was designed to survive hardware or operating system failures, and is therefore too heavyweight for the fine-grained rollback and replay needed for software debugging.

This paper presents *Flashback*, a lightweight OS extension that provides fine-grained rollback and replay to help debug software. Flashback uses shadow processes to efficiently roll back in-memory state of a process, and logs a process' interactions with the system to support deterministic replay. Both shadow processes and logging of system calls are implemented in a lightweight fashion specifically designed for the purpose of software debugging.

We have implemented a prototype of Flashback in the Linux operating system. Our experimental results with micro-benchmarks and real applications show that Flashback adds little overhead and can quickly roll back a debugged program to a previous execution point and deterministically replay from that point.

1 Introduction

As rapid advances in computing hardware have led to dramatic improvements in computer performance, issues of reliability, maintainability, and cost of ownership are becoming increasingly important. Unfortunately, software bugs are as frequent as ever, accounting for as much as 40% of computer system failures [45]. Software bugs may crash a production system, making services unavailable. Moreover, "silent" bugs that run undetected may corrupt valuable information. According to the National Institute of Standards and Technology [48], software bugs cost the U.S. economy an estimated $59.5 billion annually, approximately 0.6% of the gross domestic product! Given the magnitude of this problem, the development of effective debugging tools is imperative.

Software debugging has been the focus of much research. Popular avenues of such research include detection and analysis of data races [7, 23, 46, 63, 68, 69, 74], static compiler-based techniques to detect potential bugs [20, 24, 31, 36, 64, 76] possibly aided by static checking of user-directed rules [19, 27, 81], run-time checking of data types to detect some classes of memory-related bugs [41, 49], and more extensive run-time checks to detect more complex program errors [28, 51]. These studies have proposed effective solutions to statically or dynamically detect certain types of software bugs.

Even though previous solutions have shown promising results, most software bugs still rely on programmers to interactively debug using tools such as gdb. Interactive debugging can be a very challenging task because some bugs occur only after hours or even days of execution. Some of them occur only with a particular combination of user input and/or hardware configurations. Moreover, some bugs, such as data races, are particularly hard to find because they only occur with a particular interleaved sequence of timing-related events.

These problems motivate the need for low-overhead debugging support that allows programmers to rollback to a previous execution point and re-execute the buggy code region. A deterministic replay recreates the precise conditions that lead to the bug and helps to understand the causes of the bug. In most debugging tools today, if an error occurs, the program needs to be restarted from the very beginning and may take hours or even days to reach the buggy state. If the bug is time-related, the bug may not occur during re-execution. It would be very useful if an interactive debugger such as gdb can periodically checkpoint the process state of the debugged program during its dynamic execution. If an error occurs, the programmer can request gdb to rollback to a previ-

*This work was supported in part by NSF under grants CCR-0325603, EIA-0072102, and CHE-0121357; by DARPA under grant F30602-01-C-0078; by an IBM SUR grant; and by additional gifts from IBM and Intel.

ous state and then deterministically replay the program from this state so that the programmer can see how the bug manifests in order to catch its root cause.

Though system support for rollback and replay has been studied in the past, most previous approaches are too heavy-weight to support software debugging. The main reason is that these approaches are geared toward surviving hardware or operating system failures. Therefore, most of these systems checkpoint program state to secondary storage such as disk, remote memory or non-volatile memory [3, 10, 12, 34, 37, 38, 39, 54, 61, 77, 79, 82]. Correspondingly, these systems incur far higher overhead than is necessary or permissible to support software debugging. Unlike hardware/OS failures, we only need to rollback and replay a program when it crashes due to software bugs. Moreover, most previous systems cannot afford frequent checkpointing because of the high overheads involved in these approaches. As a result, applications may have to roll back to a point in the distant past (e.g., 1-2 hours ago).

Besides checkpointing systems, other work on rollback support – such as transaction support for main-memory data structures [11, 29, 40, 43, 60, 62], system recovery [9, 42, 65, 72] or logging and replay of system events [6, 33, 50, 66, 70] – either have problems similar to previous checkpointing systems or require applications to be rollback-aware. These limitations hinder the effectiveness of these solutions for software debugging of general programs.

In this paper, we present a lightweight OS extension called *Flashback* that provides rollback and deterministic replay support for software debugging. In order to efficiently capture the in-memory state of an executing process, Flashback uses *shadow processes* to replicate a program's execution state. Moreover, Flashback also captures the interactions between a program and the rest of the system – such as system calls, signals, and memory mapped regions – to allow for subsequent deterministic re-execution. We have developed a prototype of our proposed solution in the Linux operating system that implements a subset of the features. Our experimental results with micro-benchmarks and real applications show that our system adds little overhead and can quickly roll back to a previous execution point.

As an example of how deterministic replay support can be used for debugging, we also explore the necessary extensions to gdb in order to provide user support for checkpointing, rollback and deterministic replay. These extensions will allow programmers to roll back a program to a previous state when something has gone awry, and interactively replay the buggy code region. With such support, the programmer does not need to restart the execution of the program or to worry about the reproducibility of the bug.

This paper is organized as follows. Section 2 describes the motivation and background of our work. Section 3 presents an overview of Flashback, and sections 4 and 5 describe in greater detail our approach for rollback of state and deterministic replay. Section 6 presents the experimental results. Section 7 discusses the modifications that have been made to gdb in order to control logging, rollback and recovery from within the debugger. Section 8 concludes the paper with a brief discussion of our experience as well as plans for future work.

2 Background and Related Work

Our work builds upon two groups of research: system support for debugging and system support for rollback. In this section we discuss closely related work done in these two directions.

2.1 System Support for Debugging

Software debugging has been the subject of substantial research and development. Existing approaches mainly include compile-time static checking, run-time dynamic checking and hardware support for debugging. Some representative compile-time static checkers were proposed by Wagner [75, 76], Lujan [44] Evans [21], Engler [19, 27, 81]. Examples of run-time dynamic checkers include Rational's Purify [30], KAI's Assure [35], Lam et. al.'s DIDUCE [28, 51] and several others [41, 49, 15, 53, 58, 63]. Recently, several hardware architecture techniques have been proposed to detect bugs [26, 1, 14, 47, 56].

While these compile-time, run-time or hardware techniques are very useful in catching certain types of bugs, many bugs still cause the programmer to rely on interactive debuggers such as gdb. To characterize timing-related bugs such as race conditions, simply rerunning the program with the same input may not reproduce the same bug. Moreover, some bugs may appear only after running the program for several hours, making the debugging process a formidable task. To understand and find root causes of such bugs, it is very useful to provide system support for reproducing the occurring bug, which may only appear for a particular combination of user inputs and configurations or after a particular interleaved sequence of time-related events.

One effective method to reproduce a bug is to roll back to a previous execution state in the vicinity of the buggy code, and deterministically replay the execution either interactively inside a debugger or automatically with heavy instrumentation. This requires an efficient rollback and deterministic replay mechanism.

2.2 System Support for Rollback

Rollback capability is provided in many systems including checkpointing systems, main-memory transaction systems and software rejuvenation.

Checkpointing has been studied extensively in the past. Checkpointing enables storing the previous execution state of a system in a failure-independent location. When the system fails, the program can restart from the most recent checkpoint in either a different machine or the same machine after fixing the cause of the failure. Since most checkpointing systems assume that the entire system may fail, checkpoint data is stored either in disks [12, 34, 37, 38, 39, 79, 61], remote memory [3, 54, 82] and non-volatile or persistent memory [10, 80]. As a result, most checkpoint systems incur high overhead and cannot afford to take frequent checkpoints. They are, therefore, too heavy-weight to support rollback for software debugging.

Systems that provide transaction support for main-memory data structures also allow applications to rollback to a previous execution point [11, 29, 43, 60, 62]. For example, Lowell and Chen have developed a system that provides transaction support in the Rio Vista recoverable virtual memory system [11, 43]. Most of these approaches require applications to be written using the transaction programming model; consequently they cannot be conveniently used for debugging a general program.

Borg et al developed a system [5] that provides fault tolerance by maintaining an inactive backup process. In the event of a system failure, the backup process can take over the execution of a process that crashes. The backup process is kept up-to-date by making available to it all the messages that the active process received. Their implementation is based on the assumption that two processes starting from the same initial state will perform identically upon receiving the same input. While this assumption holds for recovery-based systems, it is not the case for general software since the state of the rest of the system may have changed in the meantime. Deterministic replay of a process requires that it receive the same non-deterministic events during replays as during the original run. These events include responses to system calls, shared memory accesses, signals, network messages et al.

Recovery-oriented computing [25, 52] is a recent research initiative that adopts the approach that errors are inevitable, so support for recovering from errors is essential for developing and validating highly available software. Though this is an interesting approach to software availability, most studies in software rejuvenation so far [4, 32, 57] have focused on restarting the whole application rather than fine-grained rollback. Crash-only software [8] is a recent approach to software development that improves the availability of software by using component building blocks that can crash and restart quickly instead of aiming for fault tolerance. These studies focus more on minimizing mean-time-to-recovery (MTTR) than on software debugging.

Feldman and Brown developed a system for program debugging [22] that periodically checkpoints the memory state of a process by keeping track of pages touched by the process. They propose using this system for program restart and comprehensive execution path logging. But their mechanism involves changes to the compiler, loader, standard library and the kernel. It tracks all memory accesses via code instrumentation and thereby this approach is very heavy-weight. Further, they do not provide deterministic replay; therefore, some errors may not manifest themselves during subsequent re-execution.

Russinovich[59] suggests a lightweight approach to log nondeterministic accesses to shared memory by merely replaying the interleaved order of processes sharing the memory deterministically. An application is instrumented to obtain fine-grained software instruction counters and the OS has to record the location of context switches. This technique can be potentially used by FlashBack to support the replay of shared-memory multi-processed program.

ReVirt[17] is a novel approach to intrusion analysis that encapsulates applications within a virtual OS that itself runs as a process in the guest OS. This technique decreases the size of the trusted computing base (TCB) and allows precise logs to be maintained by the guest OS. Flashback is significantly different from ReVirt. First, debugging support needs to checkpoint application state on timescales(minutes) that are several orders of magnitude smaller than in ReVirt(days). Second, unlike ReVirt which has to contend with malicious intruders by logging "everything", Flashback need only log changes that are made by the application being debugged and external events that affect its operation.

The constraints with existing system support for rollback motivate the need for a new lightweight, fine-grained rollback and deterministic replay solution specifically designed for software debugging.

3 Overview of Flashback

Flashback provides three basic primitives for debugging, *Checkpoint()*, *Discard(x)* and *Replay(x)*.

- *stateHandle = Checkpoint ()*: Upon this call, the system captures the execution state at the current point. A state handle is returned so that the program can later use it for rollback.
- *Discard (stateHandle)*: Upon this call, the captured execution state specified by *stateHandle* is discarded. The program can no longer roll back to this state.
- *Replay (stateHandle)*: Upon this call, the process is rolled back to the previous execution state specified by *stateHandle* and the execution is deterministically replayed until it reaches the point where *Replay()* is called.

To provide the above primitives, Flashback uses *shadow processes* to efficiently capture the in-memory execution state of a process at the specified execution point. The main idea of shadow process is to fork a new process at the specified execution point and this new process maintains the copy of the process's execution state in main memory. Once a shadow process is created, it is suspended immediately. If rollback is requested, the system kills the current active process and creates a new active process from the shadow process that captured the specified execution state. Since Flashback does not attempt to recover from system crashes or hardware failure, there is no need to store the shadow process onto disk or other persistent storage. This reduces the overhead of the checkpoint process significantly. Moreover, copy-on-write is used to further reduce the overhead.

While our method of checkpointing allows the in-memory state of a process to be reinstated, the process may not see the same set of open file descriptors or network connections during re-execution. Even if the state of file descriptors can be reproduced, it is still a cumbersome task to restore the contents of the file to the original state, and to ensure that network connections will respond exactly as in the original execution. Similarly, during replay it may be undesirable to let the process affect the external environment again by, say, deleting files or modifying their content.

In order to support deterministic replay of a rolled back process, we adopt an approach wherein we record all interactions that the executing process has with the environment. During replay, the logged information is used to ensure that the re-execution appears "identical" to the original run. When a checkpoint is initiated using the `checkpoint` primitive, in addition to capturing in-memory execution state, the system also records the interactions between the process and its environment. During replay, the previously collected information is used to give the process the *impression* that the external environment is responding exactly as it did during the original execution, and that it is affecting the environment in the same way.

Shadow processes can be used in conjunction with the deterministic replay mechanism either within a debugging environment like gdb, or through explicit calls made by the program being debugged:

- *Interactive debugging:* One possible usage scenario is where the debugging platform can periodically capture the state of an executing process by invoking `checkpoint` (similar to the insertion of breakpoints in gdb, for instance). If an error occurs, the programmer can then instruct the debugger to roll back execution to a previously captured state by specifying the time of the earlier checkpoint.

- *Explicit checkpointing and rollback:* An alternate usage scenario is that the programmer takes control of when checkpoints are taken in the code. Figure 1 shows an example of a program where the programmer has inserted explicit invocations to `checkpoint`, `replay` and `discard` primitives.

Automatic checkpoint/rollback support inside an interactive debugger is convenient and requires no changes to the program source code. On the other hand, giving the programmer explicit control on checkpoints/rollbacks enables more intelligent and meaningful checkpoint generation.

```
     .
1    checkpoint(1);
2    fd = open("file.dat", O_WRONLY, 0);
     .
     .
3    n = write(fd, buf, 80);
4    close(fd);
     .
     .
5    fd = open("file.dat", O_RDONLY, 0);
     .
     .
6    n2 = read(fd, buf2, 80);
7    if (n2 > 0)
8        discard(1);
9    else
10       replay(1);
```

Figure 1: Code for a process augmented with primitives

Figure 1 shows a program in which the programmer calls `checkpoint` in line 1. If the read operation in line 6 fails, the programmer can roll back to the execution state captured at line 1. To help characterize the bug, the execution from line 1 to line 6 can be replayed deterministically by attaching an interactive debugger or switching to a profiling mode with extensive instrumentation. If line 6 succeeds, the checkpoint is discarded.

4 Rollback Using Shadow Processes

4.1 The Main Idea

Flashback creates checkpoints of a process by replicating the in-memory representation of the process in the operating system. This snapshot of a process, known as the *shadow process*, is suspended immediately after creation and is stored within the process structure. A shadow process represents the passive state of the executing process at a previous point, and can be used to unwind the execution of the process by replacing the new execution state with the shadow state and commencing execution in the normal fashion. If a shadow state is not needed anymore, the process can discard it.

The creation of a shadow process for a running process, an event we refer to as *state-capture*, is achieved by creating a new shadow process structure in the kernel, and initializing this structure with the contents of the original process' structure. The state information captured includes process memory (stack, heap), registers, file descriptor tables, signal handlers and other in-memory state associated with a process. A pointer to this shadow structure is then stored in the original

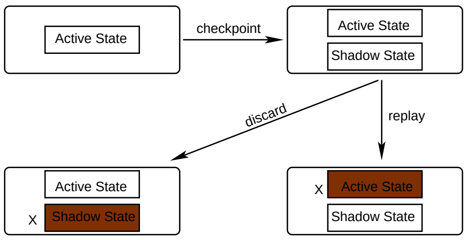


Figure 2: Effect of the primitives on the state of an executing process. When checkpoint is invoked, the process makes a clone of its execution state. Upon discard the shadow is removed; if a rollback occurs, the original execution state is discarded.

process' structure. The new representation of a process with its shadow process is shown in figure 2.

The checkpoint, discard and replay calls are either automatically generated by the debugging infrastructure at specific intervals, or inserted by the programmer in the source code (as shown in the example in the previous section). In case of a discard, the system discards the specified shadow state. If a checkpoint is requested, the system creates a new shadow of the current state and stores it. In the case of a rollback, the process rolls back the execution state to the previously generated shadow process. Figure 2 illustrates the effect of these primitives on the state of a process.

It is possible to maintain two or more shadow processes for an executing process. Multiple shadow processes are useful for progressive rollback and re-execution during debugging [78]. In some cases, when an error occurs, rolling back to only the most recent execution point before replay may not be enough to catch the root of the bug because it could have happened before this execution point. Therefore, it is necessary to roll back further and deterministically restart from an earlier execution point. It is also possible to roll back to the same shadow multiple times and cause additional checkpoints to be taken during replay.

To reduce overhead, shadow process state is maintained using copy-on-write. In other words, state capture proceeds through the creation of an in-memory copy-on-write map of the current state. When a shadow process is created, the virtual memory of the process is marked as read-only. A first write to any page by the active process would trigger the creation of a copy of the old data. This optimization has a couple of benefits. First, the time to create a shadow is significantly reduced by eliminating the need t0 copy possibly large amounts of memory state. Second, a shadow process occupies little space(in memory). Third, multiple shadows created at different execution points do not need to maintain duplicate copies of the state. Finally, the significant overlap in memory pages between a shadow process and the active process minimizes the impact on the paging behavior of the process due to discard/replay of state. However, writes onto copy-on-write protected memory during execution of the main process does incur overhead. Fortunately, our experimental results presented in Section 6 show that these overheads are not significant.

4.2 Rolling back multi-threaded processes

Rollback of a multi-threaded process requires special attention. This is because in a multi-threaded environment several components of the process state are implicitly shared across all threads that belong to the same process. For example, threads implemented using the pthread package on Linux, share memory, file descriptors and signal handlers with each other. The only thread-private states are user-space (and kernel) stacks. Such implicit sharing vastly complicates rollback because it is no longer possible for a thread to revert to pristine versions of the shadow state without impacting the execution of other threads.

There are two approaches to support fine-grained rollback of multi-threaded programs. One is to capture the process state for the entire process and roll back all threads to a previous execution point. The second approach is to track thread dependencies such as memory read-write and file read-write dependencies and roll back only those threads that depend on the erroneous thread [2, 16, 18, 67, 70].

Flashback uses the first approach to support rollback of multi-threaded programs. In other words, the underlying system captures the execution state of all threads of a process at a checkpoint. Likewise, when a rollback occurs, Flashback re-instates the execution state of all threads by reverting back to a pristine copy of the shared state. This enables maintenance of consistent state among all threads. Thread synchronization primitives, such as acquiring/releasing locks and semaphore operations are also implicitly rolled back.

Our approach has several advantages over the alternative for software debugging, even though rolling back all the threads of a process when only one of them encounters an error, may seem inefficient. First, our approach is simpler because it does not require complicated logic to keep track of thread dependencies. Tracking thread dependencies is very difficult because concurrent accesses to shared memory are not handled through software or some specialized cache coherence controller. Tracking dependencies requires either hardware support or instrumentation of application binary code to notify the operating system about data sharing. The logic to track dependencies adds overheads to the error-free execution and is also error-prone. Second, to characterize thread synchronization or data races, it might be more informative to roll back all threads and deterministically re-execute all threads step-by-step interactively. Furthermore, the inefficiency of rolling back all threads is encountered only when faults occur - the less common case, while dependency tracking, if done dynamically would lead to overhead on the common case.

4.3 Implementation in Linux

We have modified the *Linux 2.4.22* kernel by adding three new system calls – `checkpoint()`, `discard()` and `replay()` to support rollback and replay. The kernel handles these functions as described earlier. The overhead of these system calls on normal process execution is an important consideration in our implementation.

To capture shadow state, we create a new process control block (`task_struct` in Linux terminology) and initialize it with a copy of the calling process's own structure. This copy involves creation of copy-on-write maps over the entire process memory via the creation of new *mm_struct*, *file_struct* and *signal_struct*. The register contents of the current execution context when it was last in user-space are copied onto the new control block and finally the kernel stack of the new control block is initialized by hand such that the shadow process, when executed, continues execution by returning from the checkpoint system call with a different return value.

The state capture procedure is different from the fork operation in several ways. The primary difference is that after a fork operation, the newly created process is visible to the rest of the system. For instance, the module count is incremented to reflect the fact that the child process is also sharing the same modules. The newly created process is added to the scheduler's run lists and is ready to be scheduled. In contrast, a shadow process is created only for maintaining state. It is not visible to the rest of the system and does not participate in scheduling.

After capturing a shadow state, the calling process returns from the system call and continues execution as normal, with the shadow image in tow. Any changes made to the state after the checkpoint leave the shadow image in its pristine state.

A call to the `discard()` system call deletes a process's shadow image and releases all resources held by it. The `replay()` system call, on the other hand, drops the resources of the current image, and overwrites the process control block with the previously captured shadow image. Since the memory map of the current process changes during the call, the page tables corresponding to the new *mm_struct* are loaded by a call to *switch_mm*.

A subtle result of reinstating the shadow image is that the `replay()` system call never returns to the caller. As soon as the shadow becomes active for the caller, the return address for the `replay()` call is lost (it was part of the speculative state), being replaced instead with the return address of the `checkpoint()` call that corresponds to the state that the process is rolling back to.

When we implemented rollback support for multithreaded programs in Linux, we encountered many challenges because of the design of Linux thread package that our implementation is based on: pthreads. In this thread package, there is a one-to-one mapping between user-space and kernel-space threads, i.e. each user-space thread has an executable process counterpart inside the kernel. State sharing is achieved by using the *clone* system call to create lightweight processes that share access to memory, file descriptors and signal handlers among other things. POSIX compliance, with respect to delivery of signals (and other requisites), is ensured by creating an LWP thread manager that is the parent of all the threads (LWP's) associated with a process. While the one-to-one mapping allows the thread library to completely ignore the issue of scheduling between threads at user-space, it presents several complications for rollback.

Recall that when one thread attempts to process a checkpoint event, we need to capture the state of all the other threads of that process. Since every user-space thread is mapped to a kernel thread, the other threads may be executing system calls or could be blocked inside the kernel waiting for asynchronous events (sleep-SIGALRM, disk IO etc.). Capturing the transient state of such threads could easily lead to state inconsistency upon rollbacks, such as rolling back to a *sleeping* state when the corresponding kernel timer has already expired[1]. It is difficult to capture the state of an execution context from within a different execution context.

We are currently exploring a solution to this problem by explicitly identifying such troublesome scenarios and manipulating the user-space and kernel stacks to ensure that the interrupted system call is re-executed upon rollback. Specifically, threads that are blocked in system calls are checkpointed *as if* they are about to begin execution of this interrupted system call.

Notice that apparently simple solutions that circumvent this problem such as using inter-process communication or explicit barrier synchronization prior to state capture are not applicable. In the former case, IPC mechanisms such as signals and pipes increase the latency of the state capture event because their processing is usually deferred, and is often not deterministic. Barrier synchronization on the other hand, would cause the processing of a state capture event to be delayed until the event is generated on all the threads of a process, which might be unrealistic in certain applications.

5 Replay Using Record-and-Sandbox

5.1 The Main Idea

In order to deterministically replay the execution of a process from a previous execution state, we need to ensure during re-execution that the process perceives no difference in its interaction with the environment. For instance, if the process did a `read` on a file and received a particular array of bytes, during replay, the process should receive the same array of bytes and return value as before, though the file's contents may have already been changed.

[1] sleep on Linux is implemented using nanosleep which swaps out the process after adding a timer onto the kernel's timer list

Flashback does not ensure exactly the same execution during replay as during the original run. Instead, Flashback provides only an impression to the debugged process that the execution and interaction with the environment appears identical to those during the original run. It is difficult to provide the exact same execution because the external environment, such as network connections or device states, etc, is beyond the control of the operating system. As long as Flashback interacts with the debugged process in the same way, with very high probability, the bug can be reproduced during replay.

A process in Flashback can operate in one of two modes - *log* and *replay*. In the *log* mode the system logs all interactions of the process with the environment. These interactions can happen through system call invocations, memory-mapping, shared memory in multi-threaded processes, and signals. The process enters the log mode when the `checkpoint` primitive is invoked. In the replay mode, the kernel handles system interactions of the process by calling functions that simulate the effect of the original system call invocation. The replay mode is selected when the `replay` primitive is invoked. In this mode, Flashback ensures the interaction between the replayed process and the OS is the same as was logged during the original run.

5.2 System calls

Logging and replay are different for different types of system calls:

- Filesystem-related – Calls such as open, close, read, write, seek
- Virtual memory-related, such as memory allocation, mmap etc.
- Network-related – such as socket creation, polling, send, recv etc.
- Process control – such as exec, fork, exit, wait
- Interprocess communication-related – such as creation and manipulation of message queues and named pipes
- Utility functions – such as getting the time of day

When simulating the effect of a system call, Flashback has to ensure that the values returned by the system call are identical to those returned during the original execution. In addition, the original system call may return some "hidden" values by modifying memory regions pointed to by pointer arguments. For example, the read() system call loads the data from the file system into the buffer specified by one of the parameters. These side effects also need to be captured by Flashback. A faithful replay of a system call thus requires Flashback to log all return values as well as side-effects. While somewhat tedious because of the special attention required by each system call to handling its specific arguments, this support can easily be provided for a large body of system calls.

In Flashback, we intercept system calls invoked by a process during its execution. In order to do this, we replace the default handler for each system call with a function that does the actual logging and replay as shown in figure 3. In logging mode, the function invokes the original call and then logs the return values as well as the side-effects. In replay mode, the function checks to confirm that the same call is being made again, and then makes the same side-effects and returns the logged return value.

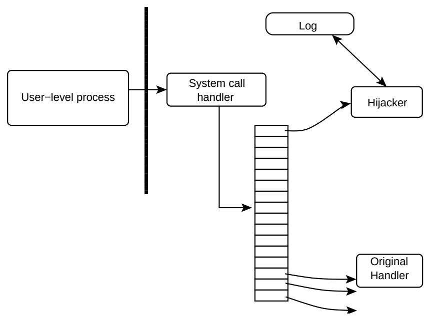


Figure 3: Hijacking System Calls for Logging and Replay in Flashback

A notable exception to bypassing the actual system calls during replay is for calls related to memory management, such as memory mapping and heap management. In this situation we cannot fake memory allocation – if the process accesses a memory location that we have faked the allocation of, then it will result in a segmentation fault. This problem arises because while memory is allocated and deallocated using the `brk()` system call, it may be accessed through direct variable assignments. The changes made to memory locations do not make any permanent changes to the system; i.e. the state is captured by a process' checkpoint exclusively. As we discuss shortly, however, this may not be the case for files that have been mapped into memory.

Once system calls have been handled, much of the process' original execution can be replayed. For instance, the process being replayed can read data from files as it did before even though these files may actually have been modified or may not even exist in the system anymore. Similarly, it will receive network packets as it originally did from remote machines. As far as the process is concerned, it believes that these events are happening as they did before in terms of both actual data exchanged and the relative timing of asynchronous events.

5.3 Memory-Mapped Files and Shared Memory

Linux supports two different flavors of shared memory for interprocess communication – System V IPC and BSD mmap. These implementations allow processes to share a single chunk of memory by mapping the shared memory onto their respective memory spaces. BSD mmap allows processes to map a previously opened file into a region of its memory, after which it can access the file using simple memory assignment instructions. When a shared segment is requested, the kernel forces the memory management unit (MMU) to generate a page fault every time a previously unused section of this memory region is accessed. In response to the page fault, the kernel loads one page of data from the file and reads it into the process' memory.

A file may be mapped as either *private* or *shared*. Any changes made to privately mapped files are visible only to that process and do not result in changes to the file. On the other hand, files that are mapped as shared may be modified when the process writes to the memory area. Further, for shared files, changes made to the file by a processes will be immediately visible to other processes that have mapped the same region of the file. Providing replay for shared memory poses problems as a process can access shared memory without making any system calls, making it harder to track changes to the shared memory and fake them later.

One simple solution for handling memory-mapped files is to make copies of pages that are returned upon the first page fault to a memory region mapped to a file. During replay of requests to create memory maps, the memory areas are mapped to dummy files, and page faults are handled by returning saved copies of pages. Due to the lazy demand-paging approach used by Linux, only those pages that are accessed during execution need to be copied, thus drastically reducing the overhead. This approach will not work when the same region of the file is mapped as a shared region by multiple processes, each of which make changes to the region. This approach works for files that have been mapped as private, as well as shared mappings where all changes to the file are made by the process being debugged.

Handling shared file-mappings with multiple processes writing to the file is a more complicated problem, and requires the kernel to force a page fault for every access to the shared region by the process being replayed instead of just the first access as in the earlier case. A possible enhancement to the logging solution would be to set the access rights of a given page to the last process to access it, and thus only fault when another process has accessed the page since this one. This way, several successive reads or updates will only suffer one costly exception instead of many. During replay, however, it would still be required to fault for each access since the other processes might not be around any more to make their changes.

In Flashback, currently, we have implemented the simple solution described earlier. In spite of the enhancement proposed for shared file-maps with multiple writers, we believe that an efficient solution to address this challenge will require support from the underlying architecture. Shared memory can be dealt with using similar mechanisms.

5.4 Multithreaded applications

While the techniques outlined above work for applications with a single thread of control, replaying multithreaded applications poses additional challenges. Logging changes made by a multithreaded application involves logging the changes of each thread of the debugged process. During replay, the interleaving of shared memory accesses and events has to be consistent with the original sequence.

Ensuring that the multiple threads are scheduled in the same relative order during replay is another issue. For multi-threaded applications running on a single processor system, we propose adopting the approach described in [13] for deterministic replay. The basic idea is to record information about the scheduling of the threads during the original execution and use this information during replay to force the same interleaving between thread executions. Since this implementation would also be in the kernel, the physical thread schedule is transparent and can be used in lieu of the logical thread schedule information proposed by [13]. We will implement this in the future in the tool, possibly with the support of architecture-level mechanisms such as those described in [55].

5.5 Signals

Signals are used to notify a process about a specific event, or to force the process to execute a special handling code when an event is detected during its execution. Signals may be sent to a process either by another process or by the kernel itself. Signals are asynchronous and are delivered proactively to a process by the kernel. They may be delivered at any time to a process. Signals present a challenge for deterministic replay because signals are asynchronous events that affect the execution of a process. The replaying mechanism has to ensure that signals are delivered at exactly the same points during re-execution as in the original execution.

Deterministic reproduction of signals may be handled using the approach proposed by Slye and Elnozahy [66], though Flashback does not currently support signal replay. The mechanism outlined in their work makes use of an instruction counter to record the time between asynchronous events. The instruction counter is included in most modern processor systems today. When a signal occurs, the system creates a log entry for it, which includes the value of the instruction counter since the last system call invocation. During replay, Flashback checks to see if the next log entry corresponds to a signal. If so, then it initializes the instruction counter with the time from the current system call till the signal. When a trap is generated because of timeout, the kernel delivers the signal to the process.

5.6 Implementation in Linux

We have implemented a prototype of Flashback's replay mechanism in Linux-2.4.22. The prototype handles replay of system calls as well as memory-mapped files to a limited extent. In Linux, a user-space process invokes a system call by loading the system call number into the `eax` register and optional arguments in other registers, and then raising a programmed exception with vector 128. The handler for this exception, the system call handler, does several checks and then runs the function indexed in the `sys_call_table` array by the *system call number*. It finally returns the results got from this action to the user process.

We used *syscalltrack* [71], an open-source tool that allows system calls to be intercepted for various purposes such as logging and blocking. The core of the tool has been implemented in a kernel module which "hijacks" the system call table by replacing the default handlers for some system calls with special functions. System call invocations can be filtered based on several criteria such as the process id of the invoking process as well as values for specific arguments. System calls that need to be logged are handled in a number of ways. At one extreme, the special function may log the invocation of the system call and let the call go through to the original handler, while at the other it may block the system call invocation and return a failure to the user process. The actual behavior of the special function is controlled using rules that may be loaded into the kernel.

In our implementation, we added a new action type that the special function can perform, namely the `AT_REPLAY` action for replaying. This action verifies that the system call invocation matches a call that the process originally made, then sets the return value according to the logged invocation and also makes the same side effects on the arguments as before. By doing this, it bypasses the actual system call handler for some system calls and overrides its behavior with that of the simulating function. For other system calls such as the `brk` call, Flashback allows the system calls to be handled by the original system call handler since memory allocations need to be made even during replay.

6 Evaluation

We evaluate our prototype implementation of Flashback using microbenchmarks as well as real applications. The timing data we present were obtained on a 1.8GHz Pentium IV machine with 512KB of L2 cache and 512MB of RAM.

6.1 Overhead of State–Capture

To perform a very basic performance evaluation of the rollback capabilities, we instrumented the `checkpoint()`, `discard()` and `replay()` system calls. We then ran a small program that repeatedly invokes `checkpoint()`, does some simple updates and then either discards the checkpoint by calling `discard()` or rolling back by calling `replay()`.

Figure 4(a) presents the time for the three basic operations: checkpoint, discard and replay. A checkpoint takes around 25-1600μs as the amount of state updates between two consecutive checkpoints varied from 4KB to 400MB. Since creation of a shadow process involves creation of a copy-on-write map, the cost is proportional to the size of the memory occupied by the process. Similarly, the cost to discard or replay a shadow is proportional to the size of memory modified by the process.

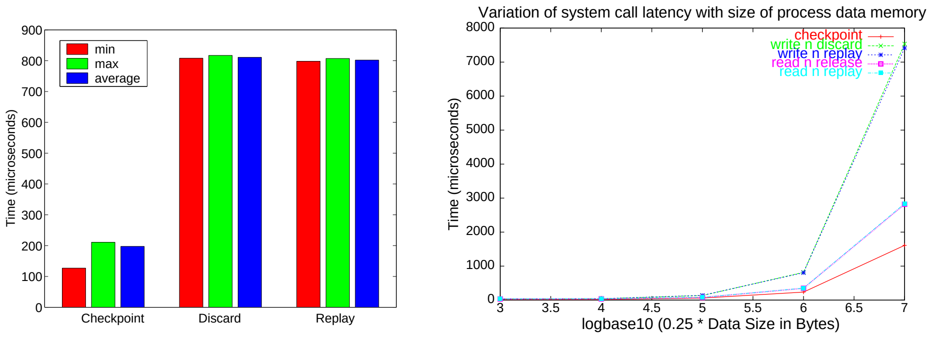
The costs of discard (replay) are also directly proportional to the number of pages in the corresponding checkpointed state (the current state). This is because both discard and replay involve deletion of one copy-on-write map. Our results show that discard and replay take around 28-2800μs when the entire memory is read, and between 28-7500μs when the entire data memory is written. The higher costs in the latter case are because the kernel has to return a large number of page frames to its free memory list when the shadow state is dropped/reinstated. Typical applications will of course not modify all pages in their address space between checkpoints, and so the costs of the discard and replay operations will be closer to the lower end of the range shown in Figure 4(b).

An important objective of our rollback infrastructure is to have minimal impact on normal application performance. We therefore consider the data for `checkpoint()` and `discard()` more important than that for `replay()`. This is because the latter is invoked only when errors occur, and will therefore not be part of common-case behavior. Regardless, the overhead imposed by the rollback call is as low as that for shadow state release. This is promising since it indicates we can restore execution state as fast as common case checkpoint discard.

6.2 Overhead due to Logging

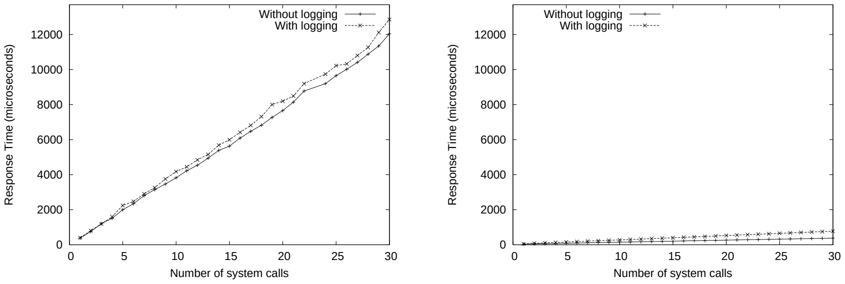
In order to evaluate the logging overhead, we wrote a simple test program that employs two threads in order to isolate the impact of the logging overhead. In the program, the parent thread forks and creates a child. It then loads the rules for logging into the framework and notifies the child to begin invoking system calls. The rules allow the kernel to filter system call invocations based on the process ID of the child.

While logging system calls that have side effects on memory regions, such as `read`, `stat` and `getsockopt`, Flashback also needs to record the contents of the buffer



(a) Data Size 40 MB Completely over-written (b) Data Sizes ranging from 4KB to 400MB

Figure 4: Microbenchmark Results for Shadow Process Creation at different sizes of process data memory



(a) Buffer-copying (BC) (b) No-copying (NC)

Figure 5: Response time overhead (microseconds) for varying number of system call invocations

or structure. Thus, with regard to logging overhead, there are two groups of system calls, those that cause side effects on some memory regions, and those that simply return a value after performing the intended action. We refer to the first class of system calls as buffer-copying (BC) and the second group as no-copying (NC). For NC system calls, there is no need to record the contents of buffers; just the system call ID and the return values will suffice.

To study the overhead on every system call due to hijacking and logging, we invoked the read and write system calls several times, gradually increasing the number of invocations. In each invocation, the number of bytes read or written is 4 KB. For each run, we start with a clean file cache in order to make the effect of caching on system call overheads consistent. Figures 5 shows the overhead imposed by the sandbox mechanism. The overhead due to sandboxing occurs because of the extra indirection of system calls imposed by Flashback. Instead of being handled directly by the system call handlers, system call requests need to pass through filters and the logging mechanism. The increase in overhead is linear with the number of system calls for both the system calls. The difference in slope between the two lines on the graphs represents the extra per-system-call overhead imposed due to logging. This is around 30 microseconds on an average.

To evaluate the effect that the copying of buffers has on the logging overheads, we invoked the read and write system calls repeatedly, gradually increasing the number of bytes read or written from 4 KB to 2 MB. The actual number of system calls is small in this case. Figure 6 shows the overhead while varying the amount of data read or written. The overhead for BC and NC system calls is comparable, and the extra copying of buffers does not appear to impose any extra overhead. This is because the contents of the log are buffered, and written to disk asynchronously. In these experiments, the disk cache was warmed since all the data for the files was prefetched before the actual execution. The values therefore reflect reads and writes entirely involving the cache only.

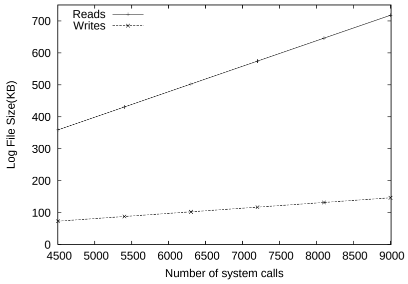
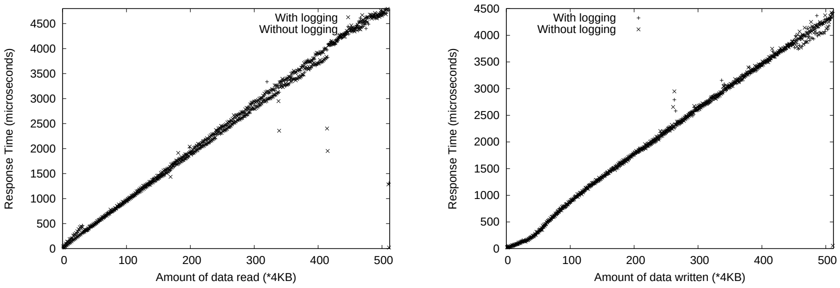


Figure 7: Size of the log file (KB) for varying number of system call invocations

Figure 7 shows the space overhead because of logging



(a) Buffer-copying (BC) (b) No-copying (NC)

Figure 6: Response time overhead (microseconds) for varying sizes of memory logging

BC and NC calls. As expected, the growth in the size of the log file is linear in terms of the number of system calls, though the slope is greater for BC since more data is written each time.

6.3 Application Results

In order to test our implementation of state-capture in a realistic environment, we measure the performance with the well-known *Apache* web-server. We evaluate the system overhead for both multiprocess version and multithread version of Apache. Our evaluation serves to demonstrate two things: first, that fine-grained rollback support is possible, and can be applied to real applications; and second, that the performance impact on common-case execution is minimal.

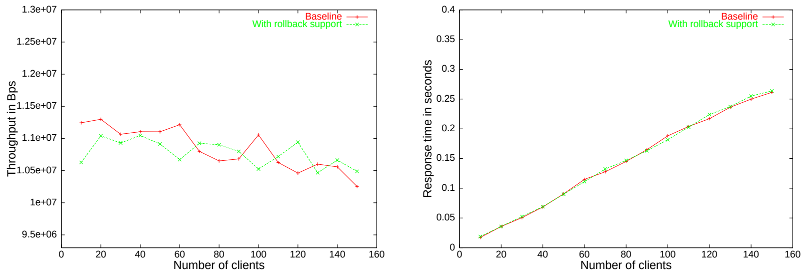
In all the experiments reported herein, the web server is bottle-necked by the network and is serving data at full network throughput of 100Mbps. We use these experiments to show that off-the-shelf machines(1.8MHz, 512MB RAM) have enough spare cpu cycles to provide fine-grained rollback without affecting client's perceived performance. The server is checkpointed multiple times (typically thrice) during the processing of each request. We essentially create a checkpoint just before reading the HTTP request off a newly accepted socket, before processing a valid HTTP request from an existing connection and before writing out the HTTP response onto the socket. Thus, at any point of time, Flashback maintains as many shadow images as the total number of requests being processed by the server. All data points in this section have been averaged over three runs.

The Apache server can be configured to run in a multiprocess or multithread mode. In the former, Apache maintains a pool of worker processes to service requests. Each worker process is a single thread and the number of workers in the pool is adapted dynamically based on load estimates. However, in the latter, Apache uses a much smaller pool of worker processes, with each worker process consisting of multiple threads implemented by the *pthread* package. We present here performance figures for both configurations of the Apache server. In this experiment, the web server checkpoints its state upon the arrival of request for a page, processes the request, and discards the checkpoint. These results reflect the overhead of capturing state. Since Flashback currently does not support replay of multithreaded execution and shared memory, we disabled logging for replay during these experiments.

To exercise the web server, we use an `http` request-generating client application, *WebStone* [73], which sends back-to-back requests to a single web server. Each request constitutes a fetch of a single file, randomly selected from a pre-defined "working set". The working set comprised files of sizes varying between 5KB and 5MB, but the majority of requests constituted a fetch of 5KB. The request generating application forks a pre-defined number of client processes, each of which submits a series of random requests to the web server. The server was run on a off-the-shelf 1.8GHz Pentium IV machine, connected to the client via a 100Mbps LAN. Performance was measured in terms of throughput, aggregate response time and load on the server CPU. In all the experiments reported here, the server was operating at the full network throughput of 100Mbps.

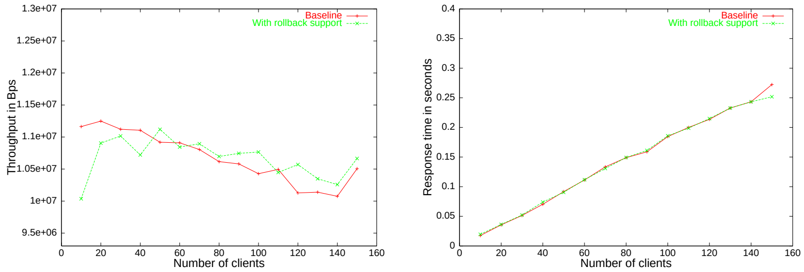
We compare the Apache web-server on the prototype system with a baseline system running the original version of Linux. Figure 8 shows throughput and response time in Flashback and the baseline system with Apache running in multiprocess mode. It is clear from the graphs that there is no significant difference between the client-perceived throughput and response time. When the number of clients is small, Flashback has 10% lower throughput, even though the average response time is the same as the baseline system. However, when the number of clients increases, the difference between baseline and Flashback disappears. In some cases, Flashback performs even better than the baseline system. We consider these small differences well within expected experimental variance, and conclude that the impact of rollback support on Apache performance is negligible.

Figure 9 shows the results for the multithread version of Apache. As expected, the overheads imposed by Flashback on multithreaded execution are slightly lower than those for the multiprocess version, evidenced by the throughput figures which more closely match one another in most cases. This lower overhead is a direct result of fewer effective system calls, because when one



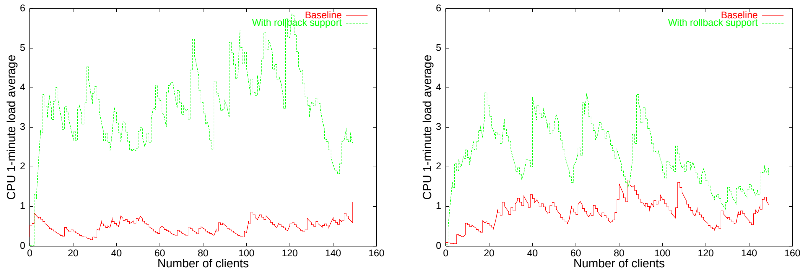
(a) Throughput (b) Response time

Figure 8: Throughput and response time with Multiprocess Apache web-server. *Baseline* corresponds to the case running in the default Linux system without rollback support, and *With rollback support* corresponds to the Linux kernel modified to include rollback support. The results shown in these figures indicate that throughput and response time are not affected by Flashback. These times reflect state-capture overhead



(a) Throughput (b) Response time

Figure 9: Throughput and response time with multithread Apache web-server. The results shown in these figures indicate that throughput and response time are not affected by Flashback. These times reflect state-capture overhead



(a) Multiprocess (b) Multithread

Figure 10: One-minute CPU load averages for the host on which the Apache web-server is running. The curves demonstrate the extra work being performed by the kernel when checkpointing is enabled.

thread undergoes a capture event, the state of all the other threads is automatically captured. Subsequent capture-events on the other threads of this process are treated as *nops* during the lifetime of this shadow process. Hence the number of capture events necessary are much fewer.

Although client-perceived system performance remains almost unaffected, the kernel does perform extra work each time a checkpoint is initiated. Of course, this does not come for free. To quantify the cost, we monitored CPU load average on the machine hosting the web-server. The metric we use measures the average number of processes waiting on the run queue over the last minute, which is an estimate of system load as it statistically captures the amount of time each process spends on the run queue. Figure 10 plots these results for the multiprocess and multithread versions. The graphs expose the overhead in capturing shadow state, which in our evaluation occurs very frequently (once every request received by the server). Note that even though the cpu utilization of the server increases by 2-4 times, the client perceived performance, in both data bytes delivered and time to respond, remains unchanged. We assert that the experimental setup is realistic as modern web-servers are often constrained by network bandwidth and have spare cpu cycles.

In both the multiprocess and multithread configurations, CPU load increases significantly. In the singlethreaded case, the extra load is quite high. This is because a multiprocess Apache webserver uses a collection of separate Unix processes to handle web requests, each of which now captures shadow state when handling a request. In the multithreaded version, the state-capture event occurs once for all threads of execution, because we capture the state of all threads, *en masse*, each time a checkpoint is taken. The smaller number of system calls, and the smaller size of the state captured (per worker thread), together contribute to the multithread configuration exhibiting better CPU load than the multiprocess configuration.

7 Using Flashback in gdb

Using Flashback, it is fairly straightforward to incorporate support for checkpointing, rollback and deterministic replay into a debugging utility such as *gdb*.

We have modified gdb to support three new commands – checkpoint, rollback, and discard, for creating checkpoints, to support rollback and deterministic replay of debugged programs. Programmers can set up breakpoints at places where they might want to create checkpoints. At these breakpoints, after seeing the state of the program, they can choose to create a new checkpoint by using the checkpoint command. They can also discard earlier checkpoints, thereby freeing system resources associated with those checkpoints by using the discard command. If they find the system state to be inconsistent, they can roll back to an earlier checkpoint by using the rollback command.

Using Flashback, gdb can be made to automatically take periodic checkpoints of the state of the process being executing. New commands are added into the debugger user interface to allow programmers to enable or disable automatic checkpointing during execution of the debugged program. Programmers also have control over the frequency of checkpointing. This frees the programmer from having to insert breakpoints at appropriate locations in the code and explicitly taking checkpoints.

In order to incorporate checkpoints into gdb, we made changes to the *target system handling component* and the *user interface* components. The target system handling component handles the basic operations dealing with the actual execution control of the program, stack frame analysis and physical target manipulation. This component handles software breakpoint requests by replacing a program execution with a trap. During execution, the trap causes an exception which gives control to gdb. The user can choose to take a checkpoint at this time. gdb does this by making a `checkpoint` system call passing the process ID of the process being debugged. Similarly, for rollback and replay, gdb uses the `rollback` and `replay` system calls respectively.

For automatic checkpointing, in addition to these changes, gdb maintains a timer that keeps track of time since the last checkpoint. The timeout for the timer can be set by the user. When a timeout occurs, gdb checkpoints the process.

8 Conclusions and Future Work

In this paper we presented a lightweight OS extension called Flashback to support fine-grained rollback and deterministic replay for the purpose of software debugging. Flashback uses shadow process to efficiently capture in-memory states of a process at different execution points. To support deterministic reply, Flashback logs all interactions of the debugged program with the execution environment. Results from our prototype implementation on real systems show that our approach has small overheads and can roll back programs quickly.

Besides software debugging, our system can also be used to improve software availability by progressively rolling back and re-executing to avoid transient errors [78]. In addition, our approach can be extended to provide lightweight transaction models that require only atomicity but not persistence.

We are in the process of combining Flashback with hardware architecture support for rollback and deterministic replay [56] to further reduce overhead. We are also evaluating Flashback with more applications. Flashback currently only works for programs that run on a single machine. We are exploring ways to extend it to support distributed client-server applications by combining with techniques surveyed by Elnozahy et al. [18].

Flashback including the patches to both Linux and gdb will be released to the open source community so that other researchers/developers can take advantage of *Flashback* in interactive debugging.

9 Acknowledgments

We would like to thank Dr Srinivasan Seshan, the shepard for the paper, for useful suggestions and comments. We also thank the anonymous reviewers for useful feedback, and the Opera group for useful discussions, and Jagadeesan Sundaresan, Pierre Salverda and Arijit Ghosh for their contribution to the project.

REFERENCES

[1] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer. Detecting Data Races on Weak Memory Systems. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 234–243, 1991.

[2] Alvisi and Marzullo. Trade-offs in implementing causal message logging protocols. In *PODC: 15th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 1996.

[3] C. Amza, A. Cox, and W. Zwaenepoel. Data replication strategies for fault tolerance and availability on commodity clusters. Proc. of the International Conference on Dependable Systems and Networks., 2000.

[4] A. Bobbio and M. Sereno. Fine grained software rejuvenation models. In *IEEE International Computer Performance and Dependability Symposium*, 1998.

[5] A. Borg, J. Baumbach, and S. Glazer. A message system supporting fault tolerance. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles (SOSP)*, volume 17, pages 90–99, 1983.

[6] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault tolerance under UNIX. *ACM Transactions on Computer Systems*, 7(1):1–24, Feb. 1989.

[7] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.

[8] G. Candea and A. Fox. Crashonly software. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, May 2003.

[9] M. Castro and B. Liskov. Proactive recovery in a byzantine-fault-tolerant system. In *OSDI*, 2000.

[10] P. M. Chen, D. E. Lowell, and G. W. Dunlap. Discount checking: Transparent, low-overhead recovery for general applications. Technical report, University of Michigan, Department of Electrical Engineering and Computer Science, July 1998.

[11] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell. The Rio file cache: Surviving operating systems crashes. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 74–83, Cambridge, Massachusetts, 1–5 Oct. 1996. ACM Press.

[12] Y. Chen, J. S. Plank, and K. Li. Clip: a checkpointing tool for message-passing parallel programs. In *Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–11. ACM Press, 1997.

[13] J. Choi and H. Srinivasan. Deterministic replay of java multithreaded applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 48–59, Aug. 1998.

[14] J.-D. Choi and S. L. Min. Race Frontier: Reproducing Data Races in Parallel-Program Debugging. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 145–154, 1991.

[15] K. D. Cooper, M. W. Hall, R. T. Hood, K. Kennedy, K. S. McKinley, J. M. Mellor-Crummey, L. Torczon, and S. K. Warren. The ParaScope Parallel Programming Environment. *Proceedings of the IEEE*, 81(2):244–263, 1993.

[16] O. P. Damani and V. K. Garg. How to recover efficiently and asynchronously when optimism fails. In *International Conference on Distributed Computing Systems*, pages 108–115, 1996.

[17] G. W. Dunlap, S. T. Kind, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. *ACM SIGOPS Operating Systems Review*, 35(SI):211–224, 2002.

[18] E. N. M. Elnozahy, L. Alvisi, Y. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408, 2002.

[19] D. R. Engler, D. Y. Chen, and A. Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *Symposium on Operating Systems Principles*, pages 57–72, 2001.

[20] D. Evans, J. Guttag, J. Horning, and Y. M. Tan. Lclint: A tool for using specifications to check code. In *Symposium on the Foundations of Software Engineering*, December 1994.

[21] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, /2002.

[22] S. Feldman and C. Brown. Igor: A system for program debugging via reversible execution. *ACM SIGPLAN Notices, Workshop on Parallel and Distributed Debugging*, 24(1):112–123, Jan. 1989.

[23] C. Flanagan and S. N. Freund. Type-based race detection for Java. *ACM SIGPLAN Notices*, 35(5):219–232, 2000.

[24] C. Flanagan, K. Leino, M. Lillibridge, C. Nelson, J. Saxe, and R. Stata. Extended static checking for java. In *PLDI*, 2002.

[25] G. Candea et. al. Reducing recovery time in a small recursively restartable system. In *DSN*, 2002.

[26] K. Gharachorloo and P. B. gibbons. Detecting Violations of Sequential Consistency. In *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 316–326, 1991.

[27] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Proceeding of the ACM SIGPLAN 2002 Conference on Programming language design and implementation (PLDI)*, 2002.

[28] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proc. 2002 Int. Conf. Software Engineering*, pages 291–301, Orlando, FL, May 2002.

[29] R. Haskin, Y. Malachi, and G. Chan. Recovery management in quicksilver. *ACM Transactions on Computer Systems (TOCS)*, 6(1):82–108, 1988.

[30] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *the Winter USENIX*, 1992.

[31] K. Havelund and T. Pressburger. Model checking java programs using java pathfinder, 1998.

[32] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton. Software rejuvenation: analysis, module and applications. In *FTCS-25*, 1995.

[33] Y. Huang and Y. Wang. Why optimistic message logging has not been used in telecommunication systems. In *Proceedings of the 1995 International Symposium on Fault-Tolerant Computing (FTCS)*, pages 459–463, june 1995.

[34] D. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pages 171–181, Aug. 1988.

[35] KAI-Intel Corporation. Assure. URL: http: //developer.intel.com/software/products/assure/.

[36] S. Kumar and K. Li. Using model checking to debug network interface firmware. In *the Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[37] K. Li, J. Naughton, and J. Plank. Concurrent real-time checkpoint for parallel programs. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 79–88, Seattle, Washington, Mar. 1990.

[38] K. Li, J. Naughton, and J. Plank. An efficient checkpointing method for multicomputers with wormhole routing. *International Journal of Parallel Programming*, 20(3):159–180, June 1991.

[39] K. Li, J. Naughton, and J. Plank. Low-latency concurrent checkpoint for parallel programs. *IEEE Transactions on Parallel and Distributed Computing*, 1994.

[40] B. Liskov. Distributed programming in argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[41] A. Loginov, S. H. Yong, S. Horwitz, and T. W. Reps. Debugging via run-time type checking. In *Fundamental Approaches to Software Engineering*, pages 217–232, 2001.

[42] D. E. Lowell, S. Chandra, and P. M. Chen. Exploring failure transparency and limits of generic recovery. In *OSDI*, 2000.

[43] D. E. Lowell and P. M. Chen. Free transactions with Rio Vista. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)*, volume 31,5 of *Operating Systems Review*, pages 92–101, New York, Oct.5–8 1997. ACM Press.

[44] M. Luján, J. R. Gurd, T. L. Freeman, and J. Miguel. Elimination of Java array bounds checks in the presence of indirection. In *Proceedings of the Joint ACM Java Grande-Iscope Conference*, pages 76–85, 2002.

[45] E. Marcus and H. Stern. Blueprints for high availablity. John Willey and Sons, 2000.

[46] J. M. Mellor-Crummey and M. L. Scott. Synchronization without contention. In *Proceedings of The 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 269–278, Apr. 1991.

[47] S. L. Min and J.-D. Choi. An Efficient Cache-based Access Anomaly Detection Scheme. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 235–244, 1991.

[48] National Institute of Standards and Technlogy (NIST), Department of Commerce. Software errors cost u.s. economy $59.5 billion annually. NIST News Release 2002-10, 2002.

[49] G. C. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. In *Symposium on Principles of Programming Languages*, pages 128–139, 2002.

[50] R. H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *PADD*, 1993.

[51] J. Oplinger and M. S. Lam. Enhancing software reliability with speculative threads, October 2002.

[52] D. A. Patterson and et. al. Recovery-oriented computing (roc): Motivation, definition, techniques, and case studies. UC Berkeley CS Tech. Report,UCB//CSD-02-1175, 2002.

[53] D. Perkovic and P. J. Keleher. A Protocol-Centric Approach to on-the-Fly Race Detection. *IEEE Transactions on Parallel and Distributed Systems*, 11(10):1058–1072, 2000.

[54] J. S. Plank, K. Li, and M. A. Puening. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–??, 1998.

[55] M. Prvulovic and J. Torrellas. Reenact: using thread-level speculation mechanisms to debug data races in multithreaded codes. In *Proceedings of the 30th Annual Symposium on Computer Architecture*, 2003.

[56] M. Prvulovic and J. Torrellas. ReEnact: Using Thread-Level Speculation to Debug Software; An Application to Data Races in Multithreaded Codes. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, June 2003.

[57] R. Rodrigues, M. Castro, and B. Liskov. BASE: Using abstraction to improve fault tolerance. In *Proceedings of the 18th ACM Symposium on Operating System Principles*, pages 15–28, Banff, Canada, Oct. 2001.

[58] M. Ronsse and K. D. Bosschere. RecPlay: a Fully Integrated Practical Record/Replay System. *ACM Transactions on Computer Systems*, 17(2):133–152, 1999.

[59] M. Russinovich and B. Cogswell. Replay for concurrent non-deterministic shared-memory applications. In *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, pages 258–266, Jerusalem, Israel, 1996. ACM Press.

[60] Y. Saito and B. Bershad. A transactional memory service in an extensible operating system. In *USENIX Annual Technical Conference*, 1998.

[61] K. Salem and H. Garcia-Molina. Checkpointing memory-resident databases. Technical Report CS-TR-126-87, Department of Computer Science, Princeton University, 1987.

[62] M. Satyanarayanan, H. Mashburn, P. Kumar, D. Steere, and J. Kistler. Lightweight recoverable virtual memory. In *SOSP*, 1993.

[63] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[64] E. Schonberg. On-the-fly detection of access anomalies. In *ACM SIGPLAN '89 Conference on Programming Language Design and Implementation (PLDI)*, June 1989.

[65] M. Seltzer, Y. Endo, and C. Small. Dealing with disaster: Surviving misbehaved kernel extensions. In *OSDI*, 1996.

[66] J. H. Slye and E. N. Elnozahy. Supporting nondeterministic execution in fault-tolerant systems. In *Proceedings of the Twenty-Sixth International Symposium on Fault-Tolerant Computing*, pages 250–261, Washington, June25–27 1996. IEEE.

[67] S. W. Smith, D. B. Johnson, and J. D. Tygar. Completely asynchronous optimistic recovery with minimal rollbacks. In *FTCS-25: 25th International Symposium on Fault Tolerant Computing Digest of Papers*, pages 361–371, Pasadena, California, 1995.

[68] N. Sterling. Warlock: A static data race analysis tool. In *USENIX Winter Technical Conference*, 1993.

[69] J. M. Stone. Debugging concurrent processes: A case study. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 1988.

[70] R. E. Strom and S. A. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, Aug. 1985.

[71] syscalltrack software home page at http://syscalltrack.sourceforge.net/how.html.

[72] C. A. Thekkath and H. M. Levy. Hardware and software support for efficient exception handling. In *ASPLOS*, 1994.

[73] G. Trent and M. Sake. Webstone: The first generation in http server benchmarking. Feb 1995.

[74] C. v. Praun and T. Gross. Object race detection. In *16th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Tampa Bay, FL, October 2001.

[75] D. Wagner and D. Dean. Intrusion detection via static analysis. In *IEEE Symposium on Security and Privacy*, pages 156–169, 2001.

[76] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, San Diego, CA, February 2000.

[77] Y. Wang, P. Y. Chung, Y. Huang, and E. N. Elnozahy. Integrating checkpointing with transaction processing. In *FTCS*, 1997.

[78] Y. Wang, Y. Huang, W. K. Fuchs, C. Kintala, and G. Suri. Progressive retry for software failure recovery in message-passing applications. *IEEE Transactions on Computers*, 46(10):1137–1141, Oct 1997.

[79] Y. Wang, Y. Huang, K.-P. Vo, P.-Y. Chung, and C. Kintala. Checkpointing and its applications. In *FTCS-25*, 1995.

[80] M. Wu and W. Zwaenepoel. eNVy: A non-volatile, main memory storage system. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 86–97, San Jose, California, Oct. 4–7, 1994. ACM SIGARCH, SIGOPS, SIGPLAN, and the IEEE Computer Society.

[81] Y. Xie and D. Engler. Using redundancies to find errors. In *Proceedings of the tenth ACM SIGSOFT symposium on Foundations of software engineering*, pages 51–60, 2002.

[82] Y. Zhou, P. M. Chen, and K. Li. Fast cluster failover using virtual memory-mapped communication. In *the 13th ACM International Conference on Supercomputing*, June 1999.