Millisecond Rollbacks to Recover from Software Failures

Sorav Bansal, Piyus Kedia
IIT Delhi

ABSTRACT

We present a method to recover from failures caused by software bugs. Our method relies on two key observations: first, bugs in a running software system are likely to be either rare and non-deterministic, or triggered by an unfriendly external input; second, the failure symptoms are highly likely to be observed within a short instruction-count window of the occurrence of the input or event triggering the bug. We refer to the input or non-deterministic choice which triggered the bug as the root-cause event, and the occurrence of the failure symptom, the failure. Our work assumes that the probability of failure due to an event decreases rapidly with the number of CPU instructions executed since the event.

We maintain two states of the running system: current and committed. The committed state remains behind the current state by a specified rollback window, measured as the number of CPU branches executed. If a failure happens on the current state, we rollback the system to the committed state in the hope of undoing the root-cause event. The restarted execution attempts to avoid the re-occurrence of the rollback events.

We use Virtual Machine Record/Replay to implement this rollback mechanism efficiently. Our method requires no system-specific knowledge. Our technique improves the reliability of a computer system manifold, and incurs less than 5% runtime overhead on all our benchmarks except one, where the overhead is around 20%. Rollbacks take between 0.2 seconds and a few seconds, depending on the size of the virtual machine, and always keep the system in correct state. Even multiple failures/rollbacks keep the system available.

1. INTRODUCTION

Software bugs are inevitable. Bugs are discovered even in software which has gone through tens of years of testing and verification, and used by millions of users[4]. The occurrence of a failure in deployed software is expensive, and best avoided. Critical services increasingly depend on software, but guaranteeing correctness of software remains an elusive goal. While recovering from hardware failures is relatively well understood ([2], [32], [21]), software failures usually lead to irrecoverable situations. Moreover, failures due to software bugs are much more common than hardware failures. Previous software fault tolerance approaches[5, 30] have provided techniques to recover from special types of failures in certain types of software systems. We present a general method to recover from software failures.

Our work relies on two premises about typical software bugs:

a. Software failures in deployed software are either caused by non-deterministic events (such as unfortunate interrupt timing), or triggered by external inputs (such as a malicious network packet). We assume that bugs of other types, which are either deterministic or input independent, should be easy to discover through testing.

b. The failure (or it’s symptom) is typically observed within a “short” instruction-count window of the occurrence of the input or non-deterministic choice that triggered the software bug (the root-cause event). This is largely true. Most software is written to operate on input as soon as possible (using interrupt-based mechanisms, for example). Even with systems with buffering and/or polling-based mechanisms, the instruction-count distance between the event and the failure is typically less than a few tens of millions of executed instructions. This is also true for interrupt-timing related bugs. Intuitively, the probability of failure due to an event decreases rapidly with the time elapsed since the occurrence of that event.

We maintain two states of the running system: current and committed. The committed state remains behind the current state by a specified instruction-count window (which we call the rollback window). If a failure occurs on the current state, the system is rolled back to the committed state and restarted. In doing so, we undo any event (interrupt or external input) which occurred within the last rollback window. For failures caused due to unfortunate timing of interrupts, the restarted execution from the committed state attempts to avoid repeating the same interrupt timing. We show that it is possible to do this at low execution overheads (<20% on all our benchmarks), and with fast recovery times (rollbacks take between 0.2 seconds to a few seconds to complete). We call this method “millisecond rollbacks” (as we typically rollback by a few tens of milliseconds of execution), or millirollbacks for short.

If the state of the system is made visible to external entities (over output devices such as the network or the console) before it is committed, a rollback can cause external inconsistencies. We discuss the nature of such inconsistencies and propose variants of millirollbacks to avoid them.

We use Virtual Machine Monitor’s Record/Replay (VM R/R) capability[36] to implement millirollbacks. For each VM, two processes, a record process and a replay process, are spawned maintaining the “current” and “committed” states respectively. Each process runs a copy of the same VM. The VM running inside the record process receives real external input, and generates an execution trace using VM R/R. The execution trace is streamed to the replay process using shared memory. The replay process recreates the state of the VM (as it must have existed in the record process) using the execution trace. The replay process runs simultaneously with the record process, but always remains behind the record process’s VM state by the specified rollback window. At any point in it’s execution, the record process can choose to rollback it’s VM to the state of the VM in the replay process.

We identify failures based on minimal knowledge about error routines inside the guest OS. For example, we consider a process abort (e.g., due to a segmentation fault or an assertion failure) inside a Linux guest as an indication of failure and attempt to recover from it. We can track failure either
on all processes or on a set of specific processes, depending on the system requirement. Failure can also be determined using periodic hello messages from an external client. This is the only system-specific knowledge required by our method. The choice of failure symptoms could be different for different systems, and it is trivial to incorporate any failure model with our method. We attempt a small number of rollbacks (e.g., three in our experiments) to try and recover from a failure before giving up and continuing just like the system would have done without millirollbacks. Rollbacks are implemented by snapshotting the state of the replay process and loading it in the record process. The snapshot includes the state of the CPU, memory, and all the virtual devices. Because we rollback the state of all I/O devices along with CPU and memory, we avoid any inconsistency between I/O device and processor state. Rollbacks do not adversely affect other system or network activities. We provide detailed experiments to measure this.

For timing-related bugs (e.g., concurrency bugs in multi-threaded software), we avoid the re-occurrence of rolled-back interrupt-timing events in restarted executions. We assume that any of these events could have been the root cause of failure. We maintain this list of suspect events cumulatively that any of these events could have been the root cause of failure before giving up and continuing just like the system would have done without millirollbacks. Rollbacks are implemented by snapshotting the state of the replay process and loading it in the record process. The snapshot includes the state of the CPU, memory, and all the virtual devices. Because we rollback the state of all I/O devices along with CPU and memory, we avoid any inconsistency between I/O device and processor state. Rollbacks do not adversely affect other system or network activities. We provide detailed experiments to measure this.

We have implemented a prototype system to test our ideas. There is one caveat to our current prototype implementation. Even though our ideas apply generally, our prototype only supports uniprocessor VMs because efficiently recording the execution trace of a multiprocessor VM is significantly harder[8, 31].

Because we spawn two identical processes per VM, our method can potentially use twice more CPU and memory than a normal execution. Current hardware trends towards increased number of CPU cores and increased memory capacities make this an acceptable tradeoff for better reliability.

The paper is organized as follows. In Section 2, we briefly discuss VM R/R and its performance. We discuss our millirollbacks approach and our design choices in Section 3. We discuss our experiments in Section 4. Section 5 discusses related work, and Section 6 concludes.

2. VIRTUAL MACHINE RECORD/REPLAY

With the rapid growth of cloud computing, virtual machines (VMs) and virtual machine monitors (VMMs) are becoming ubiquitous. One of the capabilities of a VMM is its ability to produce an execution trace (recording) of the VM, which can be replayed to produce a faithful reconstruction of the guest OS state at any point in its execution. ReTrace[36] demonstrated this capability in VMware Workstation and reported as low as 5% runtime overhead, and 0.5 byte per thousand instructions log growth rate. VM R/R works by recording all external input to virtualized devices and the timing of interrupts as they are delivered to the guest. Time is counted by counting the number of instructions (or branches) executed by the guest. On x86 platforms, the three tuple \( (n\text{branches, rip, rcx}) \) uniquely identifies a logical guest execution epoch, where \( n\text{branches} \) is the number of branches executed by the guest, \( \text{rip} \) is the guest's current instruction pointer, and \( \text{rcx} \) is the current value of count register (needed for instructions with \text{rep} pre-fix). The \( n\text{branches} \) counter is maintained using hardware performance counters. All deterministic instructions (i.e., instructions producing identical output on same input irrespective of time of execution) can be executed unmodified directly on hardware. All non-deterministic instructions must be made to trap to the VMM and their non-deterministic result recorded (e.g., \text{rdtsc} on x86). Because a huge fraction of executed instructions in common workloads are deterministic (e.g., >99% for most compute-intensive workloads), the overhead of VM R/R is minimal. All interrupts delivered to the guest are recorded along with their epoch \( (n\text{branches, rip, rcx}) \). Emulated devices are recorded by recording all the non-determinism in the device emulation code. For example, if the device uses an external input (e.g., network packets), that input is logged.

During replay, the results of non-deterministic instructions and non-deterministic device inputs are obtained from the log. Replaying interrupts requires special care. We require the guest to yield control to the VMM at the interrupt epoch \( (n\text{branches, rip, rcx}) \) for accurate delivery of the replayed interrupt. On x86 architectures, this can be achieved by configuring the performance counters to overflow at the desired branch count \( n\text{branches} \), which generates an interrupt causing the guest to yield control to the VMM. We then single-step the guest till we reach the desired \( (\text{rip, rcx}) \) before injecting the replayed interrupt to the guest. On current x86 architectures, this interrupt-on-overflow mechanism for performance counters is imprecise. It is possible for the branch count to overshoot the desired value by up to 128 before an interrupt is generated. The solution to this problem (as also noted in [8]) is to configure the performance counters to generate an interrupt at \( (n\text{branches} - 128) \) and then single-step the guest till the interrupt injection epoch \( (n\text{branches, rip, rcx}) \). This careful singlestepping near interrupt injections causes extra runtime overhead during replay (compared to record).

We implemented VM R/R inside Linux/KVM[26]. We evaluate our implementation using the benchmarks listed in Table 1. The benchmarks have been chosen to stress different components of the virtual machine monitor. Some of the benchmarks have been inspired by a previous VMM performance study[22]. The table also presents the typical log size growth rate of each benchmark. The two log size growth columns are for different ways of recording the disk device, namely output replay (records and replays the output of the disk device) and full replay (emulates the disk device fully). More details on output-replayed disks and full-replayed disks are discussed in Section 3.3. Figure 1 shows the performance characteristics of our VM R/R implementation on KVM. We show results compared to KVM as baseline. We deliberately don’t show results compared to native execution, because performance difference between KVM and native execution is either negligible (for compute-intensive workloads) or is heavily dependent on the chosen virtual devices (for I/O-intensive workloads). We simply use the default KVM/Qemu virtual devices. The full details on our experiments are provided in Section 4.

The \text{emptyloop} benchmark is compute-intensive and represents all compute-intensive workloads executing at user privilege level; \text{gpid} exercises the system call handling mechanism in Linux; \text{forkbomb} exercises the process creation and destruction methods (including page table manipulations); \text{cp} exercises the disk; \text{inet} and \text{onet} stress the network; and
sleep emulates an idle system. iscp exercises both network and disk, while lincompile combines CPU, memory, and disk usage. We measure performance using the host’s wall clock time. Figure 1 shows the performance of VM Record (kvm-record), VM Replay (kvm-replay) and three variants of VM Millirollbacks (kvm-mr, kvm-mrod, and kvm-mrdm) on these benchmarks. We discuss R/R performance in this section and discuss millirollback variants and their performance in following sections. We use full-replayed disks for our R/R experiments. Each runtime is divided into the percentage of time spent in the guest, in the host kernel, and other activities including I/O and idle time. The overhead of recording is within 20% for all our benchmarks. The performance of VM Replay can be worse due to single-stepping. The overhead of VM R/R on compute-intensive applications (emptyloop and gpid) is almost zero. The overhead is more for I/O intensive applications such as cp and iscp. forkbomb has higher R/R overhead than emptyloop due to paging activity and swap disk usage. The performance of sleep on kvm-replay is faster than on kvm because executions of the x86 halt instruction by the guest’s idle thread finish instantaneously on kvm-replay, while they wait for an external interrupt on kvm and other variants. Compute-intensive applications spend almost all their time in the guest, while I/O intensive applications spend a significant fraction of time in the host user-level code for device emulation.

3. MILLIROLLBACKS

We implemented millirollbacks on KVM/Qemu using two processes, one recording the execution trace of the guest OS, and the other replaying it. The execution trace is streamed from the recording process to the replaying process using a circular shared memory buffer. This scheme (which we call kvm-mr in Figure 1) has similar performance to kvm-record. For millirollbacks, we also show the percentage of time spent in reading, writing, or waiting for the execution log. We run on a multiprocessor machine and use a sufficiently large buffer of 20MB for the shared memory buffer, to prevent situations where the record process has to wait for the replay process. The performance overhead of millirollbacks is minimal for almost all applications. More explanation and the full details of our experimental setup are provided in Section 4.

We discuss the design choices we made during the implementation of our prototype system.

3.1 Opportunistic vs. Fixed Rollbacks

On failure, we rollback the execution to the state of the replay process. The replay process must remain behind the record process by at least the rollback “time” window, which is a system parameter. It is possible for the replay process to lag behind by more than the specified rollback window. On failure, we could choose to either rollback to the current state of the replay process (irrespective of how far behind it is), or we could choose to wait for the replay process to catch up before rolling back by exactly the rollback window. We refer to the former scheme as “opportunistic rollbacks”, and the latter scheme as “fixed rollbacks”.

Opportunistic rollbacks allow more non-deterministic choices (or external inputs) to be undone even at small rollback window sizes. Also, opportunistic rollbacks are quick. The primary disadvantage of opportunistic rollbacks is that the rollback length could be much greater than the desired rollback window. As we discuss later, this could be a problem if external entities (such as networked machines) have observed the current state of the system (through network packets, for example) and could get confused if they suddenly have to deal with a much older rolled-back state.

Fixed rollbacks wait for the replay process to reach the start of the rollback window before transferring state to the record process. They provide an upper-bound on the rollback duration. Fixed rollbacks can take longer than opportunistic rollbacks if the replay process has drifted too far from the record process. In our experiments in Section 4, we examine the nature and magnitude of this drift. We also compare the performance of fixed and opportunistic rollbacks.

A rollback involves snapshotting the state of the replay process and loading it in the record process. In our prototype implementation, the typical time to do an opportunistic rollback on a VM with output-replayed disks is around 0.2 seconds. We currently implement rollbacks by snapshotting the replay process’s VM state on disk and loading it in the record process. We implemented disk-based snapshotting for it’s relative simplicity. We believe that it is possible to further improve this rollback time through in-memory snapshotting.

3.2 Rollback Window Size

A large rollback window can result in longer rollback times and unexpected timing behaviour for networked machines. A small rollback window can result in repeat failures. There are many ways to measure the progress made by a process (“time”). Some options we considered were host wall-clock...
time, guest wall-clock time, and guest’s log entry count. We use the guest’s \texttt{nbranches} (number of branches executed by guest) to measure time. \texttt{nbranches} is the most direct measure of the amount of code executed between the root-cause event and the failure, and hence has the strongest correlation with the probability of failure. It is independent of the amount of time taken by each instruction and the system’s I/O or log activity.

To decide on an appropriate rollback window, we measured the typical number of events that occur in a certain number of branches executed by the processor. We found that this varies with the application. Table 2 gives the number of different events that occur per 1000 branches for each type of workload. For compute-intensive workloads (\texttt{emptyloop} and \texttt{gpid}), the time taken to execute 1000 branches is around 1 microsecond. For disk-intensive workloads (\texttt{scp} and \texttt{cp}), this time could be as high as 732 microseconds. For a workload exhibiting a lot of OS paging activity (\texttt{forkbomb}), the time taken to execute 1000 branches is around 9 microseconds. For an idle system (\texttt{sleep}), the time taken is around 15 milliseconds for a 1000 branches. Similarly, the number of external inputs (disk I/Os, network packets, etc.) per thousand branches is usually small with the maximum being around 0.1 event per thousand branches (\texttt{scp}).

We then performed experiments to determine the appropriate rollback window. We wrote a small user-level program with a buffer overflow vulnerability that received keyboard input. If the keyboard input is too long, our program would abort with a segmentation fault, triggering a rollback. Our millirollback scheme recovers from the fault by undoing the last (possibly more than one) keyboard input (in this case, the \texttt{Enter} key). Inside our program, we inserted a time delay $D$ between the receipt of the keyboard input and it’s use. We measured the minimum rollback window required to recover from the fault for each $D$. During the delay, the system was kept busy through a compute-intensive task to simulate an overloaded system with buffering delays. Table 3 presents our results. We tried rollback windows from 1000 branches to 50,000 branches to identify the smallest rollback window which recovered from the bug. We repeated the experiment multiple times. For fixed rollbacks, 2000 branches were enough to rollback a delay of up to 5ms. But the fixed rollback scheme failed to rollback delays of 10ms or longer. This can be explained by the 10ms scheduling time slice used by our guest OS scheduler. If $D$ is 10ms or longer, the guest must rollback at least one timer interrupt and associated interrupt handling routines. This seems to require a longer rollback window than what was used in our experiments.

Opportunistic rollbacks can provide longer rollbacks even at short rollback windows, because it is possible for the replay process to drift by more than the rollback window. As we study later in our experiments, the replay process could drift behind the record process by millions of branches for a busy system. For idle systems, this drift usually remains close to the rollback window. In a way, opportunistic rollbacks have the effect of an adaptive rollback window, where the rollback duration is more for busy systems and less for idle systems. For our experiment using opportunistic rollbacks, a rollback window of only 5000 branches was enough to rollback a delay of 100ms with a high probability, both for busy and idle systems.

Table 2 suggests longer rollback windows than what we required in our experiment of Table 3. We explain this difference by noting that while there can be a large \texttt{nbranches} gap between successive inputs, any input is likely to be processed almost immediately (within a few tens of thou-
Table 2: The number of events that occur per 1000 branches for each workload

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Time (us)</th>
<th>Interrupts</th>
<th>Disk I/Os</th>
<th>Packets sent</th>
<th>Packets Received</th>
</tr>
</thead>
<tbody>
<tr>
<td>emptyloop</td>
<td>1.5</td>
<td>0.0001</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>gpid</td>
<td>0.95</td>
<td>0.0001</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>forkbomb</td>
<td>8.72</td>
<td>0.0005</td>
<td>0.0001</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>cp</td>
<td>203.4</td>
<td>0.03</td>
<td>0.013</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sleep</td>
<td>15018</td>
<td>1.5</td>
<td>0.003</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>iscp</td>
<td>732</td>
<td>0.06</td>
<td>0.002</td>
<td>0.02</td>
<td>0.1</td>
</tr>
</tbody>
</table>

Table 3: Size of rollback window (nbranches) required to rollback D milliseconds of computation.

<table>
<thead>
<tr>
<th>D</th>
<th>1ms</th>
<th>2ms</th>
<th>3ms</th>
<th>4ms</th>
<th>5ms</th>
<th>10ms</th>
<th>100ms</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fixed</td>
<td>1000</td>
<td>2000</td>
<td>2000</td>
<td>2000</td>
<td>2000</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

3.3 Disk: Output or Full Replay

There are two ways of recording and replaying virtual storage devices:

1. Output Replay: In this scheme, only the interactions between the virtual CPU and the virtual disk device are recorded and replayed. There are two types of interactions between the CPU and the disk:
   
   - **Disk Block Reads**: For output-replayed disks, the disk blocks read by the record process are stored in the execution log. When the replay log reads those disk blocks, the data is returned from the log.
   
   - **Disk Block Writes**: For output-replayed disks, R/R implementations execute disk writes during recording and ignore them while replaying. For rollbacks, this is insufficient. To effect a rollback, we need to rollback the disk state along with the state of the CPU and memory. If we simply update the disk during recording, rolling it back will be impossible. Our record process does not update the disk directly but rather buffers the contents of the disk update in the execution log. The replay process executes (commits) the write to the disk when it reaches that point in the execution log. For a rollback, all disk writes which are buffered in the execution log and not yet committed to disk are simply discarded. In this scheme, disk reads by the record process need special care. The current copy of a disk block could now either reside in the streamed execution log or the actual disk depending on its commit status. If the record process requests a block whose current copy is in the execution log, the contents are served from the log. This was simple to implement using the shared-memory queue which we use for streaming the log. Figure 2 illustrates this scheme.

2. Full Replay: In this scheme, two copies of the disk device exist, just like two copies of CPU and memory state exist. Both record and replay processes operate on their private disk. Assuming both disks started from the same state and all disk operations were deterministic, they must always contain identical contents at identical program points. Rollbacks can be effected by simply copying the virtual disk of the replay process to the record process. This scheme reduces the logging overhead, as disk reads and writes do not need to be buffered. On the other hand, this scheme makes rollbacks prohibitively expensive; a rollback on a 100G disk could take up to several minutes.

We implement both schemes in our prototype implementation. The full-replay and output-replay schemes are labeled kvm-mr and kvm-mrod (kvm-mr-output-disk) respectively in Figure 1. For cp, a disk-bound microbenchmark, the performance of kvm-mr is 20% slower than kvm. This happens because two processes try to write to two different virtual disk images which are co-located on the same physical disk. We believe that this overhead can be reduced by using a separate physical disk for each image.

Both cp and lincompile perform significantly worse with kvm-mrod. On both these benchmarks, the record process spends a significant fraction of time waiting on the log. This shows that these benchmarks cause the shared memory buffer to fill up. We used a 20MB shared memory circular buffer, and we copied a 100MB file for cp. The amount of disk data read for lincompile was also much more than 20MB. As we study later in our experiments, increasing the size of the shared memory queue dramatically improves the performance of these benchmarks on kvm-mrod. We deliberately don’t use a large shared memory queue for this experiment, as we want to study the performance slowdown when the size of the files being read is larger than our shared memory buffer.

Rollbacks on full-replayed disks can be made faster by using thinly-provisioned virtual disks (such as those supported by the qcow2 virtual disk format[25]). These virtual disk formats allow a disk to be represented as a read-only backing file with a read-write “diff” file which contains the differences between the actual disk’s state and the backing file. The diff file remains short if the amount of data written to the virtual disk is small, but could grow for long-running VMs. It is possible to devise schemes to periodically and
asyncronously merge the diff file with the backing file, to keep the diff file short, and rollbacks quick. We have not implemented these periodic asynchronous merges between the diff file and the backing file. Given our rollback times of 0.2 seconds on VMs with output-replayed disks, we crudely and conservatively estimate rollback times on VMs with full-replayed disks at around a few seconds. This estimate is based on the assumption that using asynchronous periodic merges, the diff file can be maintained at less than 1GB with small runtime overheads.

### 3.4 Network: Immediate or Delayed sends

Rollbacks in the middle of an ongoing network conversation can be dangerous. Packets sent over the wire cannot be rolled back. Other networked entities may depend on the truth of the packets sent by our system.

There are two ways of dealing with network packet sends:

- **Immediate Sends**: In this scheme, the recording process sends network packets immediately to the remote hosts without waiting for the replay process to catch up. The replay process just ignores that packet that had to be sent.

- **Delayed Sends**: In this scheme, the recording process does not actually send the packets. Instead, it buffers the packets in the execution log till the replay process reaches that execution point and actually transmits the packet. This scheme ensures that our system state is externalized (i.e., made visible to external entities) only after it is reached by the replay process (in other words, only after it is “committed”). Delayed network packet sends can increase the perceived network latency and decrease the perceived network bandwidth.

Packet receipts in the replay process can only occur through the log. The replay process never receives any “real” packets. This approach of delaying network packet sends is labeled `kvm-mrnd` (kvm-mr-network-delay) in Figure 1. For `iscp`, the throughput reduces by around 2.8x. The slowdown depends on our choice of the rollback window size. The rollback window size for this experiment in Figure 1 was 5000 branches. The slowdown for a rollback window of 100 branches is 2.1x. For `net`, `kvm-mrnd` is actually faster than plain `kvm`. We found that this happens because we are sending packets in 4-byte chunks and `kvm-mrnd` happens to buffer and batch network sends. We do not observe this speedup if we send data in larger chunks. We measure these tradeoffs in more detail in our experiments in Section 4.

Delayed network packet sends are not always necessary. For many network protocols and applications, rollbacks after externalization of state may be acceptable. For example, rollbacks on a system running a web service after it has sent some “uncommitted” packets to its clients may only just result in a broken web session. On the other hand, there are situations where rollbacks after sending uncommitted packets are unacceptable. Some examples are connections to backend services like file servers, databases, or directory servers. An update to a file system may be non-idempotent and a rollback could confuse the network file system protocol semantics. It is possible to selectively delay network packet sends on a system based on the protocol and/or the destination host.

### 4. EXPERIMENTS

We implemented KVM R/R and KVM millirollbacks inside Linux kernel 2.6.36.4. We tested our implementation on Gentoo and Ubuntu hosts. Our millirollbacks implementation uses x86 hardware performance counters to count branches, and the monitor trap flag (MTF) for single-stepping. We implemented record/replay functionality for emulated devices inside Qemu. Our implementation is complete enough to run a full Linux guest. We tested our implementation with all types of applications including graphical and networked applications. For example, we could successfully view a Youtube video while running millirollbacks underneath. We tested the stability of our implementation by running it continuously for over 24 hours with an active guest.

We used Qemu 0.13.0 with default configuration for device emulation record/replay. Our 32-bit Gentoo Linux guest was configured with fully-emulated e1000 network and IDE disk devices. The guest ran with 128MB physical memory, 512MB swap space, 8GB disk space and other devices emulated by Qemu by default. For our experiments, we assumed that any process abort indicates failure. Process aborts include assertion failures and segmentation faults. We detected failures by tracing the guest kernel instruction address which executes on a process abort. We ran our experiments on a machine with 12 Intel Xeon X5650 2.67 GHz processor cores, 24GB memory, and 300GB disk.

We first evaluate the performance of the different variants of millirollbacks. Figure 1 presents our results. We plot the host wall-clock time (normalized to `kvm`) on the Y-axis. We performed ten trials for each reading and plotted the minimum. The variance in running times was negligible for all benchmarks except `cp`. A rollback window of 5000 branches, and a shared memory queue of 20MB was used for these experiments. The performance of `kvm-mr` is close to unmodified `kvm` except `cp` which has 20% overhead. We observed a relatively large variance in the performance of `cp`. This was because we used the space-optimized `qcow2` virtual disk format which performs adaptive deduplication and can thus result in widely varying performance. `kvm-mrod` can have significant overhead (up to 60%) for disk-intensive workloads (`cp`, `lincompile`). A majority of this extra time is spent waiting on the log. This shows that these benchmarks fill up the shared memory buffer and then wait for the replay process to catch up. We later study the effect of shared memory buffer size on the performance of disk-intensive benchmarks running on output-replayed disks. Except `cp` and `lincompile`, all other benchmarks spend negligible time waiting for the log, which confirms that our shared memory buffer is large enough to absorb any fluctuations in the rate of generation of log traffic.

Next, we evaluate the functionality of millirollbacks. We wrote a minimal ping server and introduced a buffer overflow vulnerability inside it. The buffer overflow gets triggered by a malicious network input (received over the `socket()` interface), causing the process to abort with a segmentation fault. A ping client on the host machine was made to send malicious packets to our vulnerable ping server inside the guest VM. We used a rollback window of 5000 branches with opportunistic rollbacks and output-replayed disks. Our rollback window with opportunistic rollbacks ensured that the ping server never crashed in more than 150 trials that we performed. Every malicious packet that was received was
discarded during a rollback. We varied the frequency of malicious packets to trigger different number of rollbacks. Each fault in the ping server would trigger a rollback of the entire VM, affecting other applications in the guest. We ran a workload simultaneously with our vulnerable ping server inside the Linux guest and evaluated the effect of rollbacks on the performance on that workload.

Figure 3 plots our results for emptyloop, sleep, and cp. The X-axis is the number of rollbacks that occurred during the execution of that workload and the Y-axis is the completion time of that workload. The emptyloop program took almost 2.3x longer (from 300 seconds to 700 seconds) to complete when 12 rollbacks occurred during it’s execution. Each rollback finished within 0.2 seconds, so the primary reason behind the slowdown was the length of the rollback distance. On emptyloop, the replay process is slower than the record process (recall Figure 1) causing the drift between the two processes to be much more than the rollback window. On each opportunistic rollback, the computation performed in the rollback distance was discarded, and subsequently repeated causing the slowdown. We also performed this experiment on emptyloop with fixed rollbacks and nbranches=5000 to see if that reduced the completion time. Figure 4 plots our results. Our emptyloop program slows down by around 2.5x with fixed rollbacks, which is slightly more than what was observed with opportunistic rollbacks. Fixed rollbacks wait for the replay process to catch up till the rollback window before resuming. Because the replay process is slower than the record process, fixed rollbacks take even longer than opportunistic rollbacks.

We next perform the same experiment on cp, a disk-intensive workload. For cp, we copied a 500MB file within the Linux guest. The completion times of cp are almost independent of the number of failures. Because cp generates relatively large log traffic, it’s rollback distance is upper-bounded by the size of the shared memory buffer. This ensures that rollbacks do not cause much computation to be wasted, and the cp benchmark exhibits a higher variance in running times than the slowdown due to rollbacks.

We also performed this experiment for iscp where a 200MB file was copied over the network to the local disk. Using immediate network sends, the iscp session stalled on a rollback. Analysis of the network traffic trace revealed that a rollback caused the TCP sequence numbers to be rolled back, which confused the remote host which had previously observed “uncommitted” packets with higher sequence numbers. This confirms that rollbacks with immediate network sends can cause active TCP connections to break.

We repeated the iscp experiment with delayed network sends (kvm-mrнд). In this case, a rollback causes the packets buffered in the shared memory queue to be discarded. When this happens, the transport layer (TCP) recovers from it, just like it would recover from any other type of packet loss. We sent 14 malicious ping packets to our server (triggering 14 rollbacks) at regular intervals during an active iscp session lasting 830 seconds, and we found that the throughput of iscp remained unaffected. Just like cp, the amount of computation wasted on a rollback for iscp is insignificant compared to it’s total running time.

Table 4 gives statistics on the number of branches rolled back during opportunistic rollbacks for emptyloop, sleep, and cp. These statistics were obtained after recording at least 40 rollbacks for each benchmark. For idle systems (sleep), the replay process is always on the heels of the record process, resulting in small rollbacks. Compute-intensive processes (emptyloop) can result in the largest drift between

![Figure 2: Mechanism of execution log streaming. For output-replayed disks, the actual disk writes are done by the replay process. Similarly, for delayed network packets, the packets are buffered by the record process and sent by the replay process. If the record process requests a disk block which has not yet been committed, the data is served from the shared-memory queue.](image)

![Figure 4: Performance of emptyloop with the number of rollbacks, using fixed rollbacks.](image)
Firstly, the replay process is slower than the record process and has a tendency to drift. Secondly, compute-intensive processes generate very few execution log entries, which ensures that the shared memory queue never fills up to make the record process wait for the replay process. The drift in applications which produce a large amount of execution log entries (like cp on output-replayed disks) is upper-bounded by the size of the shared memory queue. We study the distribution of drift over time later in this section.

### Timing Bugs

Next, we evaluate millirollbacks on timing-related bugs. We ran a multi-threaded program with harmful race conditions. On failure, our system rolled back the last (possibly more than one) interrupt. Additionally, on each rollback, we identified and recorded the program points where the timer interrupt was delivered during the rollback duration. In the restarted execution, we prevented the timer interrupt from being delivered at these program points and a small region (64 bytes in our experiment) around those program points. This method relies on the assumption that the delivery of the timer interrupt at one of these program points could have been the root cause of the failure. We call these program points, suspects. This list of suspect program points is maintained and avoided cumulatively. Effectively, this mechanism infers regions of code that should execute atomically on a uniprocessor system. It is possible to have false positives in the suspect list, i.e., program points that have nothing to do with the failure could get classified as suspects. In some cases, this may affect application performance. Assuming rollbacks are rare, the number and size of false positives should be typically small.

Using this mechanism, our system was automatically able to infer (unprotected) atomic code regions. We discuss an experiment on a small multithreaded program with race conditions. Figure 5 presents the pseudo-code of our program. This program has multiple threads and two shared variables debit and credit. Harmful race conditions exist on both variables, and this program almost certainly fails when executed natively.

```c
debit = 0;
credit = total;
for (t = 0; t < max_threads; t++) {
    thread_create(transfer);
}

transfer():
    for (i = 0; i < 1000; i++) {
        debit--;
        credit++;
        assert(debit + credit == total);
    }
```

With millirollbacks, the program rolls back on every assert failure (process abort), identifies the probable interruptions that caused the failure, and avoids delivering the interrupts at the same points during restarted execution. In our experiments, this program always succeeded after 1-2 rollbacks. Once the list of suspect program points covers the unprotected critical sections in the code, the program runs without failure after that.

The number of rollbacks required to converge to a correct execution depends on the static code length of the unprotected critical section. Our example in Figure 5 has a small unprotected critical section and recovers after 1-2 rollbacks. We next perform experiments with larger unprotected critical sections and study the number of rollbacks required to converge to a correct execution. To increase the length of the unprotected critical section, we unroll the loop in the debit-credit example by adding multiple copies of the increment and decrement operations within the same loop iteration. The first plot in Figure 6 shows the number of rollbacks required to converge to a correct execution for different lengths of the critical section. The growth is almost linear with the size of the critical section. The second plot in Figure 6 plots the size of the blacklisted region with the length of the unprotected critical section. The size of the blacklisted region closely follows the size of the unprotected critical section.

We note here that the effectiveness of this approach is dependent on the nature and placement of the programmer’s `assert()` statements. This technique works best if the as-

### Table 4: Distribution of the number of branches rolled back while each benchmark was running (using opportunistic rollbacks).

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Min</th>
<th>Max</th>
<th>Median</th>
</tr>
</thead>
<tbody>
<tr>
<td>sleep</td>
<td>13k</td>
<td>145m</td>
<td>13k</td>
</tr>
<tr>
<td>emptyloop</td>
<td>31k</td>
<td>45b</td>
<td>21b</td>
</tr>
<tr>
<td>cp</td>
<td>12m</td>
<td>96m</td>
<td>24m</td>
</tr>
</tbody>
</table>

### Figure 3: Performance of emptyloop, sleep and cp with the number of rollbacks, using opportunistic rollbacks.
Figure 6: The first plot shows the number of rollbacks required to converge to a correct execution for the debit-credit example. The second plot shows the size of the blacklisted region (region where delivery of interrupts is avoided) marked by our bug-avoidance algorithm. The x-axis is the size of the critical section in bytes which is increased by unrolling the transfer() loop. The typical average length of an x86 instruction is 3-5 bytes.

assertion checks are close in time (or nbranches) to the source of bug. If the assertion checks are too far from the bug, we are likely to observe failures during restarted executions too.

Scheduling

The scheduling of the record and replay processes has an impact on the performance of the system, especially during rollbacks. The drift between the replay process and the record process is lower-bounded by the size of the rollback window, and upper-bounded by the size of the shared memory buffer. A small shared-memory buffer limits the magnitude of drift by making the record process wait for the replay process when the buffer is full.

Different workloads exhibit different drifts between the record and replay processes. To measure this, we sampled the nbranches distance between the record and replay processes at periodic intervals separated by at least 50 milliseconds. We also sampled the size of the shared memory queue. Figure 7 shows our results for emptyloop, sleep, cp, and iscp. This experiment was performed with output-replayed disks (kvm-mrod). The maximum size of the shared memory buffer in this experiment was 20MB.

Because emptyloop keeps the CPU 100% busy, and replay performance is poorer than record performance, the drift keeps increasing with time. When the record process is done executing the loop (and the replay process is still unfinished), the drift starts to decrease. The buffer size shows a similar trend over time. For sleep, the drift is typically equal to the rollback window (5000 in these experiments) and the buffer is near empty. For both cp and iscp, the buffer is almost always full, and provides an upper bound on the amount of drift.

We also performed these experiments on a uniprocessor machine where the record and replay processes are time-multiplexed over the same physical processor. We plot the drift trends for emptyloop and sleep in Figure 8. The drift distribution for emptyloop on a uniprocessor is similar to that on a multiprocessor. The drift for sleep on a uniprocessor is different, and shuttles between the lower bound of 5000 (rollback window) and the upper bound of 100,000. The drift upperbound on a uniprocessor is determined by the OS scheduler’s time-slice, while on a multiprocessor, the drift remains small because both processes execute simultaneously. In general, we found that the OS scheduling can have a deep impact on the performance of the system. For example, the latency of delayed network sends (kvm-mrdn) is directly affected by the drift. Higher drifts imply longer network latencies and poorer network throughput for the guest.

Figure 8: Distribution of drift across time for emptyloop and sleep on a uniprocessor machine.

Figure 9 plots the ping latency between the host and the guest with increasing rollback window size on kvm-mrdn. A ping client was run inside the guest, and a ping server was run on the host. The out-latency measures the time from the guest to the host, and the in-latency measures the time from the host to the guest. Our ping client sends a packet at intervals of 1 second, thus keeping the guest mostly idle. Therefore, our ping client behaves much like our sleep benchmark. Figure 9 shows that the ping latency increases by around 12ms for every increase of 1000 branches in the roll-
Figure 7: Distribution of drift across time for emptyloop, sleep, cp, and iscp on a multiprocessor machine.

Figure 10: Completion time for iscp with increasing rollback window size on kvm-mrd

back window. This fits well with our data in Table 2, which shows that sleep takes 15ms to execute 1000 branches.

Figure 10 plots the network throughput for iscp with increasing rollback window size. The completion time increases with increasing window size, although the rate of increase is not as large as that seen with our results on ping latency. This is because the iscp benchmark executes more branches per unit time than our ping client, causing the increase in window size to have a less pronounced effect on the network throughput.

For delayed network sends, scheduling can have an impact on the network performance of the guest. During some of our trial runs in our experiments on kvm-mrd, we observed the network throughputs to be much lower than the average value. This happened because the stock Linux kernel scheduler used in our experiments occasionally made unfortunate scheduling choices causing the drift between replay and record processes to increase, and causing network throughput to decrease. For a real system, it is important to tune the host’s scheduler to make drift-aware decisions to provide better guest performance.

We next study the impact of the size of the shared memory buffer on workloads which generate high log traffic. In particular, we look at cp on kvm-mrod, which generates log traffic at the rate of 5 MBps. We copied a 100MB file inside the guest using cp. Figure 11 shows the completion time of this program on kvm-mrod with varying size of the shared memory circular buffer. The first bar (with label 0) represents the performance of unmodified kvm. The extra copies in the shared buffer causes around 60-80% overhead (as also reported in Figure 1). However, as our shared memory buffer becomes larger than 100MB, the performance of cp becomes better than even plain kvm. We explain this interesting result by observing that in this case, the record process finishes execution after it has dumped all the disk writes to the shared memory buffer, without waiting for it to be committed to disk. The replay process asynchronously commits this data to disk after the record process has finished executing cp. This also shows that in this case, the replay process must have drifted behind the record process by a large number of branches. This tradeoff between performance and rollback distance (drift) in our scheme is similar to the classic tradeoff between performance and durability of writes on secondary storage[18]. Just like an upper bound on the size of dirty buffer cache is necessary to provide reasonable crash behaviour for secondary storage, an upper bound on the rollback distance is necessary to provide a reasonable behaviour after failure/rollback. One way to upper-bound the drift is to limit the size of the shared memory buffer.

Figure 11: Completion time of cp copying a 100MB file with increasing size of the shared memory queue.

Finally, we note that kvm-mr (full-replayed disks and immediate sends) with opportunistic rollbacks has the best performance characteristics. As discussed in Section 3.3, rollbacks in this mode can take between 0.2 seconds to a few seconds depending on the size of the disk file. While rollbacks on kvm-mr can cause active network connections to break, this is usually still better than a crash. Moreover, in many settings, it is possible to selectively delay network packets based on protocol/destination host without degrading network performance. For these reasons, we believe that kvm-mr is the most practical mode of operation.

5. RELATED WORK

Our method improves the reliability of software in presence of rare (heisen) bugs. Baker[1] observed that availability can be improved by emphasizing fast recovery over crash prevention. Our work has similar philosophy. There
have been other similar recovery-oriented approaches to improving software reliability. Micro-reboots[5] is a technique to restart only subsets of fine-grain components on failure, instead of rebooting the whole system. The granularity of components is typically finer than the process level (e.g., EJBs in Java enterprise systems). Unlike micro-reboots, our method requires almost no information about the individual components of a software system.

Shadow Drivers[30] is another recovery-oriented approach for device drivers inside the operating system kernel. Drivers are isolated within lightweight protection domains such that the driver can be restarted on failure without affecting the rest of the kernel. The shadow drivers approach can recover from all types of device driver failures, assuming a shadow driver has been written for it.

Both micro-reboots and shadow drivers assume specific knowledge about the software component being protected and sometimes require deep instrumentation on a per-component basis. A good analysis and instrumentation can allow these techniques to recover from many types of errors in these components. In contrast, our approach can only recover from input-related or timing-related errors. The advantage of our approach is that we require no instrumentation or understanding of individual software components. We only monitor typical failure symptoms, and can thus protect the entire operating system and its applications at once. For other types of bugs, which are neither input-related nor timing-related, we rely on software testing and verification techniques to discover them.

Another work, similar in spirit to ours, is acceptability-oriented computing[27], where the concept of program correctness is replaced with a set of acceptability properties that must hold for the execution of the program to remain acceptable. One example of acceptability-oriented computing is continued execution in presence of memory-errors and corruption of data structures[28]. Another example is ClearView[23] which monitors an application for errors[11], identifies correlated invariants, and patches the application at runtime to avoid future failures. Our work follows a similar goal of continued execution in presence of errors by attempting to identify correlations between the bug and the failure at runtime.

There is an interesting parallel between millirollbacks and previous work on external synchrony for file systems[18]. Both approaches externalize output only after the corresponding state is “committed”. While the external synchrony technique resolves the tension between performance and durability on magnetic disk devices, millirollbacks is a way of guarding against software failure in a small time window.

Crash-prevention through systematic testing[3, 4, 15] or better debugging tools[6] are orthogonal approaches to ours with the common goal of improving software reliability. Time-travel debugging using record/replay is also a powerful technique. A nice property of our millirollbacks proposal is that we obtain the execution trace of the guest, which can be later used for time-travel debugging in case of failure. Both testing and debugging are pre-deployment techniques to try and expose the bug. Millirollbacks is a post-deployment technique to try and hide the bug.

Our technique relies on VM R/R and we have developed our prototype using a uniprocessor VM R/R implementation. Execution logging for multiprocessors is more expensive and harder, due to race conditions on shared physical memory. Both hardware[7, 10, 13, 14, 17, 34, 35] and software-only approaches[8, 12, 16, 19, 31] have been proposed to address this limitation. DoublePlay[31] improves the performance of multiprocessor record/replay by executing two copies of each virtual processor: one in speculative mode, and the other in correct mode. On noticing an inconsistency due to a race condition on shared memory, the speculative execution is rolled back. Advances in multiprocessor VM R/R will help make millirollbacks more performant on multiprocessor systems. Our technique requires high-fidelity replays, and R/R techniques which relax this requirement (e.g.,[20]) do not seem suited to our method.

Another approach to dealing non-deterministic bugs is to avoid them by limiting the non-deterministic choices to well-known safe values[33]. In a limited sense, our millirollbacks approach achieves a similar effect for timing-related errors automatically.

Transactions are a popular programming construct which are based on recovery from erroneous schedules. There has been a large body of recent work on transactional memory[9] in software[24] and hardware[29]. Instead of using locks or other pessimistic concurrency control mechanisms, the programmer can specify atomic regions in his code and the runtime system ensures that the corresponding code executes atomically. Transactions require the capability to rollback a computation. Our method is not a concurrency-control mechanism. However, our results on recovery from non-deterministic timing-related bugs show that it may be practical to implement transactions using millirollbacks.

Hardware fault tolerance is relatively better understood than software fault tolerance. VMWare High Availability (VMware HA[32]) is a commercial offering which uses VM R/R to run multiple copies of the same VM simultaneously on different physical machines. The execution log generated on the primary machine (master) is streamed to the secondary machines which simultaneously replay this log. In the event of a hardware failure on the primary machine, the secondary machine takes over. The low overhead of execution logging and the fast speed of failovers provides a seamless user experience even in the face of multiple hardware failures. Our scheme uses similar techniques, but we provide resilience against software failures, not hardware failures. In our scheme, both record and replay processes run on the same physical machine and communicate through shared memory. We assume that the failure-triggering event and the failure typically occur close in time, and by maintaining a safe distance between the secondary and primary VMs, we are able to rollback to a correct state in the event of failure. The small rollback time keeps the system much more available than reboot-based schemes.

It is possible to implement millirollbacks by running record and replay processes on different physical machines. However, output-replayed disks (kvm-mrodm) and delayed network sends (kvm-mrodn) are easier to implement through shared memory and will be more expensive to implement over message-passing interfaces.

6. CONCLUSIONS

We demonstrate the practicality and utility of millisecond rollbacks to improve software reliability. Our method recovers from input-related and timing-related bugs through whole-system rollbacks. The performance overhead of our
scheme is minimal for most settings. We present a detailed analysis of the performance and functionality of our prototype implementation based on Virtual Machine Record/Replay capability.

References