Research in virtual machines

Peter M. Chen, Dominic Lucchetti, Ashlesha Joshi

February 14, 2007

1 Introduction

This paper gives a rough sketch of some research ideas we're working on at the University of Michigan related to virtual machines.

2 Virtual-machine replay

We and others have built systems to log the execution of a virtual machine and to use the log to re-execute intervals of execution at a later time. VM replay can recover the state of the virtual machine at instruction-level granularity. It is especially helpful when you want to know something about the past execution or state of the VM, but didn't anticipate needing to know this particular item beforehand.

VM replay is possible on uniprocessor and multiprocessors VMs (i.e., multiple processor dedicated to a single guest), though capturing fine-grained, frequent interactions between processors (e.g., due to shared memory) can be very expensive. See George Dunlap's Ph.D. dissertation for details.

The ability to replay virtual machines at low overhead (time and space) has proven very useful in a variety of applications, such as debugging operating systems and applications, computer forensics, and intrusion detection.

We are also exploring how to use VM replay in a variety of other contexts.

2.1 Recovery for databases, file systems, and other applications

Recovering from system crashes, aborted transactions, and user errors is an important feature in all databases and file systems. It is also used in other applications that store persistent state, such as editors with auto-save. Currently, databases, file systems, and other applications keep custom logs for this purpose. These systems can take advantage of VM replay and potentially eliminate their own custom logs. Since VM replay can restore the system back to any state, a database or other application may no longer need to store its own log and rather recover using the VM replay log. For example, storing the VM replay log may obviate the need to store file system backups, since the VM replay log allows the system to recover any prior state.

2.2 Searching the past execution

A useful capability in general is to search the past execution of a system. VM replay can provide a general way of retrieving past information. We maintain indices to quickly find historical points of interest in a system's execution. We can also maintain abstracts or revision logs of important information. If a user or administrator wants to find information that we did not anticipate being needed, we can replay the VM's execution with a monitoring predicate attached. The monitoring predicate uses VM introspection to extract the information of interest from the replaying VM. E.g., the predicate may construct an extra index, or save a series of screenshots.

3 Virtual-machine introspection

Virtual-machine introspection is examining or controlling the state or events in a virtual machine from outside the virtual machine. We use the term "predicate" to refer to an event-handler that runs when an event of interest occurs in the virtual machine. A predicate can also run continuously, e.g., to monitor the executing (or replaying) virtual machine.

We have used VM introspection for debugging operating systems and applications and for detecting or preventing intrusions. This section describes some of the ways we are extending or using VM introspection.

3.1 Multi-level replay and introspection

Higher-level replay is needed to add higher-level predicates (e.g., predicates in interpreted languages like Perl and interpreted data storage like databases and file systems) with low overhead and without perturbing the state of the replaying system. To conduct higher-level replay, we instrument the interpreter (e.g., `/usr/bin/perl`, a database server, or a file system) to log the non-deterministic operations, much as VM replay logs non-deterministic events that affect the VM.

Synergistic replay provides the ability to conduct higher-level replay, yet without storing the information logged by the interpreter. Instead, the information logged by the interpreter is discarded while the VM is running. We can regenerate these higher-level logs by replaying the VM (but saving the logs instead of discarding them). The logs would be discarded or saved at the VMM level, or the logged information could be intercepted within the interpreter through a predicate on the interpreter's executable.

3.2 Interfaces

An interesting question for VM introspection is what interfaces should be provided for an outside monitor to gain visibility and control into the executing (or re-executing) virtual machine. The simplest interface directly accesses the state of the virtual machine at the machine level, e.g. virtual disk contents, virtual machine's physical memory (guest physical memory), virtual machine's registers, or the virtual machine's instructions.

We also envision providing higher-level interfaces to higher-level abstractions within the virtual machine. This requires assistance from the operating system or applications

running within the VM, or it requires the VM to duplicate the functionality needed to bridge the semantic gap between the VM's machine-level state and the higher-level abstraction.

A related possibility is providing interfaces within the virtual-machine monitor to allow third-party companies to extend the functionality of the VMM. For example, an extension may request notification on certain VMM events, such as the execution of specific instructions, the access of specific memory locations, or higher-level events.

The VMM may also be able to support more powerful debugging features that are useful for VM introspection. For example, it can provide the ability to set more virtual-hardware breakpoints than are supported by the actual hardware.

4 Fault tolerance and disaster recovery

We are using VM replay to tolerate faults or disasters. The log generated for VM replay can reconstruct the state of the failed machine, either on an on-line backup (hot spare), or on a remote or off-line backup. By replaying the log on one or more live backup VMs, the system can detect faults by noting when the outputs or data in the multiple VMs differ.

The trend toward multiple processors on a single chip or system can make this system more efficient. The low-overhead communication on these systems can be used to compare states and events more frequently and at lower cost. The ability to share disk or memory or CPU caches reduces the need to store duplicate copies of all state, and may help the backup execute at a fraction of the primary system (e.g., the primary may warm the CPU cache).

Detecting transient processor faults requires executing the virtual machine at least twice. These redundant executions will usually (though need not) take place on different processors so as to minimize latency. The VMM can be enhanced with a replication-management layer to manage these redundant executions.

Before an output goes to an external device (e.g., a device that cannot be rolled back), the replicas must compare and agree upon the results.

There are several ways to detect errors. First, the outputs of the replicas may disagree. Second, the state of the replicas (register, memory, disk) can be scanned periodically. It is an error if the states differ while the replicas are at the same point in their executions. Third, the VM replay system itself may detect the error, because the log generated on the primary may be unable to be replayed on a backup. (If more than one fault is to be tolerated, then more than two replicas must vote).

Once an error is detected, there are several strategies for identifying and recovering from the error. To identify the erroneous replica requires more than two replicas. Again, these replicas can be run continuously, or they can be created and/or run only when needed to diagnose an error. Replicas are created from a checkpoint (which has been verified to be correct at a prior time). For simplicity, our current strategy is to create two extra replicas when an error is detected, then to re-run both extra replicas (discarding the original replicas). This is sufficient to recover from transient errors if multiple errors do not re-occur in a short time window.

To roll forward during recovery, the VM replay log must be used to reconstruct the state at least up through the last committed output. Roll forward may proceed through additional log entries, but must not replay a log entry that was unable to be replayed during the failed run. After the roll-forward phase ends, execution proceeds "live", as it did before the fault. It is possible to save external input and use this after the roll-forward phase, to hide the fault from the user and other external entities.

The checkpoint that is the starting point for recovery can be stored in a variety of ways. The simplest (and most expensive) is to keep a complete copy of the VM's state (including disk, memory, memory translation tables, registers). More efficient is to leverage the current state of the VM and to keep an undo log (before images) of all data (e.g., pages, disk blocks) that have been modified since the last checkpoint. This technique is commonly known as copy-on-write. Before the checkpoint can be used, it must be verified. That is, all data in the checkpoint must be compared between two replicas and verified to be the same. (Checkpoints are useful for many purposes beside fault tolerance, of course. General "time travel" can be accomplished through undo logs (before images) and redo logs (after images).)

It is also possible to identify and recover from permanent processor faults. To do so, the system must be able to identify the processor as faulty (e.g., through some hardware check). Or, at least 3 processors must run so as to vote on which one failed.

It is possible to tolerate more than 1 simultaneous fault by increasing the number of live replicas. In general, it takes f+1 replicas to detect an error with up to f faults, and it takes 2f+1 replicas to identify faulty replicas. In addition, it is possible to detect more than f faults with this number of replicas, as long as those faults do not manifest in the same way. One can also increase the number of replicas as needed, as long as the faults are transient.

It is possible to share resources between replicas. However, there must exist some protection against hardware faults (e.g., memory or disk ECC); otherwise a single hardware fault can corrupt the state of multiple replicas.

To share memory between replicas, we can give each replica a virtual copy via copy-on-write (same as `fork` in Unix). As the replicas execute, pages that are modified will become copied. Eventually, most/all pages will require separate physical copies for each replica. To merge these pages back into a single physical copy, the system can periodically compare the pages and merge them when they have the same contents (we call this merge-on-equal). Note that verifying state for a checkpoint is trivial for state that shares a single physical copy. The periodic scans for equality can also be used to detect errors: when the replicas are synchronized, it is an error for the pages to not be equal.

Sharing disk blocks between replicas can be done in exactly the same manner as sharing memory pages. Because disk writes are less frequent, however, some additional optimizations are possible with disk. For example, it is possible to compare the disk outputs of the replicas instead of comparing the contents of each replica's disk. A completely different strategy for disk is to only send verified output to disk. This has the advantage of always keeping only one copy of the disk. Reading the disk can block for all prior writes to be compared and written, or they can read data from a buffer of writes that have not yet been compared and written to disk.

The disk can be included in a checkpoint (probably via copy-on-write undo log-

ging), or VM replay can treat the disk as a source of non-deterministic data and log all reads from disk.

One concern in this system is where to run the code that manages the replication (the replication management layer, or RML). Ideally, most or all code would run above the RML, since the RML protects all code running above it from faults. We are considering trying to run most of the VMM and the device drivers above RML. The code that cannot use the RML for automatic redundant execution can be replicated through another technique, such as hand-written redundant execution or a compiler instrumented fault tolerance.