The Design and Evaluation of a Practical System for Fault-Tolerant Virtual Machines

Daniel J. Scales (VMware) scales@vmware.com
Mike Nelson (VMware) mnelson@vmware.com
Ganesh Venkitachalam (VMware) ganesh@vmware.com

Technical Report VMware-TR-2010-001
May 11, 2010

The Design and Evaluation of a Practical System for Fault-Tolerant Virtual Machines

Daniel J. Scales (VMware) scales@vmware.com
Mike Nelson (VMware) mnelson@vmware.com
Ganesh Venkitachalam (VMware) ganesh@vmware.com

Technical Report VMware-TR-2010-001
May 11, 2010
VMware, Inc.
Palo Alto, CA 94304

Abstract

We have implemented a commercial enterprise-grade system for providing fault-tolerant virtual machines, based on the approach of replicating the execution of a primary virtual machine (VM) via a backup virtual machine on another server. We have designed a complete system in VMware vSphere 4.0 that is easy to use, runs on commodity servers, and typically reduces performance of real applications by less than 10%. Our method for replicating VM execution is similar to that described in Bressoud [3], but we have made a number of significant design changes that greatly improve performance. In addition, an easy-to-use, commercial system that automatically restores redundancy after failure requires many additional components beyond replicated VM execution. We have designed and implemented these extra components and addressed many practical issues encountered in supporting VMs running enterprise applications. In this paper, we describe our basic design, discuss alternate design choices and a number of the implementation details, and provide an evaluation of our performance for both micro-benchmarks and real applications.

Key Words and Phrases: virtual machines, fault tolerance, deterministic replay

1 Introduction

A common approach to implementing fault-tolerant servers is the primary/backup approach [1], where the execution of a primary server is replicated by a backup server. Given that the primary and backup servers execute identically, the backup server can take over serving client requests without any interruption or loss of state if the primary server fails. One method for replicating servers is sometimes referred to as the state-machine approach [13]. The idea is to model the servers as deterministic state machines that are kept in sync by starting them from the same initial state and ensuring that they receive the same input requests in the same order. Since most servers or services have some operations that are not deterministic, extra coordination must be used to ensure that a primary and backup are kept in sync.

Implementing coordination to ensure deterministic execution of physical servers [14] is difficult, particularly as processor frequencies increase and clock synchronization becomes more difficult. In contrast, a virtual machine (VM) running on top of a hypervisor is an excellent platform for implementing the primary/backup approach. A VM can be considered a well-defined state machine whose operations are the operations of the machine being virtualized (including all its devices). As with physical servers, VMs have some non-deterministic operations (e.g. reading a time-of-day clock or delivery of an interrupt), and so extra information must be sent to the backup to ensure that it is kept in sync. Since the hypervisor has full control over the execution of a VM, including delivery of all inputs, the hypervisor is able to capture all the necessary information about non-deterministic operations on the primary VM and to replay these operations correctly on the backup VM.

A system of replication based on virtual machines can replicate individual VMs, allowing some VMs to be replicated and fault-tolerant, while other VMs are not replicated. In addition, technology based on VMs does not require hardware modifications, allowing the system to ride the hardware performance improvement curve of newer microprocessors. A system based on replicated execution of physical servers requires hardware modifications and thus often lags behind the performance curve. Yet another advantage of virtual machines for this application is the possibility of physical separation of the primary and the backup: for example, the replicated virtual machines can be run on physical machines distributed across a campus, which provides more reliability than a primary/backup system running in the same building.

We have implemented fault-tolerant VMs using the primary/backup approach on the VMware vSphere 4.0 platform, which runs fully virtualized x86 virtual machines in a highly-efficient manner. Since VMware vSphere implements a complete x86 virtual machine that can run all operating systems and applications that run on an x86 platform, we are automatically able to provide fault tolerance for any x86 operating systems and applications. The base technology that allows us to record the execution of a primary and ensure that the backup executes identically is known as deterministic replay [15]. VMware vSphere Fault Tolerance (FT) is based on deterministic replay, but adds in the necessary extra protocols and functionality to build a complete fault-tolerant system. In addition to providing hardware fault tolerance, our system restores redundancy by automatically starting a new backup virtual machine on any available server in the local cluster. At this time, the production versions of both deterministic replay and VMware FT support only uni-processor VMs. Recording and replaying the execution of a multi-processor VM is still work in progress, with significant performance issues because nearly every access to shared memory can be a non-deterministic operation.

Bressoud [3] describes a prototype implementation of fault-tolerant VMs for the HP PA-RISC platform. Our approach is similar, but we have made some fundamental changes for performance reasons and investigated a number of design alternatives. In addition, we have had to design and implement many additional components in the system and deal with a number of practical issues to build a complete system that is efficient and usable by customers running enterprise applications. Similar to most other practical systems discussed, we only attempt to deal with fail-stop failures [12], which are server failures that can be detected before the failing server causes an incorrect externally visible action.

The rest of the paper is organized as follows. First, we describe our basic design and detail our fundamental protocols that ensure that no data is lost if a backup VM takes over after a primary VM fails. Then, we describe in detail many of the practical issues that must be addressed to build a correct, robust, fully-functioning, and automated system. We also describe several design choices that arise for implementing fault-tolerant VMs and discuss the tradeoffs in these choices. Next, we give performance results for our implementation for some benchmarks and some real enterprise applications. Finally, we describe related work and conclude.

2 Basic FT Design

Figure 1 shows the basic setup of our system for fault-tolerant VMs. For a given VM for which we desire to provide fault tolerance (the *primary* VM), we run a *backup* VM on a different physical server that is kept in sync and executes identically to the primary virtual machine, though with a small time lag. We say that the two VMs are in *virtual lockstep.* The virtual disks for the VMs are on shared storage (such a Fibre Channel or iSCSI disk array), and therefore accessible to the primary and backup VM for input and output. (We will discuss a design in which the primary and backup VM have separate non-shared virtual disks in Section 4.1.) Only the primary VM advertises its presence on the network, so all network inputs come to the primary VM. Similarly, all other inputs (such as keyboard and mouse) go only to the primary VM.

All input that the primary VM receives is transmitted to the backup VM via a network connection known as the *logging channel.* For server workloads, the dominant input traffic is network and disk. Additional information, as discussed below in Section 2.1, is transmitted as necessary to ensure that the backup VM executed non-deterministic operations in the same way as the primary VM. The end result is that the backup VM always executes identically to the primary VM. However, the outputs of the backup VM are always dropped

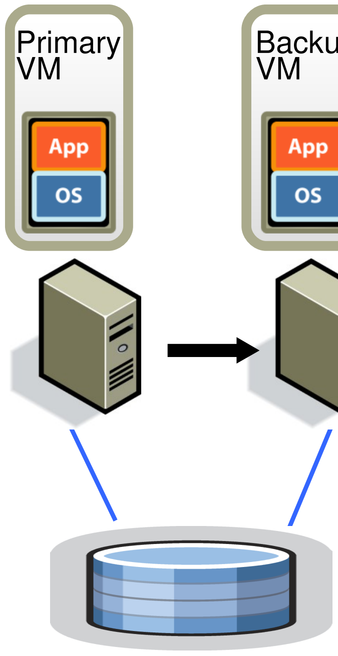

Figure 1: **Basic FT Configuration**.

by the hypervisor, so only the primary produces actual outputs that are returned to clients. As described in Section 2.2, the primary and backup VM must follow a specific protocol, including explicit acknowledgments by the backup VM, in order to ensure that no data is lost if the primary fails.

A crucial issue that is not discussed much in previous work is the actual process of determining quickly whether a primary or backup VM has failed. Our system uses a combination of heartbeating between the relevant servers and monitoring of the traffic on the logging channel. In addition, we must ensure that only one of the primary or backup VM takes over execution, even if there is a split-brain situation where the primary and backup servers have lost communication with each other.

In the following sections, we provide more details on several important areas. In Section 2.1, we give some details on the deterministic replay technology that ensures that primary and backup VMs are kept in sync via the information sent over the logging channel. In Section 2.2, we describe a fundamental rule of our FT protocol that ensures that no data is lost if the primary fails. In Section 2.3, we describe our methods for detecting and responding to a failure in a correct fashion.

2.1 Record-Replay Implementation

As we have mentioned, replicating servers (or VMs) can be modeled as the replication of deterministic state machines. If two deterministic state machines are started in the same initial state and provided the exact same inputs in the same order, then they will go through the same sequences of states and produce the same outputs. In the simplest case, one state machine is the primary, and the other is the backup. If all the inputs go to the primary, then the inputs can be distributed to the backup from the primary via a logging channel. A useful physical computer, when considered as a state machine, has a broad set of inputs ranging from a keyboard device to network input received from a client. In addition, non-deterministic events like virtual interrupts, and non-deterministic operations like reading the clock cycle counter from the processor, affect the state machine. This presents three challenges to a practical hypervisor capable of running any operating system that can run on a physical machine: (1) correctly capturing all the input and non-determinism necessary to ensure deterministic execution of a backup virtual machine, (2) correctly applying the inputs and non-determinism to the backup virtual machine, and (3) doing so in a manner that doesn't degrade performance.

VMware deterministic replay [15] provides exactly this functionality for x86 virtual machines on the VMware vSphere platform. Deterministic replay allows the inputs of a VM and all possible non-determinism associated with the VM execution to be recorded via a stream of log entries written to a log file. The VM execution may be replayed later exactly by reading the log entries from the file. Non-deterministic state transitions can either result from explicit operations executed by the VM that have non-deterministic results (such as reading the time-of-day clock), or asynchronous events (such as interrupts) which create non-determinism because the point at which they interrupt the dynamic instruction stream affects the virtual machine execution.

For non-deterministic operations, sufficient information must be logged to allow the operation to be reproduced with the same state change and output when replaying. For non-deterministic events such as timer interrupts or IO completion interrupts, the exact instruction at which the event occurred must also be recorded. During replay, the event must be delivered at the exact same point in the instruction stream. VMware deterministic replay implements an efficient event recording and event delivery mechanism that employs various techniques, including the use of hardware performance counters developed in conjunction with AMD [2] and Intel [8].

Bressoud [3] mentions dividing the execution of VM into epochs, where non-deterministic events such as interrupts are only delivered at the end of an epoch. The notion of epoch seems to be used as a batching mechanism because it is too expensive to deliver each interrupt separately at the exact instruction where it occurred. However, our event delivery mechanism is efficient enough that VMware deterministic replay has no need to use epochs. The occurrence of each interrupt is recorded and logged as it occurs and efficiently delivered at the appropriate instruction while being replayed.

2.2 FT Protocol

For VMware FT, we use deterministic replay to produce the necessary log entries to record the execution of the primary VM, but instead of writing the log entries to disk, we send them to the backup VM via the logging channel. The backup VM replays the entries in real time, and hence executes identically to the primary VM. However, we must augment the logging entries with a strict FT protocol on the logging channel in order to ensure that we achieve fault tolerance. Our fundamental requirement is the following:

> **Output Requirement**: if the backup VM ever takes over after a failure of the primary, the backup VM will continue executing in a way that is entirely consistent with all outputs that the primary VM has sent to the external world.

Note that after a *failover* occurs (i.e. the backup VM takes over after the failure of the primary VM), the backup VM will likely start executing quite differently from the way the primary VM would have continued executing, because of the many non-deterministic events happening during execution. However, as long as the backup VM satisfies the Output Requirement, no state or data is lost during a failover to the backup VM, and the clients will notice no interruption or inconsistency in their service.

The Output Requirement can be ensured by delaying any external output (typically a network packet) until the backup VM has received all information that will allow it to replay execution at least to the point of that output operation. One necessary condition is that the backup VM must have received all log entries generated prior to the output operation. These log entries will allow it to execute up to the point of the last log entry. However, suppose a failure were to happen immediately after the primary executed the output operation. The backup VM must know that it must keep replaying up to the point of the output operation and only "go live" (stop replaying and take over as the primary VM, as described in Section 2.3) at that point. If the backup were to go live at the point of the last log entry before the output operation, some non-deterministic event (e.g. timer interrupt delivered to the VM) might change its execution path before it executed the output operation.

Given the above constraints, the easiest way to enforce the Output Requirement is to create a special log entry at each output operation. Then, the Output Requirement may be enforced by this specific rule:

> **Output Rule**: the primary VM may not send an output to the external world, until the backup VM has received and acknowledged the log entry associated with the operation producing the output.

If the backup VM has received all the log entries, including the log entry for the output-producing operation, then the backup VM will be able to exactly reproduce the state of the primary VM at that output point, and so if the primary dies, the backup will correctly reach a state that is consistent with that output. Conversely, if the backup VM takes over without receiving all necessary log entries, then its state may quickly diverge such that it is inconsistent with the primary's output. The Output Rule is in some ways analogous to the approach described in [11], where an "externally synchronous" IO can actually be buffered, as long as it is actually written to disk before the next external communication.

Note that the Output Rule does not say anything about stopping the execution of the primary VM. We need only delay the sending of the output, but the VM itself can continue

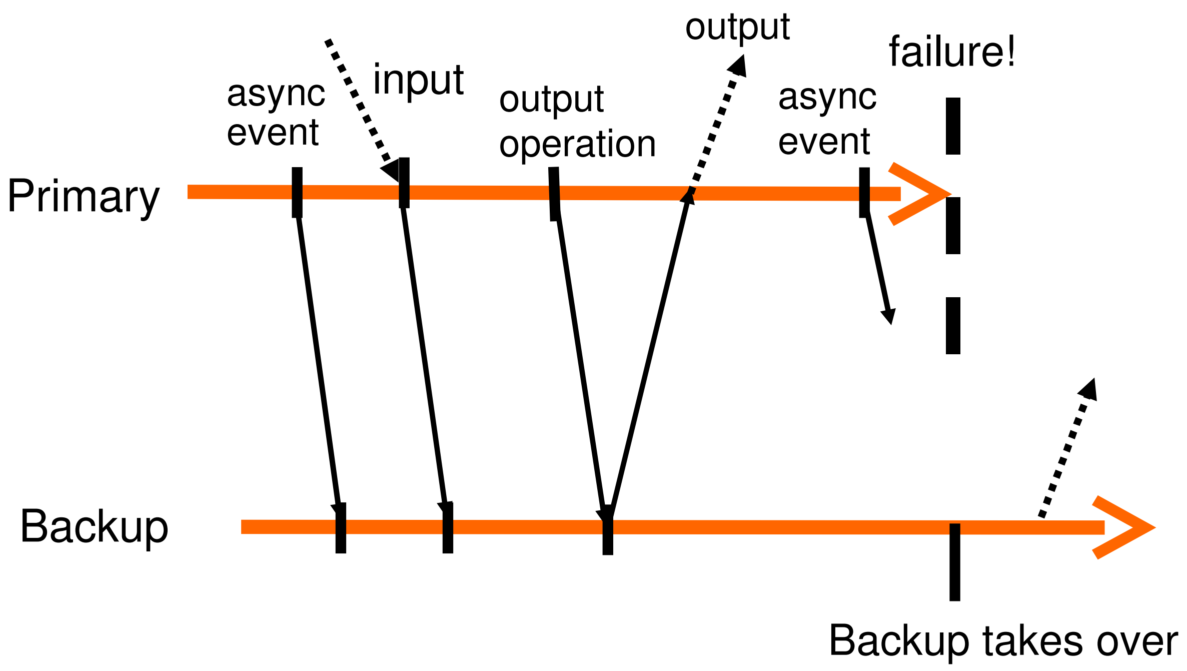

Figure 2: **FT Protocol**.

execution. Since operating systems do non-blocking network and disk outputs with asynchronous interrupts to indicate completion, the VM can easily continue execution and will not necessarily be immediately affected by the delay in the output. In contrast, previous work [3, 9] has typically indicated that the primary VM must be completely stopped prior to doing an output until the backup VM has acknowledged all necessary information from the primary VM.

As an example, we show a chart illustrating the requirements of the FT protocol in Figure 2. This figure shows a timeline of events on the primary and backup VMs. The arrows going from the primary line to the backup line represent the transfer of log entries, and the arrows going from the backup line to the primary line represent acknowledgments. Information on asynchronous events, inputs, and output operations must be sent to the backup as log entries and acknowledged. As illustrated in the figure, an output to the external world is delayed until the primary VM has received an acknowledgment from the backup VM that it has received the log entry associated with an output operation. Given that the Output Rule is followed, the backup VM will be able to take over in a state consistent with the primary's last output. There will be no loss of state even if the primary has had a non-deterministic event since its last output.

As indicated in [3, 9], we cannot guarantee that all outputs are produced exactly once in a failover situation. Without the use of transactions with two-phase commit when the primary intends to send an output, there is no way that the backup can determine if a primary crashed immediately before or after sending its last output. Fortunately, the network infrastructure (including the common use of TCP) is designed to deal with lost packets and identical (duplicate) packets.

Note that incoming packets to the primary may also be lost during a failure of the primary and therefore won't be delivered to the backup. However, incoming packets may be dropped for any number of reasons unrelated to server failure, so the network infrastructure, operating systems, and applications are all written to ensure that they can compensate for lost packets.

2.3 Detecting and Responding to Failure

As mentioned above, the primary and backup VMs must respond quickly if the other VM appears to have failed. If the backup VM fails, the primary VM will *go live* – that is, leave recording mode (and hence stop sending entries on the logging channel) and start executing normally. If the primary VM fails, the backup VM should similarly *go live*, but the process is a bit more complex. Because of its lag in execution, the backup VM will likely have a number of log entries that it has received and acknowledged, but have not yet been consumed because the backup VM hasn't reached the appropriate point in its execution yet. The backup VM must continue replaying its execution from the log entries until it has consumed the last log entry. At that point, the backup VM will stop replaying mode and start executing as a normal VM. In essence, the backup VM has been promoted to the primary VM (and is now missing a backup VM). Since it is no longer a backup VM, the new primary VM will now produce output to the external world when the guest OS does output operations. During the transition to normal mode, there may be some device-specific operations needed to allow this output to occur properly. In particular, for the purposes of networking, VMware FT automatically advertises the MAC address of the new primary VM on the network, so that physical network switches will know on what server the new primary VM is located. In addition, the newly promoted primary VM may need to reissue some disk IOs (as described in Section 3.4).

There are many possible ways to attempt to detect failure of the primary and backup VMs. VMware FT uses UDP heartbeating between servers that are running fault-tolerant VMs to detect when a server may have crashed. In addition, VMware FT monitors the logging traffic that is sent from the primary to the backup VM and the acknowledgments sent from the backup VM to the primary VM. Because of regular timer interrupts, the logging traffic should be regular and never stop for a functioning guest OS. Therefore, a halt in the flow of log entries or acknowledgments could indicate the failure of a VM or a networking problem. A failure is declared if heartbeating or traffic on the logging channel has stopped for longer than a specific timeout (on the order of a few seconds).

However, any such failure detection method is susceptible to a split-brain problem. If the backup server stops receiving heartbeats from the primary server, that may indicate that the primary server has failed, or it may just mean that all network connectivity has been lost between still functioning servers. If the backup VM then goes live while the primary

VM is actually still running, there will likely be data corruption and problems for the clients communicating with the VM. Hence, we must ensure that only one of the primary or backup VM goes live when a failure is detected. To avoid split-brain problems, we make use of the shared storage that is used to store the virtual disks of the VM. At the point where either a primary or backup VM wants to go live, it executes an atomic test-and-set operation on the shared storage. If the operation succeeds, the VM is allowed to go live. If the operation fails, then the other VM must have already gone live, so the current VM actually halts itself ("commits suicide"). If the VM cannot access the shared storage when trying to do the atomic operation, then it just waits until it can. Note that if shared storage is not accessible because of some failure in the storage network, then the VM would likely not be able to do useful work anyway because the virtual disks reside on the same shared storage. Thus, using shared storage to resolve split-brain situations does not introduce any extra unavailability.

One final aspect of the design is that once a failure has occurred and one of the VMs has gone live, VMware FT automatically restores redundancy by starting a new backup VM on another host. Though this process is not covered in most previous work, it is fundamental to making fault-tolerant VMs useful and requires careful design. More details are given in Section 3.1.

2.4 Go-live Points

The use of deterministic replay for fault tolerance purposes has driven us to add an interesting mechanism to our replay implementation. Because of network issues or the failure of the primary at any point, the stream of log entries being read and replayed by the backup can be terminated at any point. The possibility of termination at any point in the log can permeate the deterministic replay implementation, since each potential consumer of a log entry (such as a virtual device implementation) would need to check for and deal with the fact that an expected log entry is not available. For instance, given previous log entries and its current state, a virtual device implementation may expect a number of additional log entries about IO completions. The code that is replaying the device will have to be written to check for the end of the log stream, exit some possibly complex replaying code, and restore the device to a reasonable state so that the VM can go live.

To alleviate this burden on many components of the system, we have implemented *go-live points*. Any individual log entry can be marked as a go-live point. The idea is that a log entry that is marked as a go-live point represents the last log entry in a series of log entries necessary for replaying an instruction or a particular device operation. If a particular operation or instruction requires several log entries to be recorded, then only the last log entry would be marked as a go-live point. In practice, the hypervisor automatically marks the last new log entry as a go-live point when it has completed all event and device processing for a given instruction.

Go-live points are used during replaying as follows. While all log entries read from the logging channel are buffered by the hypervisor on the virtual machine that is replaying, only the log entries up to the last go-live point are allowed to be consumed by the replaying (backup) VM. That is, the replaying VM will stall after consuming the last log entry tagged as a go-live point until another series of log entries containing a log entry with a go-live point has been fetched by the hypervisor. The result is that if there is a series of log entries associated with a device operation, the virtual device implementation can assume that all the needed log entries will be available if the first log entry is encountered. Thus, the virtual device implementation does not have to do all the extra checking and recovery code needed if the log entries could be terminated at any point. Similarly, whenever a single instruction executed on behalf of the virtual machine generates multiple log entries, the hypervisor of the replaying virtual machine begins the emulation of that instruction only if all the log entries necessary for completing the emulation of that instruction are available. The tagging scheme doesn't introduce any significant delay of the replaying VM, since the hypervisor of the recording (primary) VM guarantees that last log entry of each single instruction emulation or a device operation is marked as a go-live point. Since the backup VM cannot be significantly delayed, the primary VM is also not affected by the use of go-live points.

3 Practical Implementation of FT

Section 2 described our fundamental design and protocols for FT. However, to create a usable, robust, and automatic system, there are a great many other components that must be designed and implemented.

3.1 Starting and Restarting FT VMs

One of the biggest additional components that must be designed is the mechanism for starting a backup VM in the same state as a primary VM. This mechanism will also be used when re-starting a backup VM after a failure has occurred. Hence, this mechanism must be usable for a running primary VM that is in an arbitrary state (i.e. not just starting up). In addition, we would prefer that the mechanism does not significantly disrupt the execution of the primary VM, since that will directly affect any current clients of the VM.

For VMware FT, we adapted the existing VMotion functionality of VMware vSphere. VMware VMotion [10] allows the migration of a running VM from one server to another server with minimal disruption – VM pause times are typically less than a second. We created a modified form of VMotion that creates an exact running copy of a VM on a remote server, but without destroying the VM on the local server. That is, our modified *FT VMotion* clones a VM to a remote host rather than migrating it. The FT VMotion also sets up a logging channel, causes the source VM to enter logging mode as the primary, and the destination VM to enter replay mode as the new backup. Like normal VMotion, FT VMotion typically interrupts the execution of the primary VM by less than a second. Hence, enabling FT on a running VM is an easy, non-disruptive operation.

Another aspect of starting a backup VM is choosing a server on which to run it. Fault-tolerant VMs run in a cluster of servers that have access to shared storage, and so all VMs can typically run on any servers in the cluster. This flexibility allows VMware vSphere to

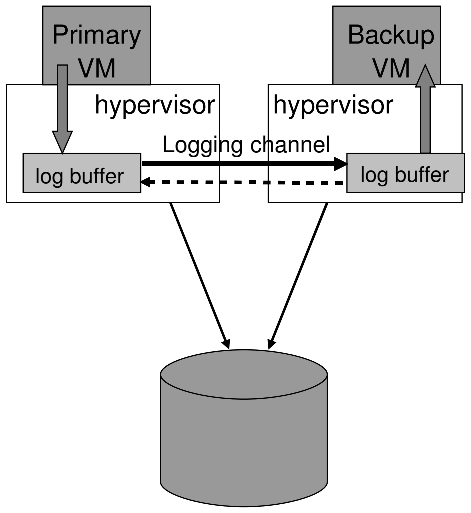

Figure 3: **FT Logging Buffers and Channel**.

restore FT redundancy even when one or more servers have failed. VMware vSphere implements a clustering service that maintains management and resource information. When a failure happens and a primary VM now needs a new backup VM to re-establish redundancy, the primary VM informs the clustering service that it needs a new backup. The clustering service determines the best server on which to run the backup VM based on resource allocations, usage, and other constraints. Then the clustering service automatically invokes an FT VMotion to create the new backup VM. Of course, there are many additional complexities, such as retrying if a first attempt to create a backup fails and automatically detecting when a server in the cluster becomes newly available. The end result is that VMware FT typically can re-establish VM redundancy within minutes of a server failure, all without any noticeable interruption in the execution of a fault-tolerant VM.

3.2 Managing the Logging Channel

There are a number of interesting implementation details in managing the traffic on the logging channel. In our implementation, the hypervisors maintain a large buffer for logging entries for the primary and backup VMs. As the primary VM executes, it produces log entries into the log buffer, and similarly, the backup VM consumes log entries from its log buffer. The contents of the primary's log buffer are flushed out to the logging channel as soon as possible, and log entries are read into the backup's log buffer from the logging channel as soon as they arrive. The backup sends acknowledgments back to the primary each time that it reads some log entries from the network into its log buffer. These acknowledgments allow VMware FT to determine when an output that is delayed by the Output Rule can be sent. Figure 3 illustrates this process.

If the backup VM encounters an empty log buffer when it needs to read the next log entry, it will stop execution until a new log entry is available. Since the backup VM is not communicating externally, this pause will not affect any clients of the VM. Similarly, if the primary VM encounters a full log buffer when it needs to write a log entry, it must stop execution until log entries can be flushed out. This stop in execution is a natural flow-control mechanism that slows down the primary VM when it is producing log entries at too fast a rate. However, this pause can affect clients of the VM, since the primary VM will be completely stopped and unresponsive until it can log its entry and continue execution. Therefore, our implementation must be designed to minimize the possibility that the primary log buffer fills up.

One reason that the primary log buffer may fill up is because the bandwidth of the logging channel is too low to carry the volume of log entries being produced. While the bandwidth on the logging channel is typically not high (as seen in Section 5), we strongly recommend the use of a 1 Gbit/s network for the logging channel to avoid any possibility of a bottleneck.

Another reason that the primary log buffer may fill up is because the backup VM is executing too slowly and therefore consuming log entries too slowly. In general, the backup VM must be able to replay an execution at roughly the same speed as the primary VM is recording the execution. Fortunately, the overhead of recording and replaying in VMware deterministic replay is roughly the same. However, if the server hosting the backup VM is heavily loaded with other VMs (and hence overcommitted on resources), the backup VM may not be able to get enough CPU and memory resources to execute as fast as the primary VM, despite the best efforts of the backup hypervisor's VM scheduler.

Beyond avoiding unexpected pauses if the log buffers fill up, there is another reason why we don't wish the execution lag to become too large. If the primary VM fails, the backup VM must "catch up" by replaying all the log entries that it has already acknowledged before it goes live and starts communicating with the external world. The time to finish replaying is basically the execution lag time at the point of the failure. Hence, the time for the backup to go live is roughly equal to the failure detection time plus the current execution lag time. So, we don't wish the execution lag time to be large (more than a second), since that will add significant time to the failover time (the time for the backup to go live).

Therefore, we have an additional mechanism to slow down the primary VM to prevent the backup VM from getting too far behind. In our protocol for sending and acknowledging log entries, we send additional information to determine the real-time execution lag between the primary and backup VMs. Typically the execution lag is less than 100 milliseconds. If the backup VM starts having a significant execution lag (say, more than 1 second), VMware FT starts slowing down the primary VM by informing the scheduler to give it a slightly smaller share of the CPU (initially by just a few percent). We use a slow feedback loop, which will try to gradually pinpoint the appropriate CPU share for the primary VM that will allow the backup VM to match its execution. If the backup VM continues to lag behind, we continue to gradually reduce the primary VM's CPU share. Conversely, if the backup VM catches up, we gradually increase the primary VM's CPU share until the backup VM returns to having a slight lag.

Note that such slowdowns of the primary VM are very rare, and typically happen only when the system is under extreme stress. All the performance numbers of Section 5 include the cost of any such slowdowns.

3.3 Operation on FT VMs

Another practical matter is dealing with the various control operations that may be applied to the primary VM. For example, if the primary VM is explicitly powered off, the backup VM should be stopped as well, and not attempt to go live. As another example, any resource management change on the primary (such as increased CPU share) should also be applied to the backup. For these kind of operations, special control entries are sent on the logging channel from the primary to the backup, in order to effect the appropriate operation on the backup.

In general, most operations on the VM should be initiated only on the primary VM. VMware FT then sends any necessary control entry to cause the appropriate change on the backup VM. The only operation that can be done independently on the primary and backup VM is VMotion. That is, the primary and backup VM can each be VMotioned independently to other hosts. Note that VMware FT ensures that neither VM is VMotioned to the server where the other VM is, since that situation would no longer provide fault tolerance.

VMotion of a primary VM adds some complexity over a normal VMotion, since the backup VM must disconnect from the source primary and re-connect to the destination primary VM at the appropriate time. VMotion of a backup VM has a similar issue, but adds an additional complexity. For a normal VMotion, we require that all outstanding disk IOs be quiesced (i.e. completed) just as the final switchover on the VMotion occurs. For a primary VM, this quiescing is easily handled by waiting until the physical IOs complete and delivering these completions to the VM. However, for a backup VM, there is no easy way to cause all IOs to be completed at any required point, since the backup VM must replay the primary VM's execution and complete IOs at the same execution point. The primary VM may be running a workload in which there are always disk IOs in flight during normal execution. VMware FT has a unique method to solve this problem. When a backup VM is at the final switchover point for a VMotion, it requests via the logging channel that the primary VM temporarily quiesce all of its IOs. The backup VM's IOs will then naturally be quiesced as well at a single execution point as it replays the primary VM's execution of the quiescing operation.

3.4 Implementation Issues for Disk IOs

There are a number of subtle implementation issues related to disk IO. First, given that disk operations are non-blocking and so can execute in parallel, simultaneous disk operations that access the same disk location can lead to non-determinism. Also, our implementation of disk IO uses DMA directly to/from the memory of the virtual machines, so simultaneous disk operations that access the same memory pages can also lead to non-determinism. Our solution is generally to detect any such IO races (which are rare), and force such racing disk operations to execute sequentially in the same way on the primary and backup. Interestingly, a single disk read operation can cause a race as well, since its scatter-gather array could reference the same block of memory multiple times, hence leaving the final contents of the memory block undetermined. Our solution is to detect this racing IO as well, and in this case ensure that the final contents of memory are sent on the logging channel, so the backup ends up with the same memory contents.

Second, a disk operation can also race with a memory access by an application (or OS) in a VM, because the disk operations directly access the memory of a VM via DMA. For example, there could be a non-deterministic result if an application/OS in a VM is reading a memory block at the same time a disk read is occurring to that block. This situation is also unlikely, but we must detect it and deal with it if it happens. One solution is to set up page protection temporarily on pages that are targets of disk operations. The page protections result in a trap if the VM happens to make an access to a page that is also the target of an outstanding disk operation, and the VM can be paused until the disk operation completes. Because changing MMU protections on pages is an expensive operation, we choose instead to use *bounce buffers*. A bounce buffer is a temporary buffer that has the same size as the memory being accessed by a disk operation. A disk read operation is modified to read the specified data to the bounce buffer, and the data is copied to guest memory only as the IO completion is delivered. Similarly, for a disk write operation, the data to be sent is first copied to the bounce buffer, and the disk write is modified to write data from the bounce buffer. The use of the bounce buffer can slow down disk operations, but we have not seen it cause any noticeable performance differences.

Third, there are some issues associated with disk IOs that are outstanding (i.e. not completed) on the primary when a failure happens, and the backup takes over. There is no way for the newly-promoted primary VM to be sure if the disk IOs were issued to the disk or completed successfully. In addition, because the disk IOs were not issued externally on the backup VM, there will be no explicit IO completion for them as the newly-promoted primary VM continues to run, which would eventually cause the guest operating system in the VM to start an abort or reset procedure. Therefore, we would like to ensure that a completion is sent to the VM for each pending IO. We could send an error completion that indicates that each IO failed, since it is acceptable to return an error even if the IO completed successfully. However, the guest OS might not respond well to errors from its local disk. Instead, we re-issue the IOs during the go-live process of the VM. Because we have eliminated all races and all IOs specify directly which memory and disk blocks are accessed, these disk operations can be re-issued even if they have already completed successfully (i.e. they are idempotent).

3.5 Implementation Issues for Network IO

VMware vSphere provides many performance optimizations for VM networking. Many of these optimizations are based on the hypervisor asynchronously updating the state of the virtual machine's network device. For example, receive buffers can be updated directly by the hypervisor while the VM is executing. Unfortunately these asynchronous updates to a

VM's state add non-determinism. Unless we can guarantee that all updates happen at the same point in the instruction stream on the primary and the backup, the backup's execution can diverge from that of the primary.

The biggest change to the networking emulation code for fault tolerance is the elimination of the asynchronous network optimizations. All updates to VM networking state must be done while the VM is not executing instructions so we can log the updates and replay the updates on the backup at the same point in the instruction stream. The code that asynchronously updates VM ring buffers with incoming packets has been modified to instead force the guest to trap to the hypervisor where it can log the updates and then apply them to the VM. Similarly, code that previously pulled packets out of transmit queues asynchronously has been disabled for FT and instead we require transmits to be done through a trap to the hypervisor (except as noted below).

The elimination of the asynchronous updates of the network device combined with the delaying of sending packets described in Section 2.2 has provided some performance challenges for networking. We've taken two approaches to improving VM network performance while running FT. First, we implemented clustering optimizations to reduce VM traps and interrupts. When we are streaming data at a sufficient bit rate, we are able to do one transmit trap per group of packets and, in the best case, zero traps, since we can transmit the packets as part of receiving new packets. Likewise, we can reduce the number of interrupts to the VM for incoming packets by only posting the interrupt for a group of packets.

Our second performance optimization for networking involves reducing the delay for transmitted packets. As noted earlier, we have to delay all transmitted packets until we get an acknowledgment from the backup that it has received the appropriate log entries. The key to reducing the transmit delay is to reduce the time required to send a log message to the backup and get an acknowledgment. Our primary optimizations in this area involve ensuring that sending and receiving log entries and acknowledgments can all be done without any thread context switch. The VMware vSphere hypervisor allows functions to be registered with the TCP stack that will be called from a deferred-execution context (similar to a tasklet in Linux) whenever TCP data is received. This allows us to quickly handle any incoming log messages on the backup and any acknowledgments received by the primary without any thread context switches. In addition, when the primary VM enqueues a packet to be transmitted, we force an immediate log flush of the associated output log entry (as described in Section 2.2) by scheduling a deferred-execution context to do the flush.

4 Design Alternatives

In our implementation of VMware FT, we have explored a number of interesting design alternatives. In this section, we explore some of these alternatives.

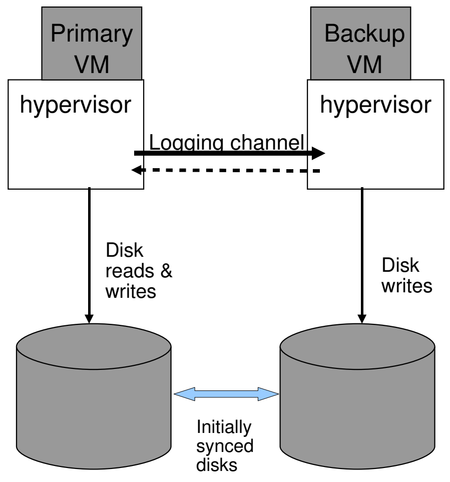

Figure 4: **FT Non-shared Disk Configuration**.

4.1 Shared vs. Non-shared Disk

In our default design, the primary and backup VMs share the same virtual disks. Therefore, the content of the shared disks is naturally correct and available if a failover occurs. Essentially, the shared disk is considered external to the primary and backup VMs, so any write to the shared disk is considered a communication to the external world. Therefore, only the primary VM does actual writes to the disk, and writes to the shared disk must be delayed in accordance with the Output Rule. The shared disk model is the one used in [3, 9, 7].

An alternative design is for the primary and backup VMs to have separate (non-shared) virtual disks. In this design, the backup VM does do all disk writes to its virtual disks, and in doing so, it naturally keeps the contents of its virtual disks in sync with the contents of the primary VM's virtual disks. Figure 4 illustrates this configuration. In the case of non-shared disks, the virtual disks are essentially considered part of the internal state of each VM. Therefore, disk writes of the primary do not have to be delayed according to the Output Rule. The non-shared design is quite useful in cases where shared storage is not accessible to the primary and backup VMs. This may be the case because shared storage is unavailable or too expensive, or because the servers running the primary and backup VMs are far apart ("long-distance FT"). One disadvantage of the non-shared design is that the two copies of the virtual disks must be explicitly synced up in some manner when fault tolerance is first enabled. In addition, the disks can get out of sync after a failure, so they must be explicitly resynced when the backup VM is restarted after a failure. That is, FT VMotion must not only sync the running state of the primary and backup VMs, but also their disk state.

In the non-shared-disk configuration, there may be no shared storage to use for dealing with a split-brain situation. In this case, the system could use some other external tiebreaker, such as a third-party server that both servers can talk to. If the servers are part of a cluster with more than two nodes, the system could alternatively use a majority algorithm based on cluster membership. In this case, a VM would only be allowed to go live if it is running on a server that is part of a communicating sub-cluster that contains a majority of the original nodes.

4.2 Executing Disk Reads on the Backup VM

In our default design, the backup VM never reads from its virtual disk (whether shared or non-shared). Since the disk read is considered an input, it is natural to send the results of the disk read to the backup VM via the logging channel.

An alternate design is to have the backup VM execute disk reads and therefore eliminate the logging of disk read data. This approach can greatly reduce the traffic on the logging channel for workloads that do a lot of disk reads. However, this approach has a number of subtleties. It may slow down the backup VM's execution, since the backup VM must execute all disk reads and wait if they are not physically completed when it reaches the point in the VM execution where they completed on the primary.

Also, some extra work must be done to deal with failed disk read operations. If a disk read by the primary succeeds but the corresponding disk read by the backup fails, then the disk read by the backup must be retried until it succeeds, since the backup must get the same data in memory that the primary has. Conversely, if a disk read by the primary fails, then the contents of the target memory must be sent to the backup via the logging channel, since the contents of memory will be undetermined and not necessarily replicated by a successful disk read by the backup VM.

Finally, there is a subtlety if this disk-read alternative is used with the shared disk configuration. If the primary VM does a read to a particular disk location, followed fairly soon by a write to the same disk location, then the disk write must be delayed until the backup VM has executed the first disk read. This dependence can be detected and handled correctly, but adds extra complexity to the implementation.

In Section 5.1, we give some performance results indicating that executing disk reads on the backup can cause some slightly reduced throughput (1-4%) for real applications, but can also reduce the logging bandwidth noticeably. Hence, executing disk reads on the backup VM may be useful in cases where the bandwidth of the logging channel is quite limited.

5 Performance Evaluation

In this section, we do a basic evaluation of the performance of VMware FT for a number of application workloads and networking benchmarks. For these results, we run the primary and backup VMs on identical servers, each with eight Intel Xeon 2.8 Ghz CPUs and 8 Gbytes of RAM. The servers are connected via a 10 Gbit/s crossover network, though as will be seen in all cases, much less than 1 Gbit/s of network bandwidth is used. Both servers access their shared virtual disks from an EMC Clariion connected through a standard 4 Gbit/s Fibre Channel network. The client used to drive some of the workloads is connected to the servers via a 1 Gbit/s network.

	performance (FT / non-FT)	logging bandwidth
SPECJbb2005	0.98	1.5 Mbits/sec
Kernel Compile	0.95	3.0 Mbits/sec
Oracle Swingbench	0.99	12 Mbits/sec
MS-SQL DVD Store	0.94	18 Mbits/sec

Table 1: Basic Performance Results

The applications that we evaluate in our performance results are as follows. SPECJbb2005 is an industry-standard Java application benchmark that is very CPU- and memory-intensive and does very little IO. Kernel Compile is a workload that runs a compilation of the Linux kernel. This workload does some disk reads and writes, and is very CPU- and MMU-intensive, because of the creation and destruction of many compilation processes. Oracle Swingbench is a workload in which an Oracle 11g database is driven by the Swingbench OLTP (online transaction processing) workload. This workload does substantial disk and networking IO, and has eighty simultaneous database sessions. MS-SQL DVD Store is a workload in which a Microsoft SQL Server 2005 database is driven by the DVD Store benchmark, which has sixteen simultaneous clients.

5.1 Basic Performance Results

Table 1 gives basic performance results. For each of the applications listed, the second column gives the ratio of the performance of the application when FT is enabled on the VM running the server workload vs. the performance when FT is not enabled on the same VM. For SPECJbb2005, Kernel Compile, Oracle Swingbench, and MS-SQL DVD Store, the performance measures are, respectively, business operations per second, compile time in seconds, transactions per second, and operations per second. The ratios are calculated so that a value less than 1 indicates that the FT workload is slower. Clearly, the overhead for enabling FT on these representative workloads is less than 10%. SPECJbb2005 is completely compute-bound and has no idle time, but performs well because it has minimal non-deterministic events beyond timer interrupts. The other workloads do disk IO and have some idle time, so some of the overhead of deterministic replay and the FT protocol may be hidden by the fact that the FT VMs have less idle time. However, the general conclusion is that VMware FT is able to support fault-tolerant VMs with a reasonable performance overhead.

In the third column of the table, we give the average bandwidth of data sent on the logging channel when these applications are run. For these applications, the logging bandwidth is quite reasonable and easily satisfied by a 1 Gbit/s network. In fact, the low bandwidth requirements indicate that multiple FT workloads can share the same 1 Gbit/s network without any negative performance effects.

For VMs that run common guest operating systems like Linux and Windows, we have found that the typical logging bandwidth while the guest OS is idle is 0.5-1.5 Mbits/sec.

The "idle" bandwidth is largely the result of recording the delivery of timer interrupts. For a VM with an active workload, the logging bandwidth is dominated by the network and disk inputs that must be sent to the backup – the network packets that are received and the disk blocks that are read from disk. We have found that a useful heuristic for the network bandwidth is:

> **FT logging bandwidth** = 1 Mbit/s + 1.2 * (average disk read throughput [Mbits/s] + average network receives [Mbits/s])

The factor of 1.2 is a "fudge factor" that approximates the extra logging bandwidth needed for disk and network IOs aside from the input data, including the log entry headers and the extra entries for completion interrupts. Hence, the logging bandwidth can be much higher than those measured in Table 1 for applications that have very high network receive or disk read bandwidth. For these kinds of applications, the bandwidth of the logging channel could be a bottleneck, especially if there are other uses of the logging channel.

The relatively low bandwidth needed over the logging channel for many real applications makes replay-based fault tolerance quite attractive for a long-distance configuration using non-shared disks. For long-distance configurations where the primary and backup might be separated by 1-100 kilometers, optical fiber can easily support bandwidths of 100-1000 Mbit/s with latencies of less than 10 milliseconds. For the applications in Table 1, a bandwidth of 100-1000 Mbit/s should be sufficient for good performance. Note, however, that the extra round-trip latency between the primary and backup may cause network and disk outputs to be delayed by up to 20 milliseconds. The long-distance configuration will only be appropriate for applications whose clients can tolerate such an additional latency on each request.

For two applications, we have measured the performance impact of executing disk reads on the backup VM (as described in Section 4.2) vs. sending disk read data over the logging channel. For Oracle Swingbench, throughput is about 4% slower when executing disk reads on the backup VM; for MS-SQL DVD Store, throughput is about 1% slower. Meanwhile, the logging bandwidth is decreased from 12 Mbits/sec to 3 Mbits/sec for Oracle Swingbench, and from 18 Mbits/sec to 8 Mbits/sec for MS-SQL DVD Store. Clearly, the bandwidth savings could be much greater for applications with much greater disk read bandwidth. As mentioned in Section 4.2, it is expected that the performance might be somewhat worse when disk reads are executed on the backup VM. However, for cases where the bandwidth of the logging channel is limited (for example, a long-distance configuration), executing disk reads on the backup VM may be useful.

5.2 Network Benchmarks

Networking benchmarks can be quite challenging for our system for a number of reasons. First, high-speed networking can have a very high interrupt rate, which requires the logging and replaying of asynchronous events at a very high rate. Second, benchmarks that receive packets at a high rate will cause a high rate of logging traffic, since all such packets must be sent to the backup via the logging channel. Third, benchmarks that send packets will be

	base bandwidth	FT bandwidth	logging bandwidth
Receive (1Gb)	940	604	730
Transmit (1Gb)	940	855	42
Receive (10Gb)	940	860	990
Transmit (10Gb)	940	935	60

Table 2: Performance of Network Transmit and Receive (all in Mbit/s)

subject to the Output Rule, which delays the sending of network packets until the appropriate acknowledgment from the backup is received. This delay will increase the measured latency to a client. This delay could also decrease network bandwidth to a client, since network protocols (such as TCP) may have to decrease the network transmission rate as the round-trip latency increases.

Table 2 gives our results for a number of measurements made by the standard `netperf` benchmark. In all these measurements, the client VM and primary VM are connected via a 1 Gbit/s network. The first two rows give send and receive performance when the primary and backup hosts are connected by a 1 Gbit/s network. The third and fourth rows give the send and receive performance when the primary and backup servers are connected by a 10 Gbit/s network, which not only has higher bandwidth, but also lower latency than the 1 Gbit/s network. As a rough measure, the ping time between hypervisors for the 1 Gbit/s connection is about 150 microseconds, while the ping time for a 10 Gbit/s connection is about 90 microseconds.

When FT is not enabled, the primary VM can achieve close (940 Mbit/s) to the 1 Gbit/s line rate for both transmits and receives. When FT is enabled for receive workloads, the logging bandwidth is very large, since all the incoming network packets must be sent on the logging channel. The logging channel can therefore become a bottleneck, as shown for the results for the 1 Gbit/s logging network. The effect is much less for the 10 Gbit/s logging network. When FT is enabled for transmit workloads, the logging bandwidth is significant since all the network interrupts must still be logged. However, the achievable network transmit bandwidths are higher than the network receive bandwidths. Overall, we see that FT can limit network bandwidths significantly at very high transmit and receive rates, but high absolute rates are still achievable.

6 Related Work

Bressoud and Schneider [3] described the initial idea of implementing fault tolerance for virtual machines via software contained completely at the hypervisor level. They demonstrated the feasibility of keeping a backup virtual machine in sync with a primary virtual machine via a prototype for servers with HP PA-RISC processors. However, due to limitations of the PA-RISC architecture, they could not implement fully secure, isolated virtual machines.

Also, they did not implement any method of failure detection or attempt to address any of the practical issues described in Section 3. More importantly, they imposed a number of constraints on their FT protocol that were unnecessary. First, they imposed a notion of epochs, where asynchronous events are delayed until the end of a set interval. The notion of an epoch is unnecessary – they may have imposed it because they could not replay individual asynchronous events efficiently enough. Second, they required that the primary VM stop execution essentially until the backup has received and acknowledged all previous log entries. However, only the output itself (such as a network packet) must be delayed – the primary VM itself may continue executing.

Bressoud [4] describes a system that implements fault tolerance in the operating system (Unixware), and therefore provides fault tolerance for all applications that run on that operating system. The system call interface becomes the set of operations that must be replicated deterministically. This work has similar limitations and design choices as the hypervisor-based work.

Napper [9] and Friedman [7] describe implementations of fault-tolerant Java virtual machines. They follow a similar design to ours and Bressoud's in sending information about inputs and non-deterministic operations on a logging channel. Like Bressoud, they do not appear to focus on detecting failure and re-establishing fault tolerance after a failure. In addition, their implementation is limited to providing fault tolerance for applications that run in a Java virtual machine. These systems attempt to deal with issues of multi-threaded Java applications, but require either that all data is correctly protected by locks or enforce a serialization on access to shared memory.

Dunlap [6] describes an implementation of deterministic replay targeted towards debugging application software on a paravirtualized system. Our work supports arbitrary operating systems running inside virtual machines and implements fault tolerance support for these VMs, which requires much higher levels of stability and performance.

Cully [5] describes an alternative approach for supporting fault-tolerant VMs and its implementation in a project called Remus. With this approach, the state of a primary VM is repeatedly checkpointed during execution and sent to a backup server, which collects the checkpoint information. The checkpoints must be executed very frequently (many times per second), since external outputs must be delayed until a following checkpoint has been sent and acknowledged. The advantage of this approach is that it applies equally well to uni-processor and multi-processor VMs. The main issue is that this approach has very high network bandwidth requirements to send the incremental changes to memory state at each checkpoint. The results for Remus presented in [5] show 100% to 225% slowdown for kernel compile and SPECweb benchmarks, when attempting to do 40 checkpoints per second using a 1 Gbit/s network connection for transmitting changes in memory state. There are a number of optimizations that may be useful in decreasing the required network bandwidth, but it is not clear that reasonable performance can be achieved with a 1 Gbit/s connection. In contrast, our record-replay based approach can achieve less than 10% overhead, typically with on the order of 10-50 Mbit/s bandwidth required between the primary and backup hosts.

7 Conclusion and Future Work

We have designed and implemented an efficient and complete system in VMware vSphere that provides fault tolerance (FT) for virtual machines running on servers in a cluster. Our design is based on replicating the execution of a primary VM via a backup VM on another host using VMware deterministic replay. If the server running the primary VM fails, the backup VM takes over immediately with no interruption or loss of data.

Overall, the performance of fault-tolerant VMs under VMware FT on commodity hardware is excellent, and shows less than 10% overhead for some typical applications. Most of performance cost of VMware FT comes from the overhead of using VMware deterministic replay to keep the primary and backup VMs in sync. The low overhead of VMware FT therefore derives from the efficiency of VMware deterministic replay. In addition, the logging bandwidth required to keep the primary and backup in sync is typically quite small, often less than 100 Mbit/s. Because the logging bandwidth is quite small in most cases, it seems feasible to implement configurations where the primary and backup VMs are separated by long distances (1-100 kilometers).

Our results with VMware FT have shown that an efficient implementation of fault-tolerant VMs can be built upon deterministic replay. Such a system can transparently provide fault tolerance for VMs running any operating systems and applications with minimal overhead. However, for a system of fault-tolerant VMs to be useful for customers, it must also be robust, easy-to-use, and highly automated. A usable system requires many other components beyond replicated execution of VMs. In particular, VMware FT automatically restores redundancy after a failure, by finding an appropriate server in the local cluster and creating a new backup VM on that server. By addressing all the necessary issues, we have demonstrated a system that is usable for real applications in customer's datacenters.

In the future, we are interested in investigating the performance characteristics of the long-distance FT configurations mentioned above. We are also interested in extending our system to deal with partial hardware failure. By partial hardware failure, we mean a partial loss of functionality or redundancy in a server that doesn't cause corruption or loss of data. An example would be the loss of all network connectivity to the VM, or the loss of a redundant power supply in the physical server. If a partial hardware failure occurs on a server running a primary VM, in many cases (but not all) it would be advantageous to fail over to the backup VM immediately. Such a failover could immediately restore full service for a critical VM, and ensure that the VM is quickly moved off of a potentially unreliable server.

Acknowledgments

We would like to thank Krishna Raja, who generated many of the performance results. There were numerous people involved in the implementation of VMware FT. Core implementors of deterministic replay and the base FT functionality included Lan Huang, Eric Lowe, Slava Malyugin, Alex Mirgorodskiy, Boris Weissman, and Min Xu. In addition, there are many other people involved in the higher-level management of FT in VMware vCenter and in implementation issues related to specific virtual devices besides network and disk. Karyn Ritter did an excellent job managing much of the work.

References

[1] Alsberg, P., and Day, J. A Principle for Resilient Sharing of Distributed Resources. In *Proceedings of the Second International Conference on Software Engineering* (1976), pp. 627–644.

[2] AMD Corporation. *AMD64 Architecture Programmer's Manual.* Sunnyvale, CA.

[3] Bressoud, T., and Schneider, F. Hypervisor-based Fault Tolerance. In *Proceedings of SOSP 15* (Dec. 1995).

[4] Bressoud, T. C. TFT: A Software System for Application-Transparent Fault Tolerance. In *Proceedings of the Twenty-Eighth Annual International Symposium on Fault-Tolerance Computing* (June 1998), pp. 128–137.

[5] Cully, B., Lefebvre, G., Meyer, D., Feeley, M., Hutchison, N., and Warfield, A. Remus: High Availability via Asynchronous Virtual Machine Replication. In *Proceedings of the Fifth USENIX Symposium on Networked Systems Design and Implementation* (Apr. 2008), pp. 161–174.

[6] Dunlap, G. W., King, S. T., Cinar, S., Basrai, M., and Chen, P. M. ReVirt: Enabling Intrusion Analysis through Virtual Machine Logging and Replay. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation* (Dec. 2002).

[7] Friedman, R., and Kama, A. Transparent Fault-Tolerant Java Virtual Machine. In *Proceedings of Reliable Distributed System* (Oct. 2003), pp. 319–328.

[8] Intel Corporation. *IntelÂ® 64 and IA-32 Architectures Software Developer's Manuals.* Santa Clara, CA.

[9] Napper, J., Alvisi, L., and Vin, H. A Fault-Tolerant Java Virtual Machine. In *Proceedings of the International Conference on Dependable Systems and Networks* (June 2002), pp. 425–434.

[10] Nelson, M., Lim, B.-H., and Hutchins, G. Fast Transparent Migration for Virtual Machines. In *Proceedings of the 2005 Annual USENIX Technical Conference* (Apr. 2005).

[11] Nightingale, E. B., Veeraraghavan, K., Chen, P. M., and Flinn, J. Rethink the Sync. In *Proceedings of the 2006 Symposium on Operating Systems Design and Implementation* (Nov. 2002).

[12] Schlicting, R., and Schneider, F. B. Fail-stop Processors: An Approach to Designing Fault-tolerant Computing Systems. *ACM Computing Surveys 1*, 3 (Aug. 1983), 222–238.

[13] Schneider, F. B. Implementing fault-tolerance services using the state machine approach: A tutorial. *ACM Computing Surveys 22*, 4 (Dec. 1990), 299–319.

[14] Stratus Technologies. Benefit from Stratus Continuing Processing Technology: Automatic 99.999% Uptime for Microsoft Windows Server Environments. At http://www.stratus.com/pdf/whitepapers/continuous-processing-for-windows.pdf, June 2009.

[15] Xu, M., Malyugin, V., Sheldon, J., Venkitachalam, G., and Weissman, B. ReTrace: Collecting Execution Traces with Virtual Machine Deterministic Replay. In *Proceedings of the 2007 Workshop on Modeling, Benchmarking, and Simulation* (June 2007).