Hypervisor-based Fault-tolerance*

Thomas C. Bressoud**
ISIS Distributed Systems
111 South Cayuga Street
Ithaca, New York 14850

Fred B. Schneider
Computer Science Department
Cornell University
Ithaca, New York 14853

March 16, 1995

ABSTRACT

Protocols to implement a fault-tolerant computing system are described. These protocols augment the hypervisor of a virtual machine manager to coordinate a primary virtual machine and its backup. The result is a fault-tolerant computing system that does not require modifying the hardware, operating system, or applications programs. A prototype system was constructed for HP's PA-RISC instruction-set architecture. Using this prototype, engineering issues and performance implications of the approach were explored.

* This material is based on work supported in part by the Office of Naval Research under contract N00014-91-J-1219, ARPA/NSF Grant No. CCR-9014363, NASA/ARPA grant NAG-2-893, and AFOSR grant F49620-94-1-0198. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not reflect the views of these agencies.

** Work performed while at Cornell University.

1. Introduction

One popular scheme for implementing fault tolerance involves replicating a computation on processors that fail independently. Replicas are coordinated so that they execute the same sequence of instructions and produce the same results. This paper describes a novel implementation of that scheme. We interpose a software layer between the hardware and the operating system. The result is a fault-tolerant computing system whose implementation does not require modifications to hardware, to the operating system, nor to any applications software.

The benefits of our approach concern engineering and time-to-market costs. We are driven by two observations. First, a manufacturer typically will build a series of realizations for a given instruction-set architecture, where each successive realization has improved cost/performance. Second, implementing replica coordination is subtle, whether done by hardware or software. Given these, we note:

- When replica coordination is implemented in hardware, a design cost is incurred each time a new realization of the architecture is produced. Because designing replica-coordination hardware takes time, support for fault-tolerance necessarily lags behind the cost/performance curve.
- Implementing replica coordination in an operating system requires that the operating system be modified. Mature operating systems are complicated, and this makes it difficult to perform the necessary modifications. In addition, modifications must be devised for every operating system supported by a given platform.
- If replica coordination is left to the application programmer, then the same problems must be solved by the programmers of every application. Moreover, all of these programmers must be acquainted with the nuances of replica coordination.

These difficulties caused us to explore alternatives to the hardware, the operating system, and the application programs as the level for implementing replica coordination in a system.

A *hypervisor* is a software layer that implements *virtual machines* having the same instruction-set architecture as the hardware on which the hypervisor executes. Because the virtual machine's instruction-set architecture is indistinguishable from the bare hardware, software run on a virtual machine cannot tell whether a hypervisor is present. Perhaps the best known hypervisor is CP-67 [MS70], developed by IBM Corp. for 360/67 and later evolved into VM/370 [IBM72] for System 370 mainframes. Hypervisors for other machines have also been constructed [PK75] [K82]. An excellent survey of research on virtual machines appears in [G74].

There are a variety of reasons one might employ a hypervisor. A hypervisor allows multiple operating systems or multiple versions of the same operating system to coexist on a single (hardware) processor. Even when virtual machines all execute the same operating system, a hypervisor provides an isolation that simplifies protection and sharing [PK74][K82]. Our research is not concerned with

the virtues and costs of hypervisors, though. We are concerned with the virtues and costs of augmenting a hypervisor to support replica coordination and, in that manner, support fault-tolerance.

Use of a hypervisor to implement replica coordination is attractive—at least, in theory. When replica coordination is implemented in a hypervisor, it instantly becomes available to all hardware realizations of the given instruction-set architecture, including realizations that did not exist when the hypervisor was written. Second, when replica coordination is implemented in a hypervisor, a single implementation suffices for every operating system that executes on that instruction-set architecture. Finally, by implementing replica coordination in a hypervisor, the applications programmer is freed from this task. The question, then, is whether hypervisor-based replica coordination is practical. What is the performance penalty?

This paper describes the protocols and the performance of a prototype implementation of hypervisor-based fault-tolerance. In §2, we describe the protocols. These protocols ensure that the sequence of instructions executed by two virtual machines running on different physical processors are identical. The protocols also coordinate I/O issued by these virtual machines. Our prototype is discussed in §3. To construct this prototype, we implemented a hypervisor for HP's PA-RISC architecture and augmented that hypervisor with our replica coordination protocols. We report in §4 on the performance of our prototype. In addition to discussing performance measurements, we describe the consequences of variations that might improve performance of the prototype. Finally, §5 discusses related work and some future research directions.

2. Replica Management Protocols

In the primary/backup approach to fault-tolerance [AD76], n processors implement a system that can tolerate $n-1$ faults. One processor is designated the *primary* and the others are designated *backups*. To obtain service, clients make requests of the primary. The primary responds to each request and informs the backups of its actions so that a backup can take over if the primary fails.[1]

Our implementation of fault-tolerant virtual machines uses the primary/backup approach in the hypervisor. A *t-fault-tolerant virtual machine* consists of a primary virtual machine, executed by one processor, and t backups, each executed by other processors. The t-fault-tolerant virtual machine continues operating as long as t or fewer of the processors experience hardware failures. Protocols ensure that

[1]Notice that the primary/backup approach works only when processors exhibit *failstop* behavior—in response to a failure, the primary must halt and do so detectably [SS83]. Arbitrary behavior in response to a failure cannot be tolerated. Today's hardware approximates the failstop model with sufficient fidelity so that the failstop assumption is reasonable unless the system must satisfy the most stringent fault-tolerance requirements. Moreover, a single backup (i.e. $n=2$) usually suffices, because the time to integrate a new backup into the system is typically short and, therefore, the risk of a second failure in that interval is acceptably low.

(1) if the primary's processor has not failed, then backup virtual machines generate no interactions with the environment, and

(2) after the primary's processor has failed, exactly one backup virtual machine generates interactions with the environment and in such a way that the environment is unaware of the primary's failure.

The *environment* for a virtual machine includes the I/O devices accessible to that virtual machine.

Our protocols are for a single backup, so we implement a 1-fault-tolerant virtual machine; generalization to t-fault-tolerant virtual machines is straightforward. The protocols cause the backup virtual machine to execute exactly the same sequence of instructions as the primary virtual machine, where each instruction executed by the backup has the same effect as when it is executed by the primary. The protocols also ensure that the environment does not see an anomalous sequence of I/O requests if the primary fails and the backup takes over while an I/O operation is in progress.

One obvious assumption, so that the backup virtual machine can take over for the primary, concerns the accessibility of I/O devices:

> **I/O Device Accessibility Assumption**: I/O devices accessible to the processor executing the primary virtual machine are also accessible to the processor executing the backup virtual machine.

Other assumptions, discussed below, concern the effects of executing various classes of instructions. We ignore here the problem of replacing the backup after a failure, it being orthogonal to replica-coordination and fairly straightforward.

2.1. Identical Instruction Streams

In our scheme, a given instruction must have the same effect whether it is executed by the primary virtual machine or the backup. This requires two assumptions about instruction execution. Both assumptions can be satisfied by HP's PA-RISC [HP87], DEC's Alpha [S92], and most other modern processors.

Define the *virtual-machine state* to include the memory and registers that change only with execution of instructions by that virtual machine. Main memory, address translation registers, the program counter, and the general-purpose registers are all part of the virtual-machine state, but a time-of-day clock, the interval timer, I/O status registers, and the contents of I/O devices would not be. We partition the instruction set into *ordinary instructions*, whose behavior is completely determined by the virtual-machine state, and *environment instructions*, whose behavior is not. Examples of ordinary instructions include those for arithmetic and data movement; examples of environment instructions include those for reading the time-of-day clock, loading the interval timer, and for performing I/O.

In order that the primary and backup virtual machines execute exactly the same sequence of instructions, both virtual machines are started in the same state. We then require that every instruction have the same effect when it is executed by the primary as when it is executed by the backup. By definition, the effects of executing ordinary instructions depend only on the virtual-machine state. Thus, ordinary instructions can be executed directly by the hardware at the primary and backup provided:

> **Ordinary Instruction Assumption**: Executing the same ordinary instruction on two processors in the same virtual-machine state has exactly the same effect.

Two ADD instructions, for example, must calculate identical sums when given identical arguments.

The second assumption ensures that when executing an environment instruction, the hypervisor at the primary and backup virtual machines have an opportunity to communicate. This allows both hypervisors to change the virtual-machine state in the same way. For example, the second assumption allows an instruction executed by the backup for reading the time-of-day clock to return the same value as returned when that instruction was executed—perhaps at a slightly different time—by the primary.

> **Environment Instruction Assumption**: Environment instructions are simulated by the hypervisor (and not executed directly by the hardware). The simulation ensures that a given environment instruction executed on two processors in the same virtual-machine state has exactly the same effect on the virtual-machine state.

To guarantee that the primary and backup virtual machines execute the same sequence of instructions, we must ensure that identical interrupts are delivered to each and at the same points in their instruction streams. The presence of a hypervisor helps. The primary's hypervisor can forward the I/O interrupts it receives to the backup's hypervisor. And, the primary's hypervisor can send to the backup's hypervisor information about the value of the interval timer at the processor executing the primary virtual machine. Thus, by communicating with the primary's hypervisor, the backup's hypervisor learns what interrupts it must deliver to the backup virtual machine.

However, even careful use of an interval timer cannot ensure that the hypervisor at the primary and backup receive control at exactly the same points in a virtual machine's instruction stream. This is because instruction-execution timing on most modern processors is unpredictable. Yet, interrupts must be delivered at the same points in the primary and backup virtual machine instruction streams. We must employ some other mechanism for transferring control to the hypervisor when a virtual machine reaches a specificed point in its instruction stream.

The *recovery register* on HP's PA-RISC processors is a register that is decremented each time an instruction completes; an interrupt is caused when the recovery register becomes negative. With a recovery register, the hypervisor can run a virtual machine for a fixed number of instructions and then

receive control and deliver any interrupts received and buffered during that *epoch*. A hypervisor that uses the recovery register can thus ensure that epochs at the primary and backup virtual machines each begin and end at exactly the same point in the instruction stream.

A recovery register or some similar mechanism is, therefore, assumed.

> **Instruction-Stream Interrupt Assumption**: A mechanism is available to invoke the hypervisor when a specificed point in the instruction stream is reached.

In addition to the recovery register on HP's PA-RISC, the DEC Alpha performance counters could be adapted, as could counters for any of a variety of events [G94]. Object-code editing [LB92] [GLW95] gives yet another way to ensure that the primary and backup hypervisors are invoked at identical points in a virtual machine's instruction stream. In this scheme, the object code for the kernel and all user processes is edited so that the hypervisor is invoked periodically. Or, one can simply modify the code-generator for a compiler to cause invocation of the hypervisor periodically whenever a program produced by that compiler is executed.

By virtue of the Instruction-Stream Interrupt Assumption, execution of a virtual machine is partitioned into epochs, and corresponding epochs at the primary and the backup virtual machines comprise the same sequences of instructions. We have only to ensure that the same interrupts are delivered at the backup as at the primary when each epoch ends. The solution to this is for the primary and backup hypervisor to communicate, and at the end of an epoch i to have the backup's hypervisor deliver copies of the interrupts that primary's hypervisor delivered at the end of its epoch i.

We now summarize the protocol that ensures the primary and backup virtual machines each performs the same sequence of instructions and receives the same interrupts. To simplify the exposition, we assume that the processors executing the primary and backup are linked by FIFO communications channels. We also assume that the processor executing the backup detects the primary's processor failure only after receiving the last message sent by the primary's hypervisor (as would be the case were timeouts used for failure detection). Counter e_p is maintained by the primary's hypervisor and e_b by the backup's hypervisor to store the number of the epoch currently being executed by the primary and backup virtual machine respectively.

First, we treat the case where a processor running the primary virtual machine has not failed. The protocol is formulated as a collection of hypervisor transitions. We write Tme_p to denote the virtual interval timers and time-of-day clocks at the processor executing the primary virtual machine and write Tme_b for the same registers at the backup virtual machine. Each epoch, Tme_p is sent to the backup's hypervisor so that hypervisor can resynchronize its clocks (denoted here by assignment $Tme_b := Tme_p$) and thus will schedule interval timer interrupts at the end of the same epochs as the primary's hypervisor.

P1: If $e_p = E$ and primary's hypervisor receives an interrupt *Int*:

- primary sends [E, *Int*] to backup
- primary buffers *Int* for later delivery

P2: If $e_p = E$ and the epoch ends at the primary:

- primary sends [Tme_p] to backup
- primary awaits acknowledgment for all messages previously sent to backup
- primary adds to buffer any interrupts based on Tme_p
- primary delivers all interrupts buffered during epoch E
- primary sends [**end**, E] to backup
- $e_p := e_p + 1$
- primary starts epoch $E+1$

P3: If backup's hypervisor receives an interrupt *Int* destined for the backup virtual machine then it ignores *Int*.

P4: If backup's hypervisor receives a message [E, *Int*] from primary:

- backup sends an acknowledgment to the primary
- backup buffers *Int* for delivery at end of epoch E

P5: If $e_b = E$ and the epoch ends at the backup:

- backup awaits [Tme_p] message from primary
- $Tme_b := Tme_p$
- backup awaits [**end**, E] message from primary
- backup adds to buffer interrupts based on Tme_b
- backup delivers all interrupts buffered for delivery at end of epoch E
- $e_b := e_b + 1$
- backup starts epoch $E+1$

Now consider the case where the processor executing the primary virtual machine fails. Suppose the failure occurs after starting epoch E but before the sending (in P2) of [**end**, E] to the backup's hypervisor. Observe that the backup virtual machine will execute the portion of epoch E that the primary executed, since the same interrupts were delivered at the backup when it started epoch E. The backup has no obligations concerning execution after the point in epoch E where the primary fails, so the backup can simply continue executing instructions (and continue ignoring interrupts from the backup processor) until the end of epoch E. However, after the backup virtual machine reaches the end of epoch E, it will not receive the expected [**end**, E] message from the primary's hypervisor. It is at this point that the backup is promoted to the role of the primary:

P6: If $e_b = E$ and the epoch ends at the backup:

- backup awaits detection of failed primary
- backup adds to buffer interrupts based on Tme_b
- backup delivers all interrupts it buffered for delivery at end of epoch E

- $e_b := e_b + 1$
- backup is promoted to primary
- backup starts epoch $E+1$

It is important to understand what P1 through P6 do and do not accomplish. P1 through P6 ensure that the backup virtual machine executes the same sequence of instructions (each having the same effect) as the primary virtual machine. P1 through P6 also ensure that if the primary virtual machine fails, then instructions executed by the backup extend the sequence of instructions executed by the primary. P1 through P6 do not guarantee that interrupts from I/O devices are not lost (nor is the protocol intended to prevent lost I/O interrupts). If the processor executing the primary virtual machine fails before successfully relaying an I/O interrupt that has been delivered to the primary's hypervisor, then that interrupt will be lost. The next subsection extends the protocol to cope with lost I/O interrupts and the more general problem of ensuring that the environment does not see anomalous behavior in response to a failure.

2.2. Interaction with an Environment

The state of the environment is affected by executing I/O instructions. We must ensure that the sequence of I/O instructions seen by the environment is consistent with what could be observed were a single processor in use, even though our 1-fault-tolerant virtual machine is built using two processors. The problem is best divided into two cases:

(i) epochs the primary completes without failing, and

(ii) epochs, called *failover* epochs, during which the primary fails.

These two cases suffice because, according to P6, the backup virtual machine is promoted to a primary for the epoch following the one in which the primary fails. So, every epoch is one that the primary completes or one during which the primary fails.

Case (i) is simple—during these epochs, I/O instructions from the backup virtual machine are suppressed by the backup's hypervisor. Recall, due to P5, the backup virtual machine does not start its epoch E until after the primary has completed that epoch (because the [**end**, E] message is awaited). The backup's hypervisor therefore knows at the start of epoch E that the primary has completed this epoch without failing. The hypervisor at the backup can thus suppress I/O attempted by the backup virtual machine during this epoch.

Case (ii) is problematic because it gives rise to an instance of the unsolvable two generals problem [G79]: No protocol can exist to inform the backup's hypervisor about I/O attempted by the primary virtual machine if the processor executing the primary can fail. This is because sending a message is the only way for the primary's hypervisor to inform the backup's hypervisor that an I/O instruction was issued. In a protocol where the notification is sent after the I/O instruction is issued, the primary's failure after the I/O but before the send would cause the backup to conclude

(erroneously) that the I/O was started; in a protocol where the notification message is sent before the I/O instruction is issued, the primary's failure after the send but before the I/O would cause the backup to conclude (erroneously) that the I/O instruction was never attempted.

To handle case (ii), we exploit the reality that I/O devices are themselves subject to transient failures and drivers already cope with these failures. All I/O devices are assumed to comply with the following interface:

IO1: If an I/O instruction is issued and performed, then the processor issuing the instruction receives a *completion* interrupt.

IO2: If the processor issuing an I/O instruction receives an *uncertain* interrupt, then the I/O may or may not have been performed.

The SCSI bus used with HP's PA-RISC machines and the I/O architecture for DEC's Alpha both satisfy these requirements, as would I/O devices accessed over a communications network. With the SCSI bus protocol, the CHECK_CONDITION command complete interrupt status has the same semantics as the uncertain interrupt of IO2.

An operating system's driver for an I/O device satisfying IO1 and IO2 may have to retry I/O instructions. Specifically, whenever an uncertain interrupt is received, a pending I/O instruction must be repeated. The environment (i.e. I/O device) must therefore tolerate repetition of I/O instructions. And, we exploit this tolerance in handling I/O operations issued by the primary virtual machine for which neither a completion nor an uncertain interrupt has been relayed to the backup's hypervisor prior to the primary's failure.

P7: The backup's hypervisor generates an uncertain interrupt for every I/O operation that is outstanding when the backup virtual machine finishes a failover epoch.

The effect of P7 is to cause certain I/O instructions to be repeated. However, as far as the environment is concerned, this repetition might be a consequence of transient events that caused I/O devices to return uncertain interrupts. The environment, therefore, sees a sequence of I/O instruction that is consistent with what could be observed were a single real processor in use.

3. A Prototype System

In order to evaluate the performance implications of hypervisor-based fault-tolerance, we constructed a prototype. This involved implementing a hypervisor and then augmenting that hypervisor with the protocols of §2. Our prototype consists of two HP 9000/720 processors connected by both a SCSI bus and by an Ethernet. We chose PA-RISC processors so that a recovery register was available to control epochs. A disk connected to the SCSI bus serves as a representative I/O device; a remote console is attached to the Ethernet and is available for control and debugging of the system. See Figure 1.

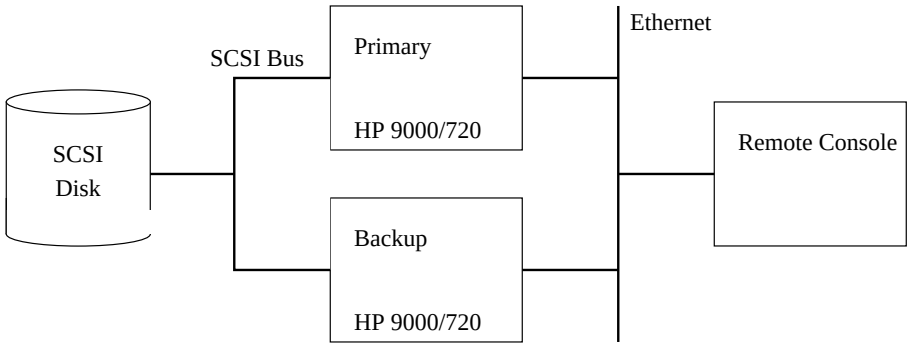

Figure 1. The Prototype

3.1. The Hypervisor

A hypervisor must not only implement virtual machines whose instruction-set architecture is indistinguishable from the bare hardware, but it must do so efficiently—a virtual machine should execute instructions at close to the speed of the hardware. Typically this is accomplished by taking advantage of a dual mode processor architecture, whereby running in *supervisor* mode allows both *privileged* and *non-privileged* instructions to be executed, but running in *user* mode allows only non-privileged instructions to be executed. The hypervisor executes in supervisor mode and receives control on any incoming interrupt or trap. All other software, including the operating system kernel of the virtual machine, executes in user mode. Whenever the virtual machine is in a virtual supervisor mode and attempts to execute a privileged instruction, a privilege trap occurs and the hypervisor simulates the instruction.

Implementing a hypervisor for HP's PA-RISC is not completely straightforward, however. Two aspects of the instruction-set architecture prevent efficient virtualization: the memory architecture and the processor's privilege levels. Fortunately, all of the problems could be addressed by constructing a hypervisor that supports only a single instance of a somewhat restricted virtual machine. That virtual machine suffices for running HP's UNIX system, HP-UX. Our hypervisor is approximately 24K lines of code (of which 5K are assembly language and the rest are C).

Memory Architecture. On HP's PA-RISC architecture, address translation is accomplished using a set of *space registers*. These space registers define logical address segments; instructions that read the registers are non-privileged and a subset of the space registers may even be written to using non-privileged instructions. The hypervisor cannot intercept non-privileged accesses to the space registers. This causes problems if multiple virtual machines are to be supported, because the hypervisor will not be invoked when a virtual machine changes the space registers. It becomes difficult to present each virtual machine with the illusion that it alone is being executed, since a virtual machine may

discover information to the contrary in the space registers.

The problem need not be addressed if the hypervisor only supports a single virtual machine. This, then, is what we do in our prototype. We include the hypervisor in the address space of the virtual machine's kernel. The hypervisor looks to the kernel like another operating system driver. Because there is only a single virtual machine, the hypervisor need not be involved in storage management and changes to the space registers. The only exception is the control of the access rights for the memory-mapped I/O pages, which the hypervisor must still control. This control is obtained by the hypervisor intercepting and changing the access rights for these pages as they are inserted into the TLB.

Processor Privilege Levels. HP's PA-RISC instruction-set architecture defines four privilege levels. Privilege level 0 is equivalent to the supervisor mode described above; levels 1 through 3 are used to differentiate levels of access control and do not permit execution of privileged instructions. The probe, gate, and branch-and-link instructions reveal the current privilege level of the processor. Execution of a branch-and-link instruction, for example, causes the processor's current privilege level to be stored in the low-order bits of the return address. Thus, probe, gate, and banch-and-link, would enable a virual machine to discover that its privilege level is not the same as the current privilege level of the hardware.

We addressed this problem by analyzing the use of privilege levels and the probe, gate, and branch-and-link instructions by HP-UX. On a bare machine, the HP-UX kernel executes at privilege level 0 and all other HP-UX software executes at privilege level 3. Privilege levels 1 and 2 are not used by HP-UX. In our protoype, the hypervisor executes at privilege level 0, virtual privilege level 0 is executed at real privilege level 1, and virtual privilege level 3 is executed at real privilege level 3. This mapping of virtual privilege levels to real privilege levels works only because HP-UX does not use all of the privilege levels. Thus, a number of problems that might be caused by the access control architecture simply do not arise with HP-UX.

To deal with the return addresses from banch-and-link instructions, we checked all uses of this instruction by HP-UX to see if the low-order bits of a return address were actually used. In the assembly-language portion of the HP-UX kernel, we found a single instance during the boot sequence where the branch and link instruction was being used as a load-position independent way of determining the current physical address. This code assumes that the low-order bits were 0 (supervisor mode), since this code always runs in supervisor mode. A solution (hack) was to modify this code fragment and mask out the privilege bits of the return address. For the rest of HP-UX, which is written in C and other high-level languages, we observed that the procedure linkage routine generated by the high level language compilers was not sensitive to the execution mode bits in the return address. Thus, HP-UX never detects the presence of our hypervisor, although if it looked, it could.

3.2. Replica-coordination in the Hypervisor

To augment our hypervisor with the replica-coordination protocols, we investigated whether the various assumptions given in §2 could be satisfied.

The I/O Device Accessibility Assumption is easy to satisfy because multiple hosts may reside on the same SCSI bus. Once bus termination considerations are resolved, the primary and backup machines can be chained together on a single SCSI bus, allowing both to access the disk. This is what we do.

We (as well as a number of HP engineers) were surprised to find that the Ordinary Instruction Assumption does not hold for the HP 9000/720 processor. In the HP PA-RISC architecture, TLB misses are handled by software. When the translation for a referenced location is not present in the TLB, a TLB miss trap occurs. If the reference is for a page already in memory, then the required information is read from the page table and the entry is inserted into the TLB. If, on the other hand, the reference is for a page that is not in memory, then the page must be retrieved from secondary storage; the TLB is updated (by software) once the transfer is complete.

The TLB replacement policy on our HP 9000/720 processors was non-deterministic. An identical series of location-references and TLB-insert operations at the processors running the primary and backup virtual machines can lead to different TLB contents. Since TLB miss traps are handled by software, differences in TLB contents become visible when a TLB miss trap occurs at one of the virtual machines and not at the other.

Our solution to this problem was to have the hypervisor take over some of the TLB management. The hypervisor intercepts TLB miss traps, performs the page table search and, if the page is already in memory, does the TLB insert operation. Only for pages that are not already in memory does the virtual machine software receive a TLB miss trap. Thus, it appears to the virtual machine as if the hardware were responsible for loading TLB entries for pages that are in memory. Strictly speaking, our hypervisor implements a virtual machine that is different from the PA-RISC instruction-set architecture. But the difference is one that does not affect HP-UX. (However, an HP-UX release with a bug in its TLB miss handler could be affected, because the bug might never be encountered when run in a virtual machine but might be when run on the raw hardware.)

The Environment Instruction Assumption concerns instructions that cause I/O. HP's PA-RISC instruction-set architecture has memory-mapped I/O. I/O controller registers are accessed through ordinary load and store instructions. To satisfy the Environment Instruction Assumption, our hypervisor alters the access protection for the memory pages associated with these I/O controller registers so that a load or store attempted by the virtual machine (executing in user mode) causes an access trap to occur. The access trap causes control to be transferred to the hypervisor.

Finally, the Instruction-Stream Interrupt Assumption is handled by using the recovery counter of the HP PA-RISC.

4. Performance of the Prototype

Performance measurements of our prototype give insight into the practicality of the protocols. We also formulated (and validated) mathematical models for hypervisor-based fault-tolerance, to better understand the affects of various system parameters. Normalized performance was identified as the figure of merit. A workload that requires N seconds on bare hardware and N' seconds on the prototype has a *normalized performance* of N'/N. Thus, a normalized performance of 1.25 for a given workload indicates that under the prototype 25% is added to the completion time. We desire a normalized performance that is as small as possible; a normalized performance of 1 is the best we might expect.

Epoch length was our paramount concern. With short epochs, interrupts are not significantly delayed by hypervisor buffering but the number of epochs for a given task—and the associated overhead—is increased. With long epochs, fewer epochs transpire but hypervisor delays for interrupt delivery may become significant.

4.1. CPU-Intensive Workload

Our first investigations concerned a CPU-intensive workload. The dominant process executed 1 million iterations of the Dhrystone 2.1 benchmark. This process was assigned the highest possible real-time priority; epoch length was set at 4K instructions. The experiment was repeated 20 times. The coefficient of variation for the parameters measured was less than .0012%, giving us confidence in the validity of using their average values.

The normalized performance for this CPU-intensive workload with 4K epochs was (an abysmal) 6.50—the overhead was very high. An average of 15.12 μsec. was required for the hypervisor to simulate each privileged instruction; approximately 8 μsec. for hypervisor entry/exit and 7 μsec. for the actual work.[2] Epoch-boundary processing (i.e. rule P2) consumed an average of 442.59 μsec. This meant that epoch boundaries added approximately 46 seconds to the the 8.8 sec. required for executing the benchmark of 4.2×10^8 instructions.

By increasing the epoch length, we reduce the total amount of time devoted to epoch-boundary processing. The normalized performance $NP_C(EL)$ for the CPU-intensive workload as a function of epoch-length EL can be approximated by the following:

[2]The HP 9000/720 on which these experiments were performed is a 50 MIPS processor, so a typical instruction should execute in .02 μsec.

$$NP_C(EL): \quad 1 \; + \; \frac{1}{RT} \left(n_{sim} \, h_{sim} \; + \; \frac{VI}{EL} \, h_{epoch} \; + \; C_{other}(EL) \right)$$

where

RT:	real time required to execute workload on bare hardware (8.8 sec.)
n_{sim}:	number of workload's instructions simulated by hypervisor
h_{sim}:	average time for hypervisor to simulate an instruction (15.12 μsec.)
VI:	number of virtual machine instructions executed for workload (4.2×10^8)
h_{epoch}:	average epoch-boundary processing time (443.59 μsec.)
C_{other}:	delays caused communication between the primary and backup hypervisors (41 msec. was measured)

A graph of $NP_C(EL)$ for epoch length EL between 1K and 32K instructions appears as Figure 2. Also indicated on that graph are measurements for our prototype with epoch lengths 1K, 2K, 4K, and 8K. The measurements agree with what the equation predicts, validating $NP_C(EL)$ for predicting performance of this workload.

The graph of Figure 2 shows that normalized performance improves as epoch length increases. When there are 32K instructions in an epoch, a normalized performance of 1.84 is predicted—a substantial improvement from the normalized performance of 6.50 that we observed for 4K epochs. However, long epochs cause interrupt delivery to be delayed. The impact of this delay gives a practical upper-bound for epoch length. HP-UX, for example, requires that epoch lengths not exceed 385,000 instructions, because of the way the clock is maintained by the kernel. For epoch lengths of 385,000 instructions, our model predicts a normalized performance of 1.24 for the CPU-intensive workload. This performance would be quite acceptable, especially since the hypervisor's simulation of instructions accounts for .18 or the .24 overhead.[3] For long epochs, then, our replica-management scheme is responsible for adding only 6% overhead.

4.2. Input/Output Workloads

We would expect a workload in which I/O occurs to perform differently than the CPU-intensive workload discussed above. First, I/O involves a significantly higher proportion of instructions that must be simulated by the hypervisor. Second, there is the added overhead for transferring the result of a disk read from the primary's hypervisor to the backup's hypervisor. The primary virtual machine may not proceed until this data has been received by the backup's hypervisor (see rule P2 of the protocol).

---

[3]Anecdotal evidence [C95] for a mature VM/370 installation places normalized performance at around 1.40. The significantly higher cost for VM/370 is undoubtedly due to supporting multiple virtual machines as well as differences in the workload.

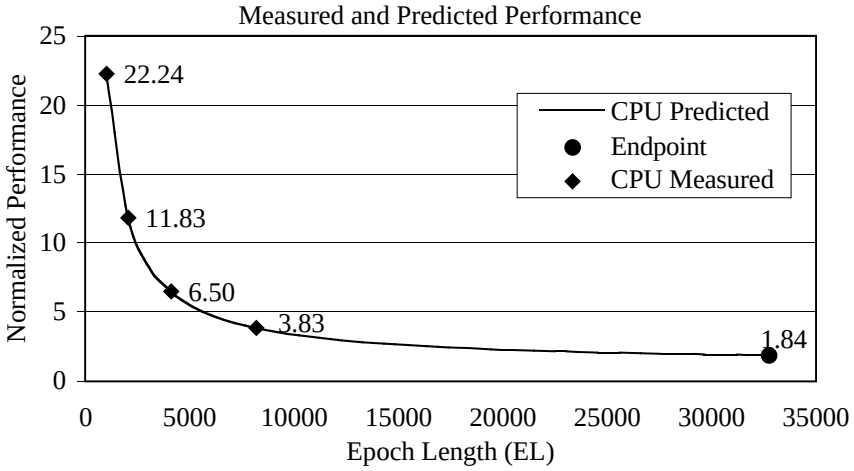

Figure 2. CPU-Intensive Workload

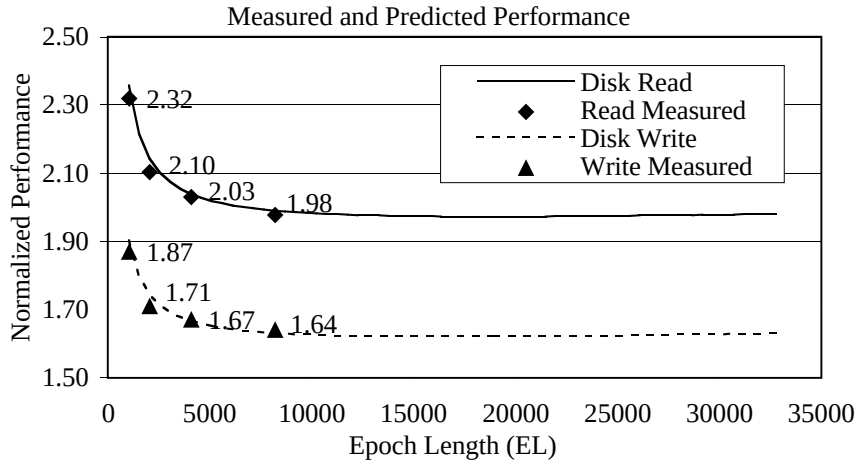

Figure 3. Input/Output Workloads

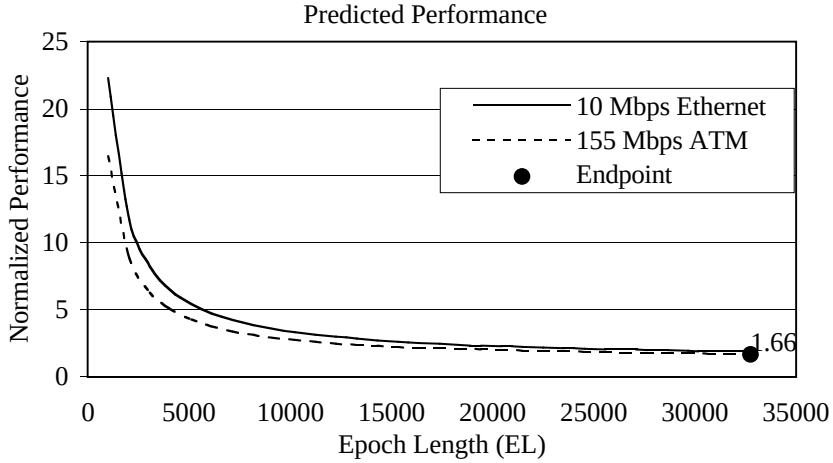

Figure 4. Faster Communication

When analyzing an I/O-intensive workload involving a disk, care must be taken to ensure that requests actually propogate beyond any buffer pools that an operating system, like HP-UX, might maintain. For reads, we must also be careful that performance measurements are unaffected by disk-block prefetches; for writes, we must prevent overlapping the data transfer with subsequent computation. This leads to the following I/O benchmarks. A large file is pre-allocated on the disk. Then, for measuring the performance of reads, the benchmark randomly selects a disk block, issues a read, and awaits the data. This is iterated 2048 times. The benchmark for writes is analagous—a disk block is randomly selected, a write is issued, and then the write completion is awaited. Notice that a rather high percentage of the instructions concern I/O. These instructions will be privileged and therefore must be simulated by the hypervisor.

We ran 20 experiments in which the write version of the I/O benchmark was executed. The coefficient of variation for the parameters measured was less than .25% giving us confidence in the validity of using averages. The normalized performance with 4K epochs was found to be 1.67. This normalized peformance includes the impact of the hypervisor on the the block selection calculation and memory-mapped I/O loads and stores to initiate the write. Therefore, we also measured the disk write times with and without the hypervisor present. When the benchmark is executed on bare hardware, a disk write takes an average of 26 msec. to complete; when the hypervisor and replica-coordination protocols are present, a disk write takes an average of 27.8 msec. Thus, disk write performance does not really suffer when epochs are length 4K. However, as we shall see, with significantly larger epochs, interrupt delivery is delayed and disk write performance can suffer.

To measure the performance of disk reads, the read version of our I/O benchmark was used. This experiment was also repeated 20 times. The coefficient of variation for the parameters measured was less than 3%, giving us confidence in the validity of using their averages.[4] The benchmark is not completely successful in selecting disk blocks not in the buffer-pool—of the 2048 read requests issued, only on average 1729 caused actual disk reads. We computed a normalized performance for the experiments (with 4K epochs) of 2.03. Because processing a read request requires the primary's hypervisor to forward a copy of the data read to the backup, disk reads are expected to take significantly longer with our replica-coordination protocols in place. When the benchmark is executed on bare hardware, an 8K disk block read takes an average of 24.2 msec. to complete; when the hypervisor and replica-coordination protocols are present, a disk read takes an average of 33.4 msec. A 10Mbps Ethernet is used in transferring the disk block from the primary to the backup; this requires 9 messages for the data and 1 message for an acknowlegement.

[4]This coefficient of variation is much larger than obtained with the other workloads because of the high variance associated with processing interrupts for communications between the primary and backup hypervisor. The other workloads involve considerably less communication.

Normalized performance $NP_W(EL)$ for the write version of the I/O benchmark can be approximated by:

$$NP_W(EL): \quad \frac{n_W(cpu(EL)+xfer_W+delay_W(EL))}{RT}$$

where

RT:	real time required to execute workload on bare hardware
n_W:	number of writes (2048 for the benchmark)
$cpu(EL)$:	elapsed time required to select a disk block and initiate the transfer of a disk block when the hypervisor is present and EL is the epoch length.
$xfer_W$:	elapsed time between initation of disk write the receipt of the corresponding interrupt (26 msec.)
$delay_W(EL)$:	elapsed time between the completion interrupt and its delivery to the virtual machine when the epoch length is EL

And, normalized performance $NP_R(EL)$ for the read version of the I/O benchmark can be approximated by:

$$NP_R(EL): \quad \frac{n_R(cpu(EL)+xfer_R+delay_R(EL))}{RT}$$

where

RT:	real time required to execute workload on bare hardware
n_R:	number of reads (1729 for the benchmark)
$cpu(EL)$:	elapsed time required to select a disk block and initiate the transfer of a disk block when the hypervisor is present and EL is the epoch length.
$xfer_R$:	elapsed time between initation of disk read the receipt of the corresponding interrupt (24.2 msec.)
$delay_R(EL)$:	elapsed time between the completion interrupt and its delivery to the virtual machine when the epoch length is EL

A graph of $NP_W(EL)$ and $NP_R(EL)$ for epoch length EL between 1K and 32K instructions appears as Figure 3. Measurements for our prototype when executed with epoch lengths 1K, 2K, 4K, and 8K are also marked on the graph. The measurements are each within 1.9% of what is predicted by $NP_W(EL)$ and $NP_R(EL)$.

As with the CPU-intensive workload, longer epochs lead to better normalized performance. This is because, in our models, the $cpu(EL)$ term dominates. But another trend is also visible. Increases to epoch length EL causes $delay_W(EL)$ and $delay_R(EL)$ to increase, because interrupts from the disk are buffered by the hypervisor for a longer period. This explains the slight upward drift of normalized performance for larger epoch lengths. In a benchmark where more computation were done before each I/O operation, the dominance of the $cpu(EL)$ term would ameliorate the normalized performance. Normalized performance for the I/O workload experiments never goes as low as for the

CPU-intensive workload, because of the high precentage of hypervisor-simulated instructions in doing I/O.

4.3. Faster Replica-Coordination

The predominant overhead for the replica-coordination procotols comes from rule P2, where the primary's hypervisor must await acknowledgments for all messages previously sent to the backup's hypervisor. This suggests that speeding-up the communication between the primary and backup processors might improve performance. Figure 4 is a graph of normalized performance if a 155Mbps ATM link is used for communication in place of the 10Mbps Ethernet. There is some improvement—for epoch length 32K, normalized performance for the Ethernet is predicted to be 1.84 and normalized performance for the ATM link is predicted to be 1.66. (This assumes I/O controller set-up time is the same for both technologies.)

A second improvement results from appreciating that it is not strictly necessary for the primary's hypervisor to await the acknowledgments, as required in rule P2. Suppose (i) a message sent by the primary's hypervisor is not delivered to the backup's hypervisor and (ii) the primary passes through P2 nevertheless (i.e. without waiting). Thus, it becomes possible that an interrupt *Int* has been delivered to the primary virtual machine but, if there is a lost message, might never be delivered to the backup virtual machine. The computation at the backup virtual machine, therefore, might diverge from the primary.

Obviously, no problem occurs unless the processor executing the primary virtual machine fails. In fact, no problem occurs even if the processor fails—provided the primary virtual machine has not revealed to the environment that *Int* was processed. If the delivery of *Int* is not revealed to the environment, then subsequent actions by the backup virtual machine—whatever they may be—are consistent with with what could be observed were there a single processor. Thus, it suffices that the acknowledgments formerly awaited in P2 be received prior to I/O operations by the primary virtual machine, since I/O operations are the only way in which the state of a virtual machine is revealed to the environment.

The modifications to the protocol of §2 are straightforward. First, in P2, the primary's hypervisor need no longer await acknowledgments for messages it sent to the backup's hypervisor. Second, in order to initiate an I/O operation, the primary's hypervisor is required to have received acknowledgements for all messages it has sent to the backup's hypervisor. We performed these modifications to the prototype and re-ran our experiments for the CPU-Instensive workload of §4.1 and the two Input/Output workloads of §4.2. As before, the normalized performance is an average obtained for 20 runs. The results are given in Table 1. The column labeled "Old" refers to the original protocol and "New" refers to the modified protocol.

Epoch Len	Workload					
	CPU Intense		Write Intense		Read Intense	
	Old	New	Old	New	Old	New
1K	22.24	11.67	1.87	1.70	2.32	1.92
2K	11.83	4.49	1.71	1.66	2.10	1.76
4K	6.50	3.21	1.67	1.66	2.03	1.72
8K	3.83	2.20	1.64	1.64	1.98	1.70

Table 1. Normalized Performance of Original and Revised Protocol

As expected, the normalized performance improves signifcantly when acknowledgements need not be awaited in P2. The effect is most pronounced in the CPU-intensive workload, because its normalized performance is most affected by the delay at epoch boundries. In the I/O intensive workloads, some of the delay at an epoch boundary is simply displaced to the I/O operation in each iteration of the benchmark.

5. Discussion

The availability of off-the-shelf microprocessors has allowed fault-tolerant computing systems to be constructed simply by adding support for replica-coordination to a bus or to systems software. In some systems, like the one offered by Stratus, the same inputs are presented by the bus to the replicas and the bus is driven by only a single replica (even though all replicas generate the same outputs) [SS92]. Other systems exploit a bus to implement fault-tolerant processes on top of an operating system. The work described in [BBG83] [BBGHO89], in [MPN92], and in [PP83] exemplify this approach. In the pioneering work of Tandem [B81], the applications themselves are responsible for ensuring coordination between the processes comprising a process-pair, the unit of replication there.

Despite the engineering and time-to-market costs, manufacturers do design and sell processors that implement replica-coordination in hardware. A design from Tandem [CMJ88] and DEC's VAXft 3000 are examples. See [SS92] for a survey of hardware-implemented fault-tolerant computing systems.

The system described in this paper is the first to implement replica-coordination above the hardware but below the operating system. We augment a hypervisor to support replica-coordination. Adding a hypervisor does have a non-trivial performance impact but, as we have shown, the additional cost of our replica-coordination protocols is not significant. And, replica-coordination becomes available on all realizations of the instruction-set architecture and for all operating systems implemented for the instruction-set architecture. Thus, our implementation of fault-tolerance is transparent to operating systems (and their applications) as well as to hardware designers. It is difficult to

compare performance costs with the cost reductions in hardware and software design, so we do not attempt a comparison. A new (faster) processor realization can be exploited immediately; a new operating system is made fault-tolerant trivially.

Augmenting a hypervisor is not the only way to support replica-coordination above the hardware but below system software. One might modify a micro-kernel, for example, and realize many of the same benefits as enjoyed when a hypervisor is augmented. This alternative remains to be investigated. Another question we have not dealt with concerns shared memory. One might imagine virtual processors that communicate using shared memory. For some memory models, this is not difficult to support and it too is the subject of on-going work.

Acknowledgments. The idea and original protocols for implementing fault-tolerance were developed by Schneider while consulting on a project at Digital Equipment Corporation. The other members of that project—Ed Balkovich and Dave Thiel—provided helpful feedback and support throughout. Input from Butler Lampson, also a collaborator, was also quite useful. Ed Balkovich encouraged the construction of this prototype and helped us formulate a plan to evaluate its performance. Cornell's Anne Gockel went beyond the call of duty in keeping alive facilities so that we could run experiments. We would also like to thank Robert Cooper for comments on an early draft of this paper.

References

[AD76] Alsberg, P.A. and J.D. Day. A principle for resilient sharing of distributed resources. *Proc. Second International Conference on Software Engineering* (San Francisco, Calif., Oct 1976), 627-644.

[B81] Bartlett, J.F. A nonstop kernel. *Proc. Eighth Symposium on Operating Systems Principles* (Asilomar, Calif., Dec. 1981), 22-29.

[BBG83] Borg, A., J. Baumbach, and S. Glazer. A message system for supporting fault tolerance. *Proc. Ninth Symposium on Operating Systems Principles* (Bretton Woods, New Hampshire, Oct. 1983), 90-99.

[BBGHO89] Borg, A., W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault tolerance under UNIX. *ACM Transactions on Computer Systems* 3,1 (Feb 1985), 63-75.

[CMJ88] Cutts, Richard W., Nikhil A. Mehta, and Douglas E. Jewett. Multiple processor system having shared memory with private-write capability. U.S. Patent 4,965,717. Oct. 1990.

[C95] Coweles, R. Private communication.

[G74] Goldberg, Robert P. Survey of virtual machine research. *Computer Magazine* (June 1974), 34-45.

[G79] Gray, J. Notes on Data Base Operating Systems. *Operating Systems: An Advanced Course, Lecture Notes in Computer Science*, Vol. 60, Springer-Verlag, New York, 1978, 393-481.

[G94] Gleeson, B. Fault tolerant computer system with provision for handling external events. U.S. Patent 5,363,503. Nov. 1994.

[GLW95] Graham, Susan L., Steven Lucco, and Robert Wahbe. Adaptable binary programs. *Proc. 1995 Usenix Winter Conference* (New Orleans, Lousiana, Jan. 1995), 315-325.

[HP87] *Precision Architecture and Instruction Reference Manual*. Part Number 09740-90014. Hewlett Packard Corporation, Cupertino, Calif., June 1987.

[IBM72] *IBM Virtual Machine Facility/370 Planning Guide* Publication No. GC20-1801-0. IBM Corporation, White Plains, New York.

[K82] Karger, Paul A. Preliminary design of a VAX-11 virtual machine monitor security kernel. DEC TR-126. Digital Equipment Corporation, Hudson, Mass., Jan. 1982.

[LB92] Larus, James R. and Thomas Ball. Rewriting executable files to measure program behavior. Technical Report 1083, Univ. of Wisconsin, Madison, Wisc., 1992.

[MPN90] Major, Drew, Kyle Powell, and Dale Nelbaur. Fault tolerant computer system. U.S. Patent 5,157,663. Oct. 1992.

[MS70] Meyer, P.A. and L.H. Seawright. A virtual machine time-sharing system. *IBM Systems Journal* 9,3 (1970), 199-218.

[PK74] Popek, G.J. and C. Kline. Verifiable secure operating system software. *AFIPS Conference Proceedings*, 1974.

[PK75] Popek, G.J. and C. Kline. The PDP-11 virtual machine architecture: A case study. *Proc. Fifth Symposium on Operating Systems Principles* (Austin, Texas, Nov. 1975), 97-105.

[PP83] Powell, M.L. and D.L. Presotto. Publishing: A reliable broadcast communication mechanism. *Proc. Ninth Symposium on Operating Systems Principles* (Bretton Woods, New Hampshire, Oct. 1983), 100-109.

[SS83] Schlichting, R. and F.B. Schneider. Failstop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems* 1,3 (Aug. 1983), 222-238.

[S92] Sites, Richard. Alpha Architecture Reference Manual. Digital Press, Bedford, Mass., 1992.

[SS92] Siewiorek, D.P. and Robert S. Swarz. *Reliable Computer System Design and Evaluation*. Digital Press, Bedford, Mass. 1992.