Hypervisor-Based Fault-Tolerance

THOMAS C. BRESSOUD
Isis Distributed Systems
and
FRED B. SCHNEIDER
Cornell University

Protocols to implement a fault-tolerant computing system are described. These protocols augment the hypervisor of a virtual-machine manager and coordinate a primary virtual machine with its backup. No modifications to the hardware, operating system, or application programs are required. A prototype system was constructed for HP's PA-RISC instruction-set architecture. Even though the prototype was not carefully tuned, it ran programs about a factor of 2 slower than a bare machine would.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*network operating systems*; D.4.5 [**Operating Systems**]: Reliability—*checkpoint/restart*; *fault tolerance*

General Terms: Algorithms, Reliability

Additional Key Words and Phrases: Fault-tolerant computing system, primary/backup approach, virtual-machine manager

1. INTRODUCTION

One popular scheme for implementing fault tolerance involves replicating a computation on processors that fail independently. Replicas are coordinated so that they perform the same sequence of state transitions and, therefore, produce the same results. This article 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 the hardware, operating system, or any application software.

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 article are those of the author and do not reflect the views of these agencies.
Authors' addresses: T. C. Bressoud, Isis Distributed Systems, 55 Fairbanks Boulevard, Marlborough, MA 01752; email: bressoud@isis.com; F. B. Schneider, Computer Science Department, Cornell University, Ithaca, NY 14853; email: fbs@cs.cornell.edu.

Our approach tolerates those processor failures that can be detected before the faulty processor performs an erroneous, externally visible action—so-called *failstop* behavior [Schlichting and Schneider 1983]. In response to a processor failure, outstanding I/O operations might be reissued, but this is done in a way that should not affect I/O devices like disks or networks. To the software running above our new layer, processor failures are transformed into transient I/O device failures, which this software can be expected to handle.

The benefits of our approach concern engineering costs versus time-to-market costs. We are driven by two observations. First, for a given instruction-set architecture, a manufacturer typically will build a series of realizations, where cost/performance improves over the series. Second, implementing replica coordination is subtle, whether done by hardware or software. The consequences are:

(1) When replica coordination is implemented in hardware, a design cost is incurred for each new realization of the architecture. Because designing replica-coordination hardware takes time, this approach to fault tolerance results in systems that necessarily lag behind the hardware cost/performance curve.
(2) Adding replica coordination to an existing operating system involves identifying state transitions implemented by the operating system, because these are what must be coordinated. Mature operating systems are invariably complicated, so identifying the state transitions and making modifications to coordinate them is difficult. In addition, the effort must be repeated for every operating system supported by a given instruction-set architecture.
(3) If replica coordination is left to application programmers, then these programmers must be acquainted with the nuances of replica coordination. Or, the programmers must be constrained to use a given interface (e.g., causal group broadcasts [Birman 1993]) or abstraction (e.g., transactions [Bernstein et al. 1987]).

These problems drove us to explore alternatives to the hardware, the operating system, and the application programs as the place for implementing replica coordination in a computing 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 determine whether a hypervisor is present. Perhaps the best-known hypervisor is CP-67 [Meyer and Seawright 1970], developed by IBM Corp. for 360/67 and later evolved into VM/370 [IBM 1972] for System 370 mainframes. Hypervisors for other machines have also been constructed [Karger 1982; Popek and Kline 1975]. An excellent survey on virtual machines appears in Goldberg [1974].

Problems (1) through (3) above can be addressed by using a hypervisor to implement replica coordination. Replica coordination implemented in a hypervisor becomes available to all hardware realizations of the given instruction-set architecture, including realizations that do not exist when the hypervisor was built. This handles problem (1) above. For problem (2), we observe that implementing replica coordination in a hypervisor means that a single implementation will suffice for every operating system that executes on that instruction-set architecture. Finally, problem (3) is avoided because implementing replica coordination in a hypervisor frees the application programmer from this task without imposing a set of primitives or abstractions that would constrain how the application program is structured.

The question, then, is whether hypervisor-based replica coordination is practical. What is the performance penalty? This article addresses these issues by describing the protocols[1] and the performance of a prototype implementation of hypervisor-based fault-tolerance. The prototype executes programs about a factor of 2 slower than a bare machine would.

The rest of this article is organized as follows. In Section 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 operations from these virtual machines. Our prototype is discussed in Section 3. To construct this prototype, we implemented a hypervisor for a subset of HP's PA-RISC architecture [Hewlett Packard 1987]; the subset was sufficient for executing HP-UX. We then augmented the hypervisor with our replica-coordination protocols. We report in Section 4 on our prototype's performance. In addition to discussing performance measurements, we consider variations that might improve performance. Section 5 discusses related work; a summary and future research directions are given in Section 6.

2. REPLICA-COORDINATION PROTOCOLS

The theory of managing replicated, deterministic state machines is well understood. It is called the *state machine approach* in Lamport [1978] and Schneider [1990]. A *deterministic state machine* M reads a sequence of *commands*, where each command causes a state transition that is completely determined by the command and the current state of M. State transitions may produce outputs. In replicating M, we ensure that each replica M_i is started in the same state and reads identical sequences of commands. Because M is deterministic, each replica M_i then performs the same sequence of state transitions and produces the same sequence of outputs.

Whereas M produced a single sequence of outputs, each replica M_i now produces a sequence of outputs. Therefore, the ensemble of replicas produces a set of output sequences. Replica failures are masked by employing a mechanism to combine the multiple output sequences into a single sequence

[1]The protocols are the subject of pending U.S. and foreign patents. Contact the second author for information.

that appears to have come from a nonfaulty state machine replica. For example, if faulty replicas produce arbitrary outputs, then a voter can serve as the mechanism for combining output sequences. An ensemble of $2t + 1$ replicas feeding a voter would mask as many as t faulty replicas.

When using the state machine approach in practice, we must identify a deterministic state machine to replicate and then implement mechanisms that ensure (i) each replica reads the same sequence of commands and (ii) the environment obtains a single output (rather than one output per replica). The state machine we replicate for our fault-tolerant computing system is an engine for processing machine-language instructions and interrupts—a virtual machine. Commands to the state machine are virtual-machine instructions to execute and virtual-machine interrupts (accompanied by their DMA data, if any) to deliver. The mechanism for ensuring that all replicas receive the same sequence of commands is built into the hypervisor, as is the mechanism to combine outputs from the ensemble.

The transitions for our state machine are virtual-processor state changes that result from executing a virtual-machine instruction or delivering a virtual-machine interrupt (with accompanying DMA data). Recall, though, the state machine must be deterministic. This forces different commands to be implemented in different ways.

Some commands are completely determined by the previous command processed by the state machine. In particular, each virtual-machine instruction updates the value of the virtual processor's program counter, thereby requesting execution of its successor. Among these commands are the following:

— A command requesting that a deterministic virtual-machine instruction be executed is implemented in our state machine directly by the processor hardware (simply by executing the specified instruction).

— A command requesting that a nondeterministic virtual-machine instruction be executed is implemented in our state machine by the hypervisor. An instruction to read the time-of-day clock is an example of such a command. For such commands, the hypervisor ensures that nondeterministic choices are resolved identically in all replicas. This is accomplished by executing the requested instruction at one replica and then having the hypervisor at that replica disseminate the instruction's outcome to all other hypervisors, so they can simulate the instruction.

Other commands are not completely determined by processing the previous command. Virtual-machine interrupts fall into this category. When such an interrupt is delivered, the virtual-processor state is changed. In terms of state machine commands, this means that the exact placement of a command corresponding to a virtual-machine interrupt delivery is not uniquely determined by the commands that have already been processed.

— A command resulting from delivery of a virtual-machine interrupt is implemented by exploiting the hypervisor. Hypervisors at replicas delay and coordinate virtual-machine interrupt delivery to ensure that the

interrupt and any accompanying DMA data are processed at the same point relative to other commands (i.e., at the same point in the instruction stream) at every replica.

Fundamental to our design is communication among the hypervisors. This implies that the hypervisors must not be partitioned by communications failures. Second, after a hypervisor resolves a nondeterministic choice, that hypervisor must communicate the choice to the others. If messages can be lost, then the sending hypervisor must wait for an acknowledgment before proceeding—a round-trip communications delay that cannot be eliminated.

The actual implementation of our scheme resembles the primary/backup approach to fault tolerance [Alsberg and Day 1976]. One replica is designated the *primary*, and the others are designated as *backups*. All nondeterministic choices are made by the hypervisor at the primary, and all interactions with the environment are through the primary.[2] If the primary fails, then some backup assumes its responsibilities.

A consequence of this primary/backup architecture is that all replicas must have access to the system's environment.

I / O Accessibility Assumption: I/O operations possible by the processor executing the primary virtual machine are also possible by the processor executing a backup virtual machine.

Were this assumption not satisfied, then the failure of a replica could cause parts of the environment to become inaccessible. Such failures could not be masked. Notice that when the environment is a network, the I/O Accessibility Assumption can be satisfied by having all processors connected to the network and using suitable addressing protocols.

2.1 Identical Command Sequences

We now turn to the protocols for ensuring that all state machine replicas read identical sequences of commands. Since a state machine replica corresponds to a virtual machine and since state machine commands are virtual-machine instructions to execute and virtual-machine interrupts to deliver, the protocols ensure that each virtual machine executes the same sequence of virtual-machine instructions.

Define an *ordinary instruction* as one whose behavior is completely determined by the initial system state and sequence of instructions that precede its execution on the processor. In contrast, an *environment instruction* is one whose behavior is not so determined and, therefore, is nondeterministic. Examples of ordinary instructions include those for arithmetic and data

[2]The primary/backup approach works only when processors exhibit failstop behavior so that, in response to a failure, the primary halts and does so detectably. Arbitrary behavior in response to a failure is not tolerated. By using timeouts, today's hardware can appear to approximate the failstop model with sufficient fidelity so that it is reasonable to make this assumption unless the system must satisfy the most-stringent fault-tolerance requirements. Moreover, a single backup usually suffices.

movement as well as those for setting the time-of-day clock and loading the interval timer; examples of environment instructions include those for reading the time-of-day clock as well as for reading a disk block.

Some instructions cause subsequent delivery of an interrupt. Ignore the nondeterminacy in interrupt delivery time when classifying whether or not an instruction is an environment instruction. To be labeled an environment instruction, the data returned with the interrupt must be nondeterministic. This means that an instruction to load the interval timer is not considered an environment instruction. An I/O instruction that causes an interrupt when a DMA transfer from disk completes is considered an environment instruction, because the DMA data accompanying the interrupt is not necessarily determined by the initial system state or the preceding instructions (on this processor).

By definition, ordinary instructions are deterministic. Consequently, they can be executed directly by the hardware. Two ADD instructions on different processors, for example, will calculate the same sums when given identical arguments. And, two identical DIV (divide) instructions having a divisor of 0 are both expected to cause the same trap.[3]

Environment instructions, by definition, are not deterministic. Therefore, the hypervisor becomes involved in executing these instructions. The following assumption asserts that the hypervisor is given that opportunity.

> **Environment Instruction Assumption:** The hypervisor is invoked whenever an attempt is made to execute an environment instruction. The environment instruction is then simulated by the hypervisor. The simulation ensures that the environment instruction executed by distinct state machine replicas has exactly the same effect.

For example, the Environment Instruction Assumption ensures that an instruction executed by the backup for reading the virtual processor's time-of-day clock will return the same value as returned when the corresponding instruction was executed—perhaps at a slightly different time—by the primary.

We must ensure that commands for virtual-machine interrupt delivery appear at the same point and accompanied by the same data in the sequences of commands read by state machine replicas. This requirement involves two tasks. One is to ensure that the same command (and accompanying DMA data) appears someplace in the sequence of commands at each replica. The second is to ensure that commands for interrupt delivery are ordered in the same way relative to each other and to other commands.

We use the hypervisors to ensure that the same command (and its accompanying data) that corresponds to a virtual-machine interrupt delivery is

[3]Imprecise trap delivery is handled by the hypervisor in the manner described below for delivering interrupts. When a trap delivery is guaranteed to occur at a fixed point in the instruction stream relative to the offending instruction, then the trap can be delivered without hypervisor intervention.

read by every state machine replica. Only virtual-machine interrupts raised at the primary become commands; interrupts at backups are ignored. This is acceptable, because backups execute the same sequence of virtual-machine instructions as the primary, so copies of the primary's interrupts suffice. The primary's hypervisor buffers and forwards all virtual-machine interrupts it receives to the backup hypervisors.

Ensuring identical ordering for commands corresponding to virtual-machine interrupts is a bit subtle. One simple solution is to insert these commands at predetermined points in the command sequence that is read by each state machine replica—and that is what we will do. 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 the instruction execution timing on most modern processors is unpredictable. We must employ some other mechanism for transferring control to the hypervisor when a virtual machine reaches a predetermined 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, a hypervisor can run a virtual machine for an *epoch* comprising a fixed number of virtual-machine instructions and then receive control. Upon receiving control, it can deliver any virtual-machine interrupts and accompanying data that it received and buffered during the 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 virtual-machine instruction stream. Epoch boundaries then serve as the predetermined points in the command sequence where virtual-machine interrupt delivery commands are placed.

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

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

In addition to the recovery register on HP's PA-RISC, the DEC Alpha [Sites 1992] performance counters could be adapted, as could counters for any of a variety of events [Gleason 1994]. Object-code editing [Graham et al. 1995; Mellor-Crummey and LeBlanc 1989] 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 operating system kernel and all user processes is edited so that the hypervisor is invoked periodically. Or, one can simply modify the code generator of a compiler to cause periodic incursions into the hypervisor whenever a program produced by that compiler is executed.

So, by virtue of the Instruction Stream Interrupt Assumption, execution of a virtual machine is partitioned into epochs. Corresponding epochs at the

primary and backup virtual machines comprise the same sequences of instructions if:

(1) The backup hypervisors are kept one or more instructions behind the primary so that a backup executing an environment instruction changes state in the same way the primary did.
(2) The backup hypervisors deliver at the end of an epoch E copies of the virtual machine's interrupts and accompanying data that primary's hypervisor delivered at the end of its epoch E. (Recall, copies of the primary's interrupts are forwarded to the backup.)

If the recovery register is readable as well as writable, it becomes possible to deliver virtual-machine interrupts at the primary as soon as they are received, thereby ending epochs dynamically. The primary would simply read the value of the recovery register when it delivers the virtual-machine interrupt and send the value read to the backups. Provided backups lag one or more epochs behind the primary, each backup hypervisor could set its recovery register to receive control and deliver virtual-machine interrupts at the same points in its instruction stream as the primary did.

We now summarize the protocol that ensures that 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 there is a single backup; generalization to multiple backups is straightforward. We also assume that the channel linking the primary and backup processors is FIFO though not necessarily reliable. Finally, we 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 sufficiently large timeouts used for failure detection).

Each hypervisor maintains an epoch counter: e_p equals the number of the epoch currently being executed by the primary; e_b is analogous for the backup. The protocol is presented as a set of routines that are implemented in the hypervisor. These routines execute concurrently.

First, we treat the case where a processor running the primary virtual machine has not failed.

P0: If primary's hypervisor processes an environment instruction at location pc:

— primary sends $[e_p, pc, Val]$ to backup, where Val is the value produced by executing the environment instruction;
— primary awaits acknowledgment for that message.

P1: If primary's hypervisor receives a virtual-machine interrupt Int:

— primary buffers Int for delivery at epoch end.

P2: If epoch ends at the primary:

— primary sends to backup all virtual-machine interrupts buffered during epoch e_p;

— primary awaits acknowledgment for that message;
— primary delivers all virtual-machine interrupts buffered during epoch e_p;
— $e_p := e_p + 1$;
— primary starts epoch e_p.

P3: If backup's hypervisor processes an environment instruction at location *pc*:

— backup awaits receipt of $[e_b, pc, Val]$ from primary;
— returned value is *Val*.

If backup's hypervisor receives a message $[E, pc, Val]$ from primary:

— backup sends an acknowledgment to the primary;
— backup buffers *Val* for delivery when environment instruction at location *pc* in epoch *E* is executed.

P4: If backup's hypervisor receives a virtual-machine interrupt *Int* destined for the backup virtual machine, then it ignores *Int*.

P5: If epoch ends at the backup:

— backup awaits message with virtual-machine interrupts for epoch e_b from primary;
— backup sends an acknowledgment to the primary;
— backup delivers all virtual-machine interrupts buffered for delivery at end of epoch e_b;
— $e_b := e_b + 1$;
— backup starts epoch e_b.

Now consider the case where the processor executing the primary virtual machine fails. Suppose the failure occurs after the primary starts epoch *X* but before sending the message in P2 to the backup's hypervisor. After the backup virtual machine begins executing epoch *X*, it will not receive in P3 and/or P5 expected messages from the primary's hypervisor. Failure detection notifications will take their place. Upon receipt of a failure detection notification, the backup computes the outcome of environment instructions by itself and no longer waits for the primary in order to start its next epoch.

P6: If backup's hypervisor receives a failure notification in place of $[e_b, pc, Val]$ then the backup's hypervisor itself executes that environment instruction.

If in P5, backup's hypervisor receives a failure notification in place of a message from the primary:

— $e_b := e_b + 1$;
— backup starts epoch e_b;
— backup is promoted to primary for epoch $e_b + 1$.

The backup is promoted to the role of the primary at the start of the epoch following the primary's failure (e.g., $X + 1$), so there is exactly one primary at the start of each epoch. If the primary fails, then an epoch may end with no primary. During such an epoch, the backup executes environment instructions as if it were the primary, but (as we discuss below) with respect to I/O instructions, the backup does not function as if it were the primary.

It is important to understand what P0 through P6 do and do not accomplish. P0 through P6 ensure that the backup virtual machine executes the same sequence of instructions (each having the same effect) as the primary virtual machine. P0 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.

P0 through P6 do not prevent an anomalous output sequence as a result of a failure. P0 through P6 also do not guarantee that virtual-machine interrupts from I/O devices are not lost. For example, 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. We address these issues in Sections 2.2 and 2.3.

2.2 Interaction with the Environment: State Machine Outputs

In the state machine approach, replica failures are masked by the mechanism that combines replica outputs. The output of this mechanism ideally is a sequence that would be produced by a single fault-tolerant state machine. This ideal is unattainable in our system, because all outputs during a given epoch are produced by a single state machine replica. In particular, no protocol can exist to inform one replica whether another replica produced an output before failing, since distinct operations are required (i) for a replica to perform an output and (ii) for one replica to communicate with another.[4] Consequently, there is no way for a state machine replica to know if another replica has produced a given output, and our fault-tolerant computing system may repeat some outputs issued prior to a failure. (Section 2.3 details when outputs will be repeated.) The environment must be able to tolerate this behavior.

Our state machine outputs are I/O operation requests. Possible repetition of outputs is the price we pay for not changing the hardware interface between processors and I/O devices. Any hardware mechanism (e.g., a voter) for combining I/O operation requests from multiple replicas would change the interface between the processor and I/O devices (as well as itself being a single point of failure).

[4]The argument is as follows. In a protocol where the send to the backup replica occurs after the output is performed, the primary replica's failure after the output, but before the communication, would cause the backup to conclude (erroneously) that the output was not performed; in a protocol where the notification is sent before the output is performed, the primary's failure after the send, but before the output, would cause the backup to conclude (erroneously) that the output occurred.

The mechanism we use to "combine" outputs produced by state machine replicas is quite simple: only outputs from the primary are passed to the environment. This means that I/O instructions executed by a backup are absorbed by the backup's hypervisor. That is, no I/O instruction at the backup actually results in an I/O operation being requested, although the hypervisor at the backup simulates making the request. If the primary fails, then the backup becomes the primary (see P6) and I/O instructions executed by the backup virtual machine will cause I/O operations to be requested.

2.3 Interaction with the Environment: State Machine Inputs

Use of a single state machine replica to resolve all nondeterministic choices has subtle ramifications, because future nondeterministic choices might be constrained by past choices. For example, clocks are monotonic. So, reading a clock—a nondeterministic choice—must return a value that is constrained to be larger than every past value returned. As another example, having delivered an I/O interrupt for an I/O instruction might preclude or constrain subsequent delivery of other interrupts for that request.

When a single replica (i.e., the primary) makes all of the nondeterministic choices, it is not difficult to preserve arbitrary constraints on these choices. But if this replica fails, and another replica (i.e., the backup) takes over, sufficient information must be available so that subsequent nondeterministic choices remain consistent with those made in the past. We identified two sets of constraints for the state machines of our system. One concerned clocks, and the other concerned I/O interrupt delivery.

Clocks on different processors run at slightly different rates. This means that the real-time clock on the backup is not necessarily synchronized with the one at the primary. They must be. Otherwise, switching from the primary's clock to the backup's may violate programmers' expectations that (i) real time increases and (ii) that the interval timer, which is driven off the real-time clock, decreases at the same rate as the real-time clock.

We wanted to avoid using a standard clock synchronization protocol because of the complexity and expense. For our purposes, it suffices that the value of the primary's time-of-day clock be sent to the backup at the end of each epoch. This information is included in the message sent in P2 by the primary's hypervisor to the backup's hypervisor. The information allows a newly promoted primary to ensure that any clock values returned or interval timer interrupts scheduled are consistent with prior execution by the now-failed primary. Two actions by the hypervisor are required. At the start of each epoch, the backup's hypervisor loads a corrected time-of-day value into the time-of-day clock. Second, the backup's hypervisor, if necessary, delays the next interval timer interrupt so that it is delivered at a time consistent with the corrected time-of-day clock.

I/O interrupt delivery is the other nondeterministic choice made by the primary where past choices constrain future choices. The sequence of interrupts delivered by a virtual machine must be one that could be delivered by a fault-free processor. We saw at the end of Section 2.1 that a failure may

prevent an interrupt that was delivered by the primary's hypervisor from reaching the backup's hypervisor. This cannot be avoided.

Our solution was to define a set of constraints that could be satisfied by the sequence of I/O interrupts a processor delivers, even when some interrupts delivered at the primary are not forwarded to the backup. That is, we defined what the operating system's I/O device drivers must assume about the behavior of I/O devices.

I / O Model Assumptions:

IO1: If an I/O instruction is executed, and the requested I/O operation is performed, then a corresponding *completion* interrupt is delivered by the processor issuing that I/O instruction unless that processor has since failed. If the issuing processor fails before delivering the *completion* interrupt, then the I/O device continues as if the interrupt had been delivered.

IO2: In the course of processing an I/O operation, an I/O device may cause an *uncertain* interrupt to be delivered by the processor issuing the corresponding I/O instruction. An uncertain interrupt signifies that the I/O operation has been terminated and is delivered in lieu of a completion interrupt. The instruction may have been in progress, may have been performed, or may not have been started.

For disks and networks, a device driver will reissue its last I/O instruction upon receiving an uncertain interrupt. This works because the state of a disk is insensitive to repetitions of the last I/O operation, and network protocols themselves send and ignore duplicate messages. A device driver for a tape drive, upon receiving an uncertain interrupt, might rewind the tape, reread it, and repeat the last instruction if necessary. So, although the device is not insensitive to repetitions of the last I/O operation, the driver can interact with the device to determine whether the last I/O operation executed successfully. Not all I/O devices, however, are insensitive to repetitions and/or testable in the sense we require. Such devices cannot be used with our scheme. Fortunately, they are not very common.

With the I/O Model Assumptions, a backup hypervisor can tolerate not receiving interrupts buffered by the primary hypervisor. A backup promoted to primary simply delivers uncertain interrupts for outstanding I/O operations:

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 (i.e., just before the backup is promoted to primary).

The effect of P7 is to notify the state machine at the backup of the primary's failure, because the uncertain interrupt is a state machine command. Each such command is processed by I/O device drivers that are awaiting interrupts. Implementing P7 requires the hypervisor to keep track of outstanding I/O operations. Thus, this provides a second reason for the hypervisor to receive control whenever an I/O instruction is executed: not only must some I/O operation requests be suppressed by the hypervisor (i.e., those at the

backup) but all I/O operation requests must be noted so that uncertain interrupts can be generated, if appropriate, at the end of a failover epoch.

An alternative design would be to delete P7 and IO2, modify P6 so that the backup is promoted one epoch earlier, and have the backup's hypervisor repeat any I/O operation requests that are outstanding when the backup starts the epoch in which the primary failed. Not only does the hypervisor repeat I/O operation requests, but the backup virtual machine would repeat I/O instructions executed by the primary during the epoch in which the primary fails. Clearly, I/O devices must be insensitive to all this repetition. Since some I/O devices are testable but not insensitive to repetition, this alternative design supports simpler I/O Model Assumptions at a cost of handling a slightly smaller class of I/O devices.

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 that can execute a single HP-UX (HP's UNIX system) virtual machine and then augmenting that hypervisor with the protocols of Section 2. Our prototype consists of two HP-9000/720 PA-RISC processors connected by both a SCSI bus and an Ethernet. We chose these processors because they have a recovery register. 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.

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, efficiency is achieved by taking advantage of a dual-mode processor architecture, whereby running in *supervisor* mode allows both *privileged* and *nonprivileged* instructions to be executed, but running in *user* mode allows only nonprivileged 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 that instruction.

Implementation of a hypervisor for all of HP's PA-RISC architecture is not completely straightforward, however. Two aspects of the instruction-set architecture prevent efficient virtualization: the memory architecture and the processor's privilege levels. We avoided these difficulties because we constructed a hypervisor that supports only a single instance of a virtual machine that executes HP-UX. It does not virtualize all of the PA-RISC architecture and cannot support multiple virtual machines. Our hypervisor is

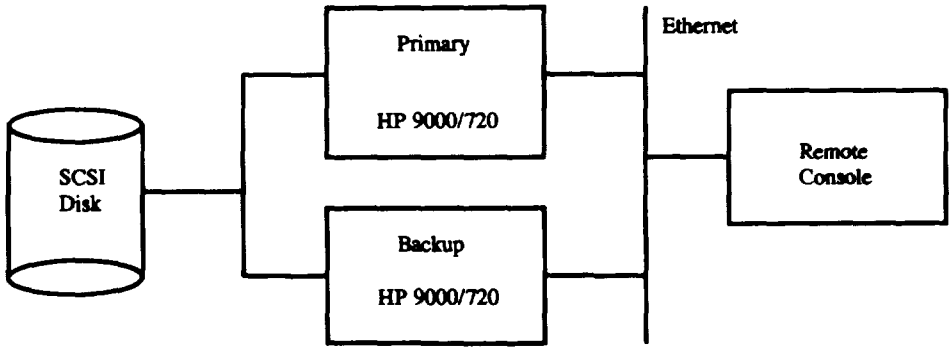

Fig. 1. The prototype.

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 uses a set of *space registers*. Instructions that read the registers can be nonprivileged, and a subset of the space registers may even be written to using nonprivileged instructions. Because the hypervisor cannot intercept all accesses to space registers, supporting multiple virtual machines is impossible. This is what led us to implement a hypervisor that supports only a single virtual machine.

We include the hypervisor in the address space of the virtual machine's kernel; the hypervisor appears to be a device driver to the kernel. 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 reads and writes to access rights for 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 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 current privilege level to be stored in the low-order bits of the return address. The presence of the hypervisor then becomes visible to programs.

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 prototype, 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.

To deal with the return addresses from branch-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.

3.2 Replica Coordination in the Hypervisor

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

The I/O 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 ordinary instructions are not necessarily deterministic on 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 (by software). 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 nondeterministic. An identical series of location references and TLB insert operations at the processors running the primary and backup virtual machines could 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. In particular, the number of instructions executed by the primary and backup virtual machines will differ.

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.

The Environment Instruction Assumption instantiated for the HP 9000/720 concerns instructions that (i) read the time-of-day clock, (ii) read data from I/O devices, and (iii) read from the registers of the SCSI I/O controller. Case (i) is dealt with because the instruction to read the time-of-day clock causes a trap to the hypervisor. As discussed above, case (ii) is dealt with by delivering to the backup copies of the interrupts delivered to the primary. For case (iii), we employ the virtual memory of HP PA-RISC processors, because the registers of the SCSI I/O controller are memory mapped. SCSI I/O controller registers are accessed through ordinary load and store instructions. Our hypervisor alters the access protection for the memory pages associated with these SCSI I/O controller registers so that a load or store attempted by the virtual machine causes an access trap to occur. The access trap transfers control to the hypervisor.

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

Our prototype implements two optimizations to the protocol of Section 2. First, P1 is modified so that interrupts are forwarded to the backup immediately rather than waiting until each epoch ends at the primary. This modification reduces the communications delay incurred at epoch boundary processing, since some communication is now overlapped with execution of the primary virtual machine. The impact of the modification depends on the amount of information that must be forwarded, the communications channel bandwidth, and the cost of sending a message. Interrupts from read I/O operations may be accompanied by the data read, so we believed the amount of data sufficient to warrant the additional complication to the protocol.

Second, P0 has been modified so that the primary does not immediately await an acknowledgment after forwarding to the backup the value produced by executing an environment instruction. We require only that the acknowledgment be received before the primary virtual machine requests an I/O operation, since I/O operations are the only way the outcome of the environment instruction could be revealed to the environment. Even if the primary fails after executing the environment instruction, and the message to the backup conveying the outcome of that instruction is lost, provided no I/O operation request has been made by the primary, subsequent actions by the backup virtual machine—whatever they may be—will be consistent with what the environment could observe for a single nonfaulty processor.

3.3 Implementing the I/O Model

The interface provided by the SCSI I/O controller differs from that implied by I/O Model Assumptions IO1 and IO2. The SCSI controller executes *I/O programs*, which reside in memory. Execution of an I/O program, usually initiated by a device driver, is caused by loading the address of that program into one of the I/O controller registers. The I/O program instructions cause

changes to the lines of SCSI bus, leading to data transfers to or from the I/O devices connected to the bus. Each I/O program—not each I/O instruction—terminates by causing an interrupt.

Because each I/O program terminates with an interrupt, I/O programs correspond to the I/O instructions of IO1. The CHECK_CONDITION interrupt status of the SCSI I/O controller has the same meaning as the uncertain interrupt of IO2. Thus, P7 can deliver a CHECK_CONDITION interrupt status for each outstanding I/O operation request, and the HP-UX SCSI disk driver will correctly handle these "uncertain interrupts."

However, the SCSI bus protocol allows a controller to disconnect itself from the bus while an I/O device is processing certain I/O instructions. A controller in this state never produces a CHECK_CONDITION interrupt, so there are times when the hypervisor had better not present such an interrupt to the driver. This restriction on presenting CHECK_CONDITION interrupts would seem to prevent P7 from being implemented. However, the problem is easily solved by implementing in the hypervisor a coarse simulation of the SCSI controller. Such a simulator is feasible because the hypervisor receives control whenever a load or store to a SCSI I/O controller register is attempted. The simulator in the backup hypervisor accepts I/O program execution requests and delays delivering the CHECK_CONDITION until the SCSI I/O controller is in a state where delivery is feasible. During a failover, the simulator also executes the necessary I/O program to place the SCSI controller in the state that it should be (given previous activity by the primary).

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 effects of various system parameters.

Normalized performance was identified as the figure of merit. A workload that requires N seconds on bare hardware has a *normalized performance* of N'/N if that workload requires N' seconds when executed by a primary virtual machine that communicates with a backup virtual machine, as implemented by our hypervisor. 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. Note that normalized performance does not reflect the use of two processors to accomplish the work of one, albeit with a degree of fault tolerance.

Quantifying the effect of epoch length on normalized performance was our paramount concern. A second concern was interrupt delay. 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 happen, but hypervisor delays for interrupt delivery may become significant.

4.1 CPU-Intensive Workload

Our first investigations concerned a CPU-intensive workload. A high-priority process executed 1 million iterations of the Dhrystone 2.1 benchmark. Each experiment was repeated 20 times.[5] Epoch boundary processing (i.e., rule P2) was measured to consume an average of approximately 442 μsec. on an HP 9000/720 processor,[6] as follows.

- 25μsec. hypervisor entry code
- 42μsec. Ethernet controller setup
- 141μsec. transmission time for end of epoch message and backup's acknowledgment
- 220μsec. await backup's receipt of end of epoch message and sending acknowledgment
- 14μsec. hypervisor exit code

Our Ethernet controller-setup and transmission measurements are consistent with the experimental data reported in Thekkath and Levy [1993].

We also measured that an average of 15μsec.—the time required to execute approximately 750 instructions—is required for the hypervisor to simulate each privileged instruction, broken down as follows.

- 8μsec. for hypervisor entry/exit
- 7μsec. for instruction-specific simulation

Thus, unless epochs are extremely long and unless there are many simulated instructions, epoch boundary processing is the dominant cost. In particular, for epoch length EL instructions, where s is the fraction of virtual-machine instructions that must be simulated by the hypervisor, the worst-case delay for an interrupt is given in μsec. by the sum of the time for executing the EL instructions, the time for the hypervisor to simulate the s EL instructions, and the epoch boundary processing time:

$$0.02\ EL + 15\ s\ EL + 442$$

Typically s will be less than 1%, and therefore with epochs comprising 10K instructions, an interrupt might be delayed as long as (approximately) 2msec. (Notice that this delay is being dominated by the time devoted to simulation of instructions.) A 2msec. delay would be significant for a network adapter but not significant for a disk.

By increasing the epoch length, the total time devoted to epoch boundary processing for a given benchmark is reduced. The normalized performance

[5]The coefficient of variation for the parameters for this and the other experiments we report was sufficiently low for us to have confidence in the validity of using their averages. See Bressoud [1996] for details.

[6]The HP 9000/720 is approximately a 50MIPS processor, so a typical instruction should execute in 0.02 microseconds.

$NP_C(EL)$ for the CPU-intensive workload as a function of epoch length EL can be approximated by the following:

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

where

RT: real time required to execute workload on bare hardware (8.8 sec.)

n_{sim}: number of instructions simulated by hypervisor (1.08×10^5 or 0.025% of the total)

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 (442 μsec.)

C_{other}: delays caused by flow control for interrupt-forwarding communication between primary and backup hypervisors—needed because the backup has a limited capacity for buffering DMA inputs that accompany an interrupt (41msec. 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 we made of our prototype for epoch lengths 1K, 2K, 4K, and 8K. (Our prototype does not support longer epochs due to limited buffer space in the backup for information sent by the primary. In retrospect, we would have been better off building a prototype that did permit longer epochs.) 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. However, long epochs cause delays in interrupt delivery. There are no I/O operations in our CPU-intensive benchmark, so delaying I/O interrupts is not a concern. But HP-UX does require that epoch lengths not exceed 385,000 instructions (10msec.), because of the way the clock is maintained by the kernel. With this CPU-intensive benchmark and epoch lengths of 385,000 instructions, our model predicts a normalized performance of 1.24.[7] This performance would be quite acceptable, especially since the hypervisor's simulation of instructions accounts for 0.18 of the 0.24 overhead. For long epochs, then, we predict that our replica-coordination scheme would be responsible for adding only 6% overhead beyond that incurred to simulate instructions.

4.2 Input / Output Workloads

We would expect a workload in which I/O operations occur to perform differently than the above CPU-intensive workload. First, requesting an I/O

[7]Anecdotal evidence (private communication, R. Coweles, 1995) 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.

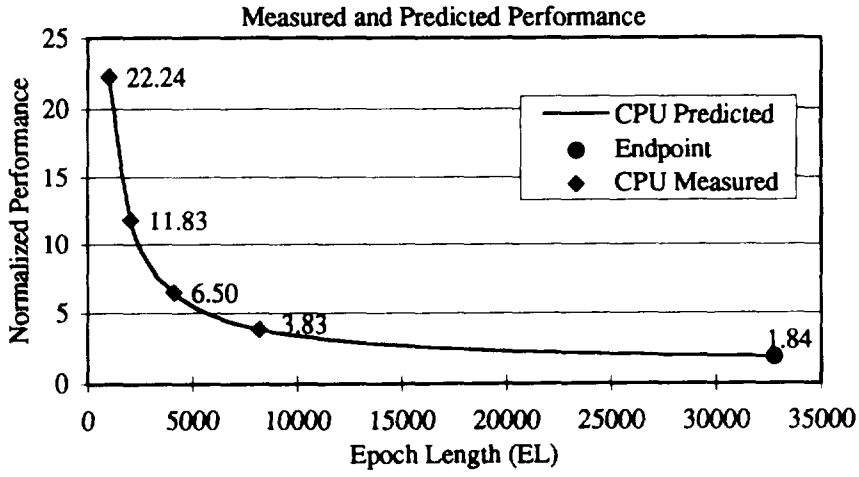

Fig. 2. CPU-intensive workload.

operation involves a significantly higher proportion of instructions that must be simulated by the hypervisor. Second, there is the added cost 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).

When analyzing an I/O-intensive workload involving a disk, care was taken to ensure that I/O operation requests actually cause I/O activity rather than being satisfied by a buffer pool maintained by the operating system. 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. Only then is the result a benchmark that measures the worst-case impact of our protocols. Thus, operating systems that make effective use of buffer pools or I/O operation pipelining should see better performance than indicated by our measurements.

This leads to the following I/O benchmarks. A large file is preallocated 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. To set-up each read, approximately 4.25% of the executed instructions were privileged and had to be simulated by the hypervisor. The benchmark for writes is analagous—a disk block is randomly selected; a write is issued; and then the write completion is awaited. Here, 5% of the instructions for set-up were privileged and had to be simulated by the hypervisor.

We ran experiments in which the write version of the I/O benchmark was executed. The normalized performance with 4K epochs was found to be 1.67. This normalized performance includes the impact of the hypervisor on the block selection calculation and memory-mapped I/O loads and stores to

execute the I/O instruction to cause a write operation. With 5% hypervisor-simulated instructions and each such simulation taking an average of 750 instructions, the hypervisor instruction simulation costs alone would suggest a normalized performance of 37. That we measured only 1.67 suggests that the workload was dominated by time spent waiting for the disk. Therefore, we also measured the disk write times. When the benchmark is executed on bare hardware, a disk write takes an average of 26msec. to complete; when the hypervisor and replica-coordination protocols are present, a disk write takes an average of 27.8msec. 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. The benchmark is not completely successful in selecting disk blocks not in the buffer pool—of the 2048 read requests issued, on average only 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 24.2msec. to complete; when the hypervisor and replica-coordination protocols are present, a disk read takes an average of 33.4msec. 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 acknowledgment or approximately 6.4msec. The remaining 2.8msec, then, is overhead.

Normalized performance $NP_{IO}(EL)$ for the I/O benchmark can be approximated by:

$$NP_{IO}(EL): \quad \frac{n_{IO}(cpu(EL) + xfer_{IO} + delay_{IO}(EL))}{RT}$$

where

RT:	real time required to execute workload on bare hardware
n_{IO}:	number of I/O operations (2048 for the write benchmark and 1729 for the read 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
$xfer_{IO}$:	elapsed time between initiation of disk I/O and receipt of the corresponding interrupt (26msec. for the write benchmark and 24.2msec. for the read benchmark)
$delay_{IO}(EL)$:	elapsed time between the completion interrupt and its delivery to the virtual machine when the epoch length is EL

A graph of $NP_{IO}(EL)$ for the write and read benchmarks with epoch length EL between 1K and 32K instructions appears as Figure 3. Measurements for

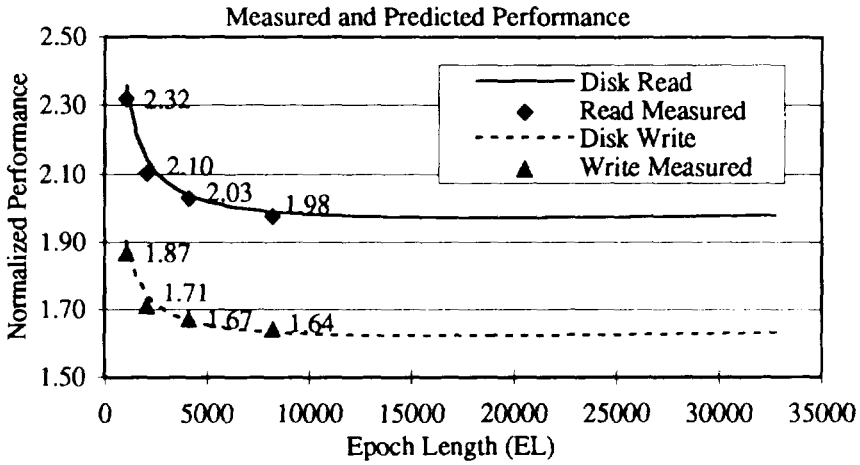

Fig. 3. Input/output options.

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.

As with the CPU-intensive workload, longer epochs lead to better normalized performance. This is because the $cpu(EL)$ term dominates in our models. But another trend is also visible. Increases to epoch length EL cause $delay_{IO}(EL)$ to increase, because interrupts from the disk are buffered for a longer period by the hypervisor. This trend explains the slight upward drift of normalized performance for larger epoch lengths. In a benchmark where more computation is done before each I/O operation, the dominance of the $cpu(EL)$ term would ameliorate the normalized performance. In the limit, as epoch length increases, normalized performance for the I/O workload experiments would be worse than for the CPU-intensive workload, because of the high percentage of hypervisor-simulated instructions for doing I/O.

4.3 Faster Replica Coordination

The predominant overhead for the replica-coordination protocols 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.

We simulated a faster network without changing the per-message execution overhead. For our CPU-intensive workload, there is not a big improvement, since most of the time is in that overhead. See Figure 4, which was obtained by scaling the 141 μsec. epoch boundary processing delay to reflect the use of an ATM link instead of an Ethernet. We would expect a larger improvement for our read benchmark and a still-larger improvement if we

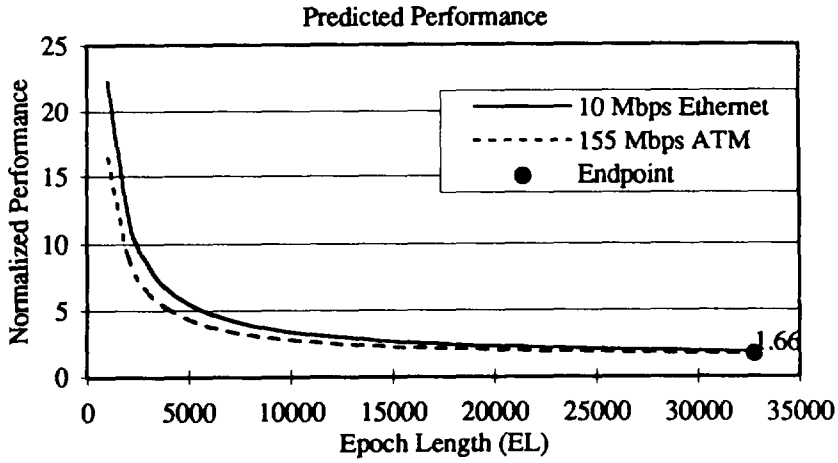

Fig. 4. Faster communication.

also improved the per-message overhead by optimizing the routines for I/O controller set-up time and the software driver execution.

A second improvement results from appreciating that it is not strictly necessary for the primary's hypervisor to await the acknowledgment we are requiring in P2. The argument is the same as used above for awaiting acknowledgments in P0. If the delivery of an interrupt *Int* is not revealed to the environment and if the primary fails, then subsequent actions by the backup virtual machine will be consistent with what could be observed by the environment were there a single nonfaulty processor (even if that interrupt is not forwarded to the backup). Thus, it suffices that the acknowledgments formerly awaited in P2 be received prior to the next I/O operation request by the primary virtual machine, since such an operation is the only way in which the receipt of that interrupt is revealed to the environment.

The modifications to the protocol of Section 2 are straightforward. First, in P2, the primary's hypervisor need no longer await acknowledgments for interrupts it forwarded to the backup's hypervisor. Second, in order to initiate an I/O operation, the primary's hypervisor is required to have received acknowledgments for all interrupts it has forwarded to the backup's hypervisor prior to the start of the current epoch.

We performed these modifications to the prototype and reran our experiments for the CPU-intensive workload of Section 4.1 and the two input/output workloads of Section 4.2. The results are given in Table I. The column labeled "Old" refers to the original protocol, and "New" refers to the modified protocol. The maximum epoch length reported is 8K instructions, because that is the longest epoch currently allowed by our prototype.

As expected, normalized performance improves significantly when acknowledgments need not be awaited in P2. The effect is most pronounced in the CPU-intensive workload, because its normalized performance is most affected

Table I. Normalized Performance of Original and Revised Protocol

	Workload					
	CPU Intense		Write Intense		Read Intense	
Epoch Length	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 II. Predicted Normalized Performance and Variable-Length Epochs

Workload	Normalized Performance
CPU Intense	1.24
Write Intense	1.57
Read Intense	1.92

by the delay at epoch boundaries. 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.

A final design alternative is to use variable-length epochs. Instead of terminating each epoch after a fixed number of instructions execute at the primary, an epoch is terminated only when an interrupt becomes available for delivery at the primary. Using this scheme with the CPU-intensive workload would cause epochs to be terminated according to the clock interrupt rate; with the I/O-intensive workloads, all CPU set-up for a transfer would occur in a single epoch; and when the completion interrupt for a transfer arrived, the epoch would immediately terminate, and epoch boundary processing would begin. Table II shows predicted normalized performance for the three workloads. Details can be found in Bressoud [1996].

5. RELATED WORK

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. Despite the engineering and time-to-market costs, manufacturers continue to design and sell processors that implement replica coordination in hardware. A design from Tandem [Cutts et al. 1988] and DEC's VAXft 3000 are examples. See Siewiorek and Swarz [1992] for a survey of such hardware-implemented fault-tolerant computing systems.

In some systems, like 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) [Siewiorek and Swarz 1992]. In the pioneering work of Tandem [Bartlett 1981], the applications themselves are responsible for ensuring coordination between the processes comprising a process-pair, the unit of replication there.

Other systems exploit a bus or broadcast network to implement fault-tolerant processes on top of an operating system. The work described in Borg et al. [1983; 1985] and in Powell and Presotto [1983] exemplify this approach. (See Schneider [1990] for a survey on various incarnations of the state machine approach in hardware and software for implementing fault-tolerant systems.) Novell's NetWare [Major et al. 1992; 1994] is the most similar to our system. Both are structured as state machines, and both employ a primary backup scheme with failovers. However, in NetWare, a rigid internal structure is forced on the operating system, including the proscription of preemption. In our system, we do not impose a structure or decomposition on operating system internals, but instead introduce a hypervisor. Also, our system permits preemption. Finally, failovers are not masked from the environment in NetWare. NetWare expects I/O that is lost during a failover to be rerequested.

6. SUMMARY AND OPEN QUESTIONS

The prototype described in this article implements replica coordination above the hardware but below the operating system by augmenting a hypervisor. The hypervisor does have a significant performance impact, but, as we have shown, the additional cost of our replica-coordination protocols is not significant provided epochs are long enough. For epochs that are not too long (i.e., under 8K instructions) workloads involving I/O experienced a factor of approximately 2 slowdown. Our CPU-intensive workload requires much longer epochs (e.g., 32K instructions) before a factor of 2 slowdown is achieved. But longer epochs are not problemmatic for a CPU-intensive workload because such a workload, by definition, is unaffected by the delayed delivery of I/O interrupts entailed by having longer epochs.

Without a doubt, much work remains to be done in understanding how epoch lengths and attendant interrupt delays impact system performance. By building a prototype and experimenting with it, we hoped to show:

(1) The approach has sufficient potential to justify further implementation and experimentation.
(2) A recovery register can be quite useful for implementing fault tolerance and should be contemplated when defining an instruction-set architecture.

We believe that we have succeeded. Work of Elnozahy [1995] (recently brought to our attention) is now exploring a variety of ways that a recovery register can be employed in operating system and applications software, including support for fault tolerance without introducing a hypervisor.

It is difficult to compare the additional performance costs entailed by our approach with the savings it brings to hardware and software design costs. With our approach, a new (faster) processor realization can be exploited rapidly, since a hypervisor for a given instruction-set architecture should not require modifications for each realization. And, all operating systems for a given instruction-set architecture are made to be fault tolerant in a transparent manner without the need to modify each one individually.

Augmenting a hypervisor is not the only way to use our approach and support replica coordination above the hardware but below system software. One might modify a microkernel, for example, and realize many of the same benefits as enjoyed when a hypervisor is augmented. This alternative remains to be investigated. Or, one might employ object-code editing. Note, however, that our approach only tolerates uncorrelated failstop failures of the system at or below the replica-coordination level. Programming errors and hardware design errors usually cause correlated failures, so they are not addressed. Failures of I/O devices or other hardware external to a CPU are also not handled by the approach.

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 ongoing work.

Some questions remain that could be addressed by modifying our prototype. Performance measurements with longer epochs should be attempted, but this will require changing how information is buffered at the backup. The protocols for varying-length epochs should also be implemented and measured. We have not measured the delay that is observed between a primary's failure and the takeover by the backup. For our prototype, this failover time is quite high, but we believe that this is due to the way in which the backup's hypervisor resets the SCSI bus when being promoted to the primary.

Finally, we have not considered any of the issues associated with restarting a failed backup. The problem is to communicate the state of the primary without significantly delaying its execution. Also, in some cases it should be possible to delay or reorder certain outputs, if the environment is insensitive to this aspect of the state machine's behavior. While the general problem seems quite difficult, the problem might be tractable for specific systems.

ACKNOWLEDGMENTS

The idea and original protocols for implementing fault tolerance using a hypervisor were developed by Schneider while consulting at Digital Equipment Corporation. The other members of that project—Ed Balkovich and Dave Thiel—provided helpful feedback and support throughout. Ed Balkovich encouraged the construction of this prototype and helped us formulate a plan to evaluate its performance. Input from Butler Lampson was quite useful.

HP's John Wilkes was instrumental in providing us with access to HP-UX source code and getting answers to technical questions about our hardware. 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 R. Cooper, E. N. Elnozahy, shephard Andrew Birrell, and the SOSP reviewers for their detailed comments on earlier drafts of this article.

This article is an extensive revision of the SOSP publication, prompted by a detailed and thoughtful review provided by Lampson (too late to make the SOSP publication deadline). Comments by two other *TOCS* reviewers, Greg Ganger and Barbara Liskov, led to the discovery of a bug in the protocol description and to a number of clarifications regarding our implementation.

REFERENCES

ALSBERG, P. A. AND DAY, J. D. 1976. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd International Conference on Software Engineering* (San Francisco, Calif.). IEEE, New York, 627–644.

BERNSTEIN, P. A., HADZILACOS, V., AND GOODMAN, N. 1987. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, Mass.

BIRMAN, K. P. 1993. The process group approach to reliable distributed computing. *Commun. ACM 36*, 12 (Dec.), 37–52.

BARTLETT, J. F. 1981. A nonstop kernel. In *Proceedings of the 8th Symposium on Operating Systems Principles* (Asilomar, Calif., Dec.). ACM, New York, 22–29.

BRESSOUD, T. C. 1996. Hypervisor-based fault-tolerance. Ph.D. dissertation, Computer Science Dept., Cornell Univ., Ithaca, N.Y. Jan.

BORG, A., BAUMBACH, J., AND GLAZER, S. 1983. A message system for supporting fault tolerance. In *Proceedings of the 9th Symposium on Operating Systems Principles* (Bretton Woods, New Hamp., Oct.). ACM, New York, 90–99.

BORG, A., BLAU, W., GRAETSCH, W., HERRMANN, F., AND OBERLE, W. 1985. Fault tolerance under UNIX. *ACM Trans. Comput. Syst. 3*, 1 (Feb.), 63–75.

CUTTS, R. W., NIKHIL, A. M., AND JEWETT, D. E. 1990. Multiple processor system having shared memory with private-write capability. U.S. Patent 4,965,717, U.S. Patent Office, Washington, D.C. Oct.

ELNOZAHY, E. N. 1995. An efficient technique for tracking nondeterministic execution and its applications. Tech. Rep. CMU-CS-95-157, Carnegie-Mellon Univ., Pittsburgh, Pa. May.

GOLDBERG, R. P. 1974. Survey of virtual machine research. *Comput. Mag. 7*, 3 (June), 34–45.

GLEESON, B. 1994. Fault tolerant computer system with provision for handling external events. U.S. Patent 5,363,503, U.S. Patent Office, Washington, D.C. Nov.

GRAHAM, S. L., LUCCO, S., AND WAHBE, R. 1995. Adaptable binary programs. In *Proceedings of the 1995 USENIX Winter Conference* (New Orleans, La., Jan.). USENIX Assoc., Berkeley, Calif., 315–325.

HEWLETT PACKARD. 1987. *Precision Architecture and Instruction Reference Manual*. Part no. 09740-90014, Hewlett Packard, Cupertino, Calif. June.

IBM. 1972. *IBM Virtual Machine Facility/370 Planning Guide*. Pub. no. GC20-1801-0. IBM Corp., White Plains, N.Y.

KARGER, P. 1982. Preliminary design of a VAX-11 virtual machine monitor security kernel. DEC Tech. Rep. TR-126, Digital Equipment Corp., Hudson, Mass. Jan.

LAMPORT, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (July), 558–565.

MELLOR-CRUMMEY, J. M. AND LEBLANC, T. J. 1989. A software instruction counter. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, Mass., Apr.). ACM, New York, 78–86.

MAJOR, D., MINSHALL, G., AND POWELL, K. 1994. An overview of the NetWare operating system. In *Proceedings of the 1994 Winter USENIX* (San Francisco, Calif., Jan.). USENIX Assoc., Berkeley, Calif., 355–372.

MAJOR, D., POWELL, K., AND NELBAUR, D. 1992. Fault tolerant computer system. U.S. Patent 5,157,663, U.S. Patent Office, Washington, D.C. Oct.

MEYER, P. A. AND SEAWRIGHT, L. H. 1970. A virtual machine time-sharing system. *IBM Syst. J. 9*, 3, 199–218.

POPEK, G. J. AND KLINE, C. 1974. Verifiable secure operating system software. In the *AFIPS Conference Proceedings*. AFIPS, Montvale, N.J.

POPEK, G. J. AND KLINE, C. 1975. The PDP-11 virtual machine architecture: A case study. In *Proceedings of the 5th Symposium on Operating Systems Principles* (Austin, Tex., Nov.). ACM, New York, 97–105.

POWELL, M. L. AND PRESOTTO, D. L. 1983. Publishing: A reliable broadcast communication mechanism. In *Proceedings of the 9th Symposium on Operating Systems Principles* (Bretton Woods, New Hamp., Oct.). ACM, New York, 100–109.

SCHLICHTING, R. AND SCHNEIDER, F. B. 1983. Failstop processors: An approach to designing fault-tolerant computing systems. *ACM Trans. Comput. Syst. 1*, 3 (Aug.), 222–238.

SCHNEIDER, F. B. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv. 22*, 4 (Dec.), 299–319.

SITES, R. 1992. *Alpha Architecture Reference Manual*. Digital Press, Bedford, Mass.

SIEWIOREK, D. P. AND SWARZ, R. S. 1992. *Reliable Computer System Design and Evaluation*. Digital Press, Beford, Mass.

THEKKATH, C. A. AND LEVY, H. M. 1993. Low-latency communication on high-speed networks. *ACM Trans. Comput. Syst. 11*, 2 (May), 179–203.

Received July 1995; revised September 1995; accepted October 1995