BUILDING A VIRTUALLY FAULT-TOLERANT SYSTEM

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Thomas Charles Bressoud

May 1996

BUILDING A VIRTUALLY FAULT-TOLERANT SYSTEM

Thomas Charles Bressoud, Ph.D.

Cornell University 1996

All schemes for implementing fault-tolerance involve some form of replication. Replicas are assumed to fail independently, and each replica performs the same computation. Replicas may execute in parallel or, in the case of primary-backup protocols, in response to failures. Replication only works, however, if replicas are coordinated. Each replica must receive the same inputs in the same order, and each must be deterministic in its response to these inputs.

The key engineering issue that the designer of a fault-tolerant computing system must address is deciding where in the system to implement replica coordination. Some of the alternatives include implementing replica coordination in the processor or network hardware, in the operating system, or in the applications software.

A new solution is to implement replica coordination by augmenting the hypervisor of a virtual-machine manager and coordinating a primary virtual machine with its backup. This hypervisor-based fault-tolerance is transparent to the operating system and the applications programs executing above the hypervisor. In addition,

this selection allows a single hypervisor design to be used for all processors in an architectural family.

In this dissertation, we describe the protocols to implement hypervisor-based fault-tolerance. To assess the practicality of the approach, we constructed a prototype system for HP's PA-RISC architecture. The prototype hypervisor supports a single HP-UX virtual machine and implements the replica-coordination protocols. The prototype hypervisor has been instrumented to measure the overhead of all hypervisor-based activity. We have measured the performance of CPU-intensive workloads and disk I/O intensive workloads in this architecture and have built models allowing us to predict the performance for some alternative architectures.

Biographical Sketch

Thomas Charles Bressoud (Tom) was born on February 18, 1961 in Silver Spring, Maryland, the fourth child of six born to George and Leigh Bressoud. He graduated from Springfield North High School in Springfield, Ohio in 1979, distinguishing himself in mathematics and science classes and pursuing a strong avocation in theater.

Tom attended Muskingum College in New Concord, Ohio where he majored in mathematics and computer science. He also minored in business and was a fixture in the theater department. It was in the theater department that he met and fell in love with Suzanne Albrecht. In 1983, he graduated Magna Cum Laude with a Bachelor of Science.

After graduation, Tom accepted a position with MIT Lincoln Laboratory in Lexington, Massachusetts as an applications programmer. He quickly moved on to greater responsiblities, culminating in the design and implementation of a real-time radar imaging system, specializing in both the display systems and the real-time kernel. During this period, Tom also worked part-time on a Master of Science degree in computer science from Boston University, and received it in 1987.

Tom and Suzanne married on September 1, 1985. Then, in 1987, they moved to Kwajalein in the Marshall Islands. Still working for Lincoln Laboratory, Tom helped take the real-time imaging system developed in Lexington the final step—interfacing it to a live radar data stream.

In 1989, Tom and Suzanne decided that the time was right for Tom to pursue his goal of a Ph.D. in computer science. That decision brought them to Ithaca, New York and Cornell University. There, Tom received a Master of Science degree in 1992.

With both the course work and research effort of his Ph.D. complete, Tom joined Isis Distributed Systems in Ithaca in the fall of 1994. On May 5, 1995, Amanda Carol Bressoud was born and became the joy of Tom and Suzanne's lives. In August of 1995, the family moved to Massachusetts with Isis to pursue opportunities of furthering Tom's research.

Tom received his Ph.D. from Cornell in May 1996.

To Suzanne for her love and understanding

and

To Mom and Dad for instilling in me the desire to know more

Acknowledgements

I must first thank my family for all of their support and encouragement over the years. Most especially, I thank my parents for presenting so many opportunities to grow and learn and allowing me to be myself. I also thank my brothers and sisters for being my friends and for each bringing out a different facet of my personality. My family is so very important to me.

I am grateful to my advisor, Fred B. Schneider. The key to a strong dissertation is to start with a good idea—an idea that excites and motivates. In my case, Fred brought such a good idea into the partnership and helped me develop it into research that fit my strengths and creativity. I also thank Fred for his role as an editor. He taught me a clarity and consistency in writing that I have grown to appreciate.

I also thank the other members of my committee. Keith Marzullo has a wonderful ability to get to the key practical issues that is invaluable. Richard Conway and his class at the Johnson School provided an excellent alternative direction for my intellectual pursuits. Thanks also to Ed Balkovich from Digital Equipment Corporation, who served as an unofficial member of my committee. Ed encouraged the construction of the prototype and asked important performance level questions, which led to the formulation of the performance models and their analysis.

There are many students and staff in the Cornell Computer Science Department that I must also thank. My office-mates and fellow graduate students Aleta Ricciardi and Lorenzo Alvisi head the list. They always made the time to listen and discuss my ideas. I hope I did the same for them. Two masters students, Steven Smith and Jackie Proulx, used parts of the communication within the hypervisor as their master's project. Their contribution is appreciated. The staff in the department have always gone out of their way to help me out. In particular, I must name Anne Gockel, Linda Competillo, Carol Ayer, Jan Batzer, and Becky Personius.

There is more to life than research. Students from both the Applied Math and the Operations Research departments at Cornell helped provide that balance in my life. It may have kept me sane. Within the ranks of my teammates in intramural volleyball and basketball and softball, I have made some lifelong friends. Thanks especially to "the gang" at 206 Winthrop, Beth and Frank, Rob, and Patty.

Isis Distributed Systems, now a division of Stratus Computer, has been my employer for the last year and a half. They have encouraged me and have been patient with my continued pursuit in writing my thesis and working on papers in my research area. They have also allowed me to use company resources in the preparation of my thesis and papers. I am grateful to Robert Cooper, Brad Glade, and Ken Birman in particular for their support and technical discussions.

Last, and most important, I thank Suzanne. She is my wife and my friend and we work together so well. She encouraged me to follow my path toward a Ph.D., even though it meant additional responsibilty and a loss of security. She has been patient and understanding. I deeply appreciate her love and help. Without her, this work would have been impossible.

Support for this work was provided 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 presented in this work are mine and do not reflect the views of these agencies.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Fault-tolerance requires replication. The computation is replicated and the replicas are coordinated so they execute the same sequence of instructions on components that fail independently. The central problem in building a fault-tolerant system is to coordinate the replicas.

Many solutions to the replica coordination problem exist [Web92, BBG+89, WLG+78, Bar81, BvR94]. Each has a different degree of impact on the application programmer. At one extreme, solutions specific to an application may be crafted. Doing so, however, requires that an application programmer be knowledgeable about various message-ordering schemes and broadcast protocols. Alternatively, a solution may provide libraries of tools that programmers can then use in building solutions. Unfortunately, this solution forces the programmer into a particular programming paradigm — a paradigm that may or may not fit the application. At the other extreme, fault-tolerance support is hidden from the programmer. Such *transparent fault-tolerance* allows existing software to be run in a fault-tolerant way without re-

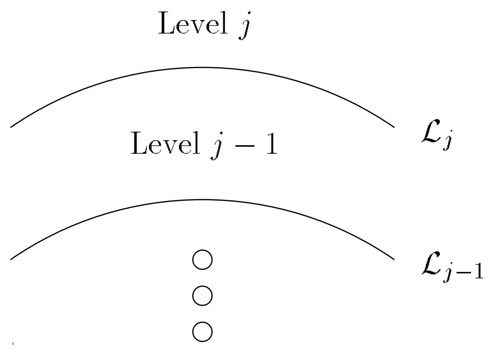


Figure 1.1: Levels and Interfaces

quiring redesign or restructuring. Transparent fault-tolerance is the subject of this thesis.

1.1 Actions: Instructions and Exceptions

Let level j be the level of the system where execution of replicas is to be coordinated. The interface between level j and level $j-1$ defines a set $\mathcal{L}_j$ of *actions* that level $j-1$ supports. (See Figure 1.1.) Each action transforms the state of level j. The actions of $\mathcal{L}_j$ can be partitioned into instructions and exceptions. An *instruction* of $\mathcal{L}_j$ is an action that is initiated by level j. An *exception* of $\mathcal{L}_j$ is an action initiated by level $j-1$.

Exceptions interrupt the normal sequence of instructions and are *delivered* by level $j-1$ at some point in the action sequence. Consider the action sequence

$$i_1 \; i_2 \; \ldots \; i_k \; e \; i_{k+1} \; \ldots$$

where e denotes an exception. Exception e is delivered after i_k. Note that i_{k+1} is not necessarily the action that would follow i_k had e not been delivered.

Some exceptions are associated with execution of the immediately preceeding instruction. For example, i_k might be an attempt to divide by 0, and e could be the resulting exception. Exceptions delivered immediately after i_k and arising from execution of i_k are said to be *synchronous* with respect to i_k. An exception that may be delivered at an arbitrary point in the instruction sequence is called an *asynchronous* exception. Clocks and I/O devices typically cause asynchronous exceptions.

In general, an action a of $\mathcal{L}_j$ may be realized by level $j-1$ in two ways. First, a may cause control to transfer to level $j-1$ so that level $j-1$ can perform the desired state change using actions from $\mathcal{L}_{j-1}$. In this case, we say a is *implemented by* level $j-1$. Second, a may be simply passed down to a lower level. For example, an instruction may be realized as a single instruction of $\mathcal{L}_{j-1}$, essentially bypassing level $j-1$. Or, an exception may be the same at $\mathcal{L}_{j-1}$ and $\mathcal{L}_j$; when such an exception occurs in the action sequence at $\mathcal{L}_j$, control of instruction execution bypasses level $j-1$. In both cases, we say such an action is *inherited* by $\mathcal{L}_j$ from $\mathcal{L}_{j-1}$.

An instruction of $\mathcal{L}_j$ may be implemented by level $j-1$ by executing a sequence of actions of $\mathcal{L}_{j-1}$. We call the execution of this sequence a *sub-computation* of level j. Likewise, instructions in a sub-computation may themselves be implemented by executing actions from even lower levels. To ensure all replicas of level j perform the same sequence of actions, it is not necessary for all replicas to perform the same sub-computations. It suffices that the sequence of actions of $\mathcal{L}_j$ be the same, regardless of sub-computations arising from those actions.

We equate the lowest level of a system with the *environment.* If an instruction i from $\mathcal{L}_j$ can affect the environment, then we call i an *output instruction.* If the state of the environment can be sensed by executing i, then we call i an *input instruction.* An instruction that reads the current time of day is an example of an input instruction. Finally, if the state of the environment can be sensed by an exception e, then e is called an *input exception*. For example, an input exception occurs when a message buffer is received, causing the program counter to be set to a handler routine.

1.2 Computations

Execution of an action of $\mathcal{L}_j$ is defined to be *deterministic* if the resulting state of level j is uniquely determined by its initial state. An action is *deterministic* if every execution is. We say that an action sequence $a_1a_2 \ldots a_n$ is *deterministic* if its execution produces a state that is completely determined by the initial state; otherwise, $a_1a_2 \ldots a_n$ is *non-deterministic*. Because input actions sense the environment, they are not deterministic. Thus, occurrence of an input action during an action sequence may cause that action sequence to be non-deterministic.

The sequence of actions at a given interface forms a *computation* at that interface. A sequence of actions can be partitioned into a series of possibly empty deterministic action sequences alternating with non-deterministic choices. Non-deterministic choices in a sequence of actions arise from state transformations for any action that is not deterministic and from the choice of points for delivering asynchronous exceptions.

1.3 Replica Coordination

To provide fault-tolerance at level j, replicas of that level are coordinated so that all non-faulty replicas execute the same sequence of actions. The desired coordination can be accomplished if replicas executed by non-faulty components begin in the same state and make non-deterministic choices in the same way. The computation performed by each replica will then be the same. Thus, the effects of output instructions of all correct replicas will be identical.

A *replica coordination* mechanism serves two functions:

(1) It combines outputs from the replicas to provide a single output from the ensemble.

(2) It ensures that all replicas resolve the same non-deterministic choices in the same way.

Function (1), how output from replicas is combined, as well as the number of replicas required to achieve fault-tolerance, depends on a *failure model* — a set of assumptions made about the behavior of faulty components. A component is considered *faulty* once its behavior is no longer consistent with its specification. Two representative failure models are:

- **Failstop.** A processor fails by halting. Once it halts, the processor remains in that state. The fact that a processor has failed is detectable by other processors [Sch84].

- **Byzantine.** A processor fails by exhibiting arbitrary behavior [LSP82].

A system is *t-fault-tolerant* if it satisfies its specification provided no more than t of its components become faulty.

If processors experience only failstop failures, then $t + 1$ replicas are required to build a t-fault-tolerant system; the output of any single replica can be used as the output of the ensemble. This is because with $t+1$ replicas, even if t failures occur, a correct replica remains and the output that it generates is correct. If, however, Byzantine failures can occur, then at least $2t+1$ replicas must comprise the ensemble; the output of the ensemble is taken to be the output produced by the majority of replicas. Here, if t failures occur, then $t+1$ correct replicas remain, so a majority of the outputs of the ensemble will be correct and a majority voter can be used to identify this correct output.

Function (2), for supporting replica coordination, requires replicas to resolve non-deterministic choices identically. There are two general approaches to this. The first is distributed. Here, candidates for each non-deterministic choice are disseminated to all correct replicas before any proceeds. Given the set of non-deterministic choices, a deterministic protocol is used in isolation by the replica coordination module at each replica to select one. The replica itself behaves as a deterministic state machine processing each choice provided by replica coordination. All correct replicas thus resolve each non-deterministic choice in the same way. This general method for managing replication is called the *state machine approach* [Lam78].

A second solution to ensuring that all replicas make the same non-deterministic choices is to have one replica designated as the *primary*. The primary makes the choices and conveys them to other replicas, called *backups*. This general method is called the *primary-backup approach* [AD76]. Note that the primary-backup approach cannot tolerate Byzantine failures since, were the primary faulty, it might send incorrect non-deterministic choices to correct backups, thereby causing them to diverge from the primary. In addition, the primary could simply produce incorrect outputs.

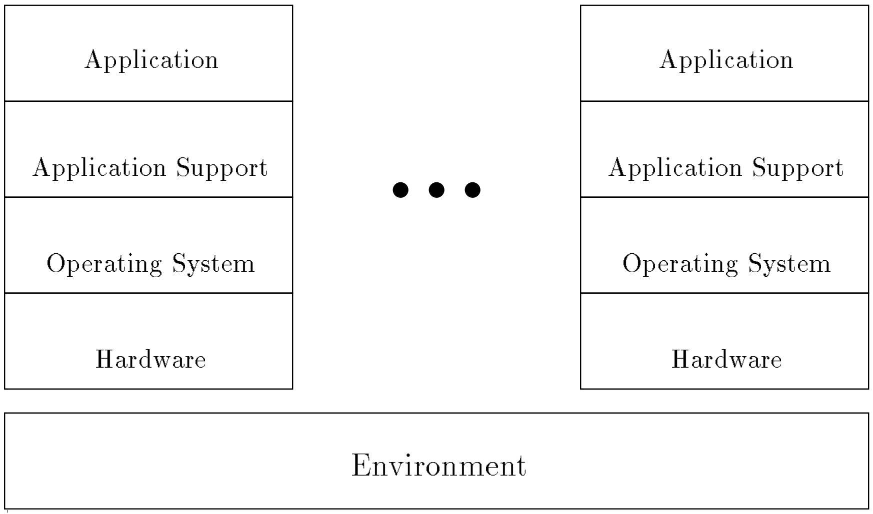


Figure 1.2: Level Structured System

1.4 Choosing a Level for Replica Coordination

With both the state machine approach and the primary-backup approach, one must decide at what level of the system to provide replica coordination. Consider the traditional level-structured system of Figure 1.2:

- At the lowest level, and common to all replicas, is the environment. The environment is the source of input and the destination of outputs for the system. Input instructions and input exceptions allow the state of the environment to be sensed by higher levels. Output instructions allow the environment to be affected.

- The hardware level interacts with the environment below and has an interface above that is described in the machine's *architectural definition.* The exceptions at this level are the hardware interrupts, typically generated by I/O devices in the environment.

- The next level is the operating system. It has access to the entire instruction set described in the architectural definition. The level manages the hardware and provides an extended interface for the levels above it. This extended interface includes the non-privileged instructions of the architectural definition along with additional instructions — *system calls* — that provide services and allow the next higher level to use machine resources. Exceptions provided by the operating system interface might include so-called signals or software interrupts.

- The level up from the operating system is typically an application support level. It provides additional functionality to higher levels, making these levels easier to program. Exceptions at this interface might take the form of message buffers arriving from outside the system.

Selecting a level for replica coordination defines a *replication management interface*, the interface between the selected level and the levels above it. The levels above are the fault-tolerant abstraction whose computation (i.e. action sequence), including output instructions, are coordinated between replicas.

Once a level $j-1$ for replica coordination has been chosen, we must modify the system so that non-determinstic choices affecting the computation at the replication management interface are disseminated among the replicas. This imposes restrictions on the way levels may behave. Level $j-1$ must be modified to provide replica coordination. Modification of other levels might also be necessary, as follows.

We must first restrict how actions at the replication management interface $\mathcal{L}_j$ are selected for execution. Given a point in a computation at level j, if the next instruction to execute or next synchronous exception to be delivered were not determined by the current state of level j, then that would constitute a choice that is not controlled by the replica coordination level. Different choices might be made by different replicas and the ensuing computation would diverge at these replicas. Therefore, we must not allow such action selection.

R1: Provided no asynchronous exception is delivered in the computation at the replication management interface, the next instruction or synchronous exception is uniquely determined by the state of the level above the replication management interface (replica).

An input instruction or an input exception that is part of the replication management interface and that is inherited from a lower level would again constitute a choice that is not controlled by the replica coordination level. Such choices cannot be regulated, so we prohibit such actions:

R2: All input actions of the replication management interface are implemented (i.e., not inherited) by the replica coordination level.

Likewise, if the replica coordination level does not receive control for an output instruction, it cannot combine outputs from the replicas. This gives rise to a restriction on inheriting output actions in the level supporting replica coordination.

R3: All output instructions of the replication management interface are implemented (i.e. not inherited) by the replica coordination level.

We must ensure that non-deterministic choices the replica coordination level handles — the input actions — are sufficient to cause all replicas to perform the same computation. This is accomplished by disallowing other non-deterministic actions.

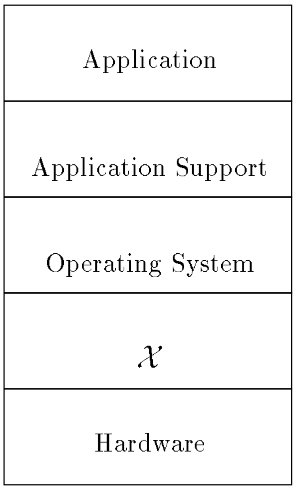


Figure 1.3: Interposing a Level

R4: All non-input actions of the replication management interface are deterministic.

We do not restrict the implementation of actions by the replica coordination level and lower levels. For example, the subcomputations implementing an action at different replicas may differ. As long as restrictions R1 – R4 are satisfied by the replica coordination level (and its interface), replica coordination may be achieved.

1.5 Selecting the Virtual Machine Level

Consider interposing a level $\mathcal{X}$ between the operating system and the hardware, as in Figure 1.3. If the interface to level $\mathcal{X}$ is identical to the architectural definition and properties R1 – R4 are satisfied, then $\mathcal{X}$ will support fault-tolerance in a way that is transparent to all the software of the original system.

This is significant. Its implications include the following.

- The same replica coordination software works for multiple hardware platforms in the same architectural family, because (by definition) all such platforms have the same architectural definition. Moreover, the solution tracks the performance of new members of the architectural family. It is not uncommon for a computer vendor to build different realizations of the same architecture over time. A few prominent examples include the System 370 architecture by IBM, the Vax architecture by DEC, and the Precision Architecture by HP.

- The solution supports all existing operating systems for the given architectural definition. Thus, individual operating systems do not require modification in order to be fault-tolerant.

- The solution need not be changed for operating system upgrades.

- The extant software base becomes fault-tolerant, transparently. This applies to *all* software that runs on the target platform.

A disadvantage of choosing such a low level in the hierarchy is that software failures occurring in higher levels are not addressed by our scheme. Such a failure would be treated simply as an action sequence of the computation, and all replicas would repeat the same failure.

We have postulated a level $\mathcal{X}$ that presents the same interface to upper levels as does the architectural definition and that controls all input and output. Is it possible to build a level having these properties?

Levels with a machine's architectural definition as both interfaces have existed for many years [Bai70, MS70, Gol74]. Such a level is known as a *virtual machine monitor* (VMM) or *hypervisor*, and the levels above it are know collectively as a *virtual machine*. If a machine's architecture can support a hypervisor, then we can augment this level with replica coordination and attain our goal of transparent fault-tolerance.

Virtual Fault-Tolerance (VFT) selects the hypervisor level for replica coordination. In theory, this makes it possible to build a transparent fault-tolerant system. However, just because such a solution is possible does not mean that it is practical. The practicality of our approach depends on its cost. To evaluate this, we have built a prototype fault-tolerant virtual machine on the HP Precision Architecture. Our prototype allows the costs of virtualizing this architecture and implementing replica coordination in a hypervisor to be measured.

Given these costs, we can then build models that allow us to predict the performance of such a system for varying design parameters. This modeling also allows the performance of alternative design choices to be predicted.

1.6 Organization of Thesis

This thesis investigates the practicality of providing transparent fault-tolerance by coordinating replicas at the hypervisor level. We built a prototype that supports HP's Unix, HP–UX, and then modelled and measured its performance. This involved implementing a hypervisor and integrating replica coordination protocols.

In Chapter 2, we describe the VFT protocol itself. Chapter 3 examines the issues involved in implementing a virtual machine system using the prototype target architecture. Chapter 4 describes the realization of the protocol in the target architecture. In Chapter 5, we discuss the performance of the prototype system and alternative designs. We conclude in Chapter 6 by comparing our work with other approaches to providing fault-tolerance; we also present suggestions for applying and extending this work.

Chapter 2

Virtual Fault-Tolerance

The Virtual Fault-Tolerance protocol (VFT protocol) described in this chapter allows a 1-fault-tolerant processor to be built using two fail-stop (ordinary) processors. The scheme is not concerned with implementing fault-tolerant disks or other peripherals. Other methods should be employed to address those issues. (See [LS80] for a discussion of fault-tolerant disks and [Cri90] for a discussion of failure detection.)

A production implementation of our VFT protocol would also provide a method to bring a repaired replica back online, reestablishing the system's ability to tolerate a single failure. Supporting reintegration does not appear to be difficult: a snapshot of the current state of the virtual machine at the primary would have to be communicated to the new backup before normal VFT protocol processing could continue. Such snapshot protocols are discussed in the literature [AAD$^+$90].

The VFT protocol was originally developed by Fred Schneider when consulting for Digital Equipment Corp. As of this writing, a patent on the protocol has just issued [SLBT96]; until now, however, it had not been implemented. Our implementation of the VFT protocol has led to an extension to the protocol (to handle input instructions) and a number of optimizations.

A *correct* VFT protocol ensures that all interactions between the environment and an ensemble of virtual machine replicas are consistent with interactions possible were a single 1-fault-tolerant processor present. We achieve this by causing the state of each replica to be consistent with the interactions between the ensemble and the environment. Thus, after a failure, any surviving replica can continue that interaction in a consistent way.

The discussion of the VFT protocol in this chapter uses the structure presented in Chapter 1, with the hypervisor as the level selected for replica coordination. The interface to that level, hereafter called the *virtual machine interface*, constitutes the replication management interface. The action sequence that is coordinated among replicas by the VFT protocol is the virtual machine computation, henceforth referred to as simply "computation".

The VFT protocol uses the primary-backup approach. The hypervisor at the primary makes all non-deterministic choices and conveys these to the hypervisor at the backup, enabling the backup virtual machine to perform an identical computation. In addition, until failing, the primary is responsible for all interactions with the environment. Following a failure, the backup takes over interacting with the environment as if it were the primary.

Figure 2.1 illustrates a primary and backup fail-stop processor combined to build a single 1-fault-tolerant processor. Both replicas share access to the environment, submit requests to the same elements of the environment, and receive responses for those requests.

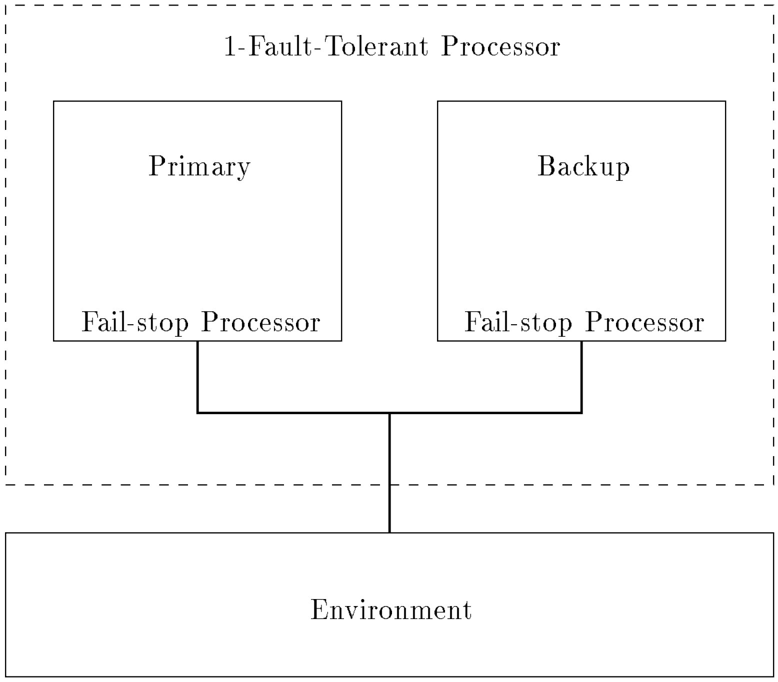


Figure 2.1: Fault-Tolerant Processor

2.1 An I/O Model

Possible interactions between a virtual machine and the environment must be discussed before the details of the VFT protocol can be presented. These interactions define what we call an *I/O model*. Obviously, the I/O model should impose the fewest restrictions on target architectures, so the VFT protocol will have the broadest applicability.

Since a virtual machine interface is identical to the architectual definition, it suffices to describe our I/O model in terms of that. On conventional processors, all interactions with the environment occur through I/O controllers, which control devices like disks and network adapters. Output instructions command these I/O controllers.

Output instructions found at the architectural definition take various forms. Some architectures provide dedicated instructions. On the IBM System 370 [Int87], for example, an I/O operation is initiated with a Start I/O instruction. Other architectures use memory loads and stores to initiate I/O operations by associating certain memory locations with the control registers of the I/O controllers. The Motorola 680x0 family of processors [Mot86] exemplify this type of architecture. Either form of output instruction is allowed by our I/O model.

When an output instruction is executed, by definition, it affects the environment. An output instruction might be a read or write request to a disk, a packet transmission request for a network, or a request to update a display. Such I/O operations are not completed instantaniously. I/O controllers and the devices they control are typically slow relative to the speed of the processor. To allow a processor to continue while an I/O operation is in progress, output instructions simply initiate activity by a controller. Once such a *request* has been made, the computation continues. Our first I/O model assumption is thus:

IO1: Output instructions give rise to requests, which initiate activity by an I/O controller.

When an I/O controller completes requested activity, an asynchronous exception is delivered to the processor. Such an asynchronous exception is called a *completion.*

A completion usually includes status indicating the outcome of the request. We call the status information returned in a completion a *completion status.* All completions reveal information about the environment through the completion status and, therefore, are input exceptions. Note that we even consider the completion for a request that outputs to the environment (such as a disk write) to be an input exception.

A completion status indicates either success or some error condition. Possible error conditions range from severe errors, which indicate I/O is no longer possible using the given I/O controller, to transient I/O errors, which indicate that there may have been a problem performing a particular request.

These assumptions regarding completions lead to the second requirement of our I/O model:

IO2: I/O controller activity in response to a request always terminates with a completion. The completion indicates the outcome of the request by providing a completion status. When a completion status indicates a transient I/O error, no assumption can be made about whether the request was started or was completed successfully.

Transient I/O errors are addressed in greater detail in Section 2.3.2.

To allow any individual replica to interact with the environment, all replicas must have access to all of the devices of the environment. The environment must also be able to return a completion to the appropriate replica. Thus our next I/O model requirement is:

IO3: All processors access the same devices in the environment. When an output instruction naming a device is executed, the completion is sent to the source of the request.

For example, a dual-ported disk allows two machines to read and write the same disk. The completion is sent through the port the request was made.

In addition to requests and completions, a processor may interact with the environment through input instructions. Recall that input instructions are synchronous with the computation. The state of the environment sensed by an input instruction is the *input instruction value* and some part of the state of the machine is updated with the input instruction value.

IO4: A processor may sense the environment through input instructions.

A final property concerning I/O allows communication between the hypervisor at the primary and the hypervisor at the backup. This enables them to communicate non-deterministic choices:

Comm: Communication is possible between the hypervisor at the primary and the hypervisor at the backup. Moreover, for all messages m and m', if m is sent before m' at the primary, m is delivered before m' at the backup.[1]

The VFT I/O model assumes that any device in the environment can exhibit a transient I/O error. If processing a request is idempotent, then the processor can

[1] This FIFO delivery is easily accomplished by adding sequence numbers to each message.

perform the simplist error recovery when notified of a transient I/O error — retry the request. Disks and network interfaces allow this and typical drivers for these devices already provide for transient errors with retry recovery.

Other devices may not fit as well with this model. The software handling a tape drive, for example, might have to backspace the tape in its recovery action. A device such as a card punch is even more problematic.

2.2 Instruction Sequences

We begin the description of the VFT protocol by assuming no failures and describing the coordination necessary to cause identical computations at different replicas. Failures are addressed in Section 2.3, where we examine the interaction of virtual machines with the environment.

By having the primary and backup virtual machines perform the same computations, the sequence of output instructions executed by the primary and by the backup will be identical, and the state of the virtual machine at the primary and backup will be the same after each action completes. By having only the primary interact with the environment, that interaction of the ensemble will be consistent with what is possible given a single machine. Thus, the state of the environment will be consistent with the interactions requested.

The VFT protocol assumes that the primary and backup virtual machines are initially in identical states. The *state* of a hypothetical virtual machine might include:

- Memory, including code and data.

- General purpose registers.

- Control registers. One control register might be a virtual program status word (PSW) specifying the current status of the machine. Another control register might be a register that enables or disables external interrupt delivery.

- Program counter (PC). The value of this register is the address of the next virtual machine instruction to be executed.

Given identical virtual machine initial states, then by ensuring that:

IS1: The next action executed at the primary virtual machine and the next action executed at the backup virtual machine are identical.

IS2: The state of the virtual machine is identical after execution of that action.

we guarantee that the computation will remain the same at both the primary and backup.

If x is an action of the virtual machine interface and S and S' are states of the virtual machine, we write $S \xrightarrow{x} S'$ to denote that execution of action x in state S can result in state S'.

Suppose the primary and backup both begin in state S, the next step x_P in the action sequence at the primary satisfies $S \xrightarrow{x_P} S'_P$, and the next step x_B in the action sequence at the backup satisfies $S \xrightarrow{x_B} S'_B$.

To discharge IS1, we must establish that $x_P = x_B$ holds. To discharge IS2, we must establish that $S'_P = S'_B$ holds. In general, the next action to be executed is either an instruction i, a synchronous exception s, or an asynchronous exception a. In the next few sections we argue that, in the absence of asynchronous exceptions, IS1 and IS2 hold. We then show how a VFT protocol maintains IS1 and IS2 even when asynchronous exceptions are possible.

In the following discussion, we refine the virtual machine state to separate the PC value and the set of pending synchronous exceptions Q from the remainder of the virtual machine state R. A triple $\langle \mathrm{PC}, Q, R \rangle$ denotes such a virtual machine state.

2.2.1 Next Action: Non-input Instruction

Suppose both the primary and backup virtual machine are in the same state $S \equiv \langle \mathrm{PC}, \{\}, R \rangle$, where no synchronous exceptions are pending.

We assume that:

A1: When no synchronous exceptions are pending, the next instruction to be executed is completely determined by PC and the contents of the virtual machine memory location referred to by PC.

Since the virtual machine state is the same at the primary and backup, the instruction i to be executed is also the same; thus IS1 holds.

Suppose further that i is not an input instruction. So, for the primary

$$\langle \mathrm{PC}, \{\}, R \rangle \xrightarrow{i} \langle \mathrm{PC}_P, Q_P, R_P \rangle$$

holds, and for the backup

$$\langle \mathrm{PC}, \{\}, R \rangle \xrightarrow{i} \langle \mathrm{PC}_B, Q_B, R_B \rangle$$

holds. If we make the assumption that:

A2: All non-input instructions are deterministic in the state of the virtual machine.

then $\mathrm{PC}_P = \mathrm{PC}_B$, $Q_P = Q_B$, and $R_P = R_B$ hold, satisfying IS2.

2.2.2 Next Action: Input Instruction

Again suppose both the primary and backup virtual machine are in the same state $S \equiv \langle \mathrm{PC}, \{\}, R \rangle$, where no synchronous exceptions are pending.

By A1, the next instruction i to be executed is the same at the primary and the backup, thus IS1 holds. Furthere suppose i is an input instruction.

For IS2 to hold, the transformation in state at the primary and backup as a result of executing the input instruction must be the same. Since the transformation is dependent on the input instruction value returned, the input instruction value must be the same at the primary and backup.

The primary is responsible for all interactions with the environment, so the primary must actually execute the input instruction and determine its value. It can then convey this value to the backup, so the backup may return the same value.

The required hypervisor control on input instruction execution is only possible if we assume:

A3: All virtual machine input instructions are implemented (i.e. not inherited) by the hypervisor.

This assumption is the level-specific instantiation of R2 (page 9) from Chapter 1.

Suppose v is the value obtained by the hypervisor at the primary from executing input instruction i. Let $\mathit{simulate}(\langle \mathrm{PC}, Q, R \rangle, i, v)$ be a function that transforms the virtual machine state $\langle \mathrm{PC}, Q, R \rangle$ as if input instruction i were executed returning (and updating the virtual machine state by) v.

Suppose the state of the virtual machine at both the primary and the backup is given by $S \equiv \langle \mathrm{PC}, Q, R \rangle$. If $\mathit{simulate}(\langle \mathrm{PC}, Q, R \rangle, i, v)$ is applied by the hypervisor at the primary and backup, then if $simulate(\langle \mathrm{PC}, Q, R\rangle, i, v) = \langle \mathrm{PC}', Q', R'\rangle$ holds, then $\langle \mathrm{PC}, Q, R\rangle \xrightarrow{i} \langle \mathrm{PC}', Q', R'\rangle$ holds for both the primary and backup. Thus IS2 is satisfied when *simulate* is applied to identical states at the primary and the backup. This can be accomplished by the primary obtaining the value v and communicating it to the backup.

2.2.3 Next Action: Synchronous Exception (Trap)

The semantics of an action x may cause one or more exceptions that occur as a result of executing x in certain states. At the virtual machine interface, such a synchronous exception is called a *trap*. In general, the effect of a trap is to record the current instruction (and/or its address and additional state information) and begin executing a *trap handler*, which is an instruction sequence. Typical examples of a trap include an instruction attempting to divide by zero or a memory access violation.

Suppose execution of some action x results in state $\langle \mathrm{PC}, Q, R\rangle$. We must assume that:

A4: The hardware uses a deterministic algorithm to select which trap $s \in Q$ to deliver.

Assumptions A1 and A4 are the level-specific instantiation of R1 (page 9) from Chapter 1 and are satisfied by most current hardware architectures.

So, if the primary and backup are both in state $S \equiv \langle \mathrm{PC}, Q, R\rangle$ in which $Q \neq \{\}$ holds, then the next action $s \in Q$ will be the same at both the primary and backup, again satisfying IS1.

For the primary $\langle \mathrm{PC}, Q, R\rangle \xrightarrow{s} \langle \mathrm{PC}_P, Q_P, R_P\rangle$ holds, and for the backup $\langle \mathrm{PC}, Q, R\rangle \xrightarrow{s} \langle \mathrm{PC}_B, Q_B, R_B\rangle$ holds. If we assume that:

A5: All traps are deterministic in the state of the virtual machine.

then, once again, $\mathrm{PC}_P = \mathrm{PC}_B$, $Q_P = Q_B$, and $R_P = R_B$ hold, satisfying IS2. Note that A5 does not require that trap handlers be deterministic — just that which trap handler will be invoked is computed determinstically.

Assumptions A2 and A5 are the level-specific instantiation of R4 (page 10) from Chapter 1.

2.2.4 Next Action: Asynchronous Exception (Interrupt)

At the virtual machine interface, asynchronous exceptions that may be delivered by the hypervisor to the virtual machine are known as *interrupts*. An interrupt is a change in the flow of control caused not by the current computation but by something else (typically related to the environment). Interrupts differ from traps because interrupts are asynchronous to the computation and traps are not.

The key to continuing identical computations when interrupts are possible is to deliver the same interrupts at the same position in the computation of both the primary and the backup. This results in the same state changes at each point in the computation.

In the VFT protocol, the hypervisor at the primary receives all interrupts. These interrupts are translated into *virtual interrupts* for the virtual machine and must be delivered at both the primary and the backup. Once the virtual interrupt has been computed, where the interrupt is delivered in the action sequence is the only non-deterministic choice that the hypervisor at the primary must make. The hypervisor at the primary can convey enough information to the backup so that the hypervisor there can then deliver the same virtual interrupt at the same position.

Given an interrupt a arriving at the hypervisor of the primary, let function $\hat{a}$ specify how the virtual machine state is transformed as a result of delivering the corresponding virtual interrupt to the virtual machine. For example, when $\hat{a}$ is applied to virtual machine state S, $\hat{a}$ might add the virtual interrupt corresponding to a to the set of pending interrupts in S and set the completion status word in S to the completion status of a. The virtual interrupt corresponding to a is delivered to the virtual machine at some position in the action sequence simply by applying $\hat{a}$ to the state of the virtual machine at the appropriate point in the action sequence. The hypervisor at the primary can communicate this position and an encoding of $\hat{a}$ to the hypervisor at the backup, thereby allowing the backup to reproduce the action sequence.

Suppose the state of the virtual machine at both the primary and the backup is given by $S \equiv \langle \mathrm{PC}, Q, R \rangle$. If the same function $\hat{a}$ is applied to that state by the hypervisor at the primary and backup, then if $\hat{a}(\langle \mathrm{PC}, Q, R \rangle) = \langle \mathrm{PC}', Q', R' \rangle$ holds, then $\langle \mathrm{PC}, Q, R \rangle \xrightarrow{a} \langle \mathrm{PC}', Q', R' \rangle$ holds for both the primary and backup. Thus IS2 is satisfied when $\hat{a}$ is applied to identical states at the primary and the backup.

We are left with the problem of how to deliver the same interrupts at the same point in the computation at the primary and the backup. The VFT solution to this problem is as follows. The virtual machine instruction sequence is partitioned into fixed length blocks called *epochs*. All interrupts that arrive at the hypervisor of the primary during an epoch are translated to virtual interrupts and buffered until the next epoch boundary is reached. At the epoch boundary, the same virtual interrupts are delivered in the same order by the hypervisor at both the primary and the backup.

```
invoked when: Virtual Machine executes input instruction inputInstr at Primary

input
  vmState : VMState;                -- virtual machine state at time of input instruction
  inputInstr : InputInstruction;    -- input instruction attempted by Virtual Machine

global var
  currentEpoch : integer;

local var
  inputVal : InputValue;            -- value returned by input instruction

inputVal := hwInstruction(inputInstr);
send ⟨"InputValueMsg", currentEpoch, inputVal⟩ to Backup;
WaitStable(Backup);
vmState := SimInstruction(vmState, inputInstr, inputVal);
ResumeVirtMach(vmState);
```

Figure 2.2: Input Instruction Processing at Primary

The VFT protocol thus requires that the virtual machine transfer control to the hypervisor at the same point in the computation at both the primary and backup. This is done using a trap. One implementation of this *epoch trap* — using the HP–PA recovery register — is discussed in greater detail in Chapter 4 and leads to the assumption:

A6: The target architecture includes some mechanism to implement epoch traps.

2.2.5 Instruction Sequence Coordination Pseudo-code

Primary Processing

Figure 2.2 gives pseudo-code for the processing required by the hypervisor at the primary when the virtual machine attempts to execute an input instruction. As with

invoked when: hardware interrupt at Primary

input
vmState : VMState; -- virtual machine state at time of interrupt
hyperInterrupt : HWIntStruct; -- hardware interrupt information

global var
currentEpoch : **integer**;
Outstanding : **integer** $\longrightarrow$ VirtIntStruct*;

local var
virtualInterrupt : VirtIntStruct; -- virtual interrupt information

virtualInterrupt := hyperToVirtualInt(hyperInterrupt);
Outstanding[currentEpoch] := Outstanding[currentEpoch] $\oplus$[a] virtualInterrupt;
send ⟨"InterruptMsg", currentEpoch, virtualInterrupt⟩ **to** Backup;
ResumeVirtMach(vmState);

[a]The $\oplus$ operator appends either a singleton element or a queue of elements to an existing queue, returning the new queue.

Figure 2.3: Interrupt Processing at Primary

all incursions into the hypervisor, the current state of the virtual machine, vmState, is an input. The input instruction of the virtual machine inputInstr is also an input.

The given input instruction is executed by the hypervisor by calling hwInstruction, returning the input instruction value in inputVal. Next, a copy of the input instruction value, tagged with the current epoch, is sent to the backup. The primary then invokes WaitStable(Backup) to wait for all information sent to the backup to be received. The virtual machine state is then updated with the input instruction value according to the semantics of the inputInstr instruction. Finally, the primary hypervisor resumes execution of the virtual machine using vmState as an argument to ResumeVirtMach.

Figure 2.3 gives pseudo-code for the processing required by the hypervisor at the primary when an interrupt arrives that must be delivered to the virtual machine.

A structure describing the hypervisor interrupt hyperInterrupt is an input. The hypervisor interrupt must be translated into a structure that enables a corresponding virtual interrupt to be delivered at a later point in the computation. This structure is buffered by placing it on a queue of outstanding virtual interrupts for the current epoch (Outstanding[currentEpoch] in Figure 2.3). The primary then sends a copy of the virtual interrupt structure along with the epoch of arrival to the backup. Finally, the primary hypervisor resumes execution of the virtual machine.

The virtual interrupt structure must include enough information to later deliver the virtual interrupt to the virtual machine. Delivering an interrupt resulting from a disk block read, for example, might include setting a bit in the PSW to indicate the interrupt is pending, setting a control register to the level of the interrupt, and updating a buffer with the contents of the disk block read.

In Figure 2.4, pseudo-code for epoch-boundary processing by the hypervisor at the primary is presented. First, the primary sends the current epoch number to the backup. The primary then invokes WaitStable(Backup) to wait for all information sent to the backup to be received. While the primary waits, however, additional interrupts may arrive, causing the interrupt processing of Figure 2.3. These interrupts are buffered for delivery at the end of the upcoming epoch (currentEpoch). Interrupts must be disabled for the interval beginning with sending the epoch number message and ending with creating a new queue and setting the currentEpoch variable to use that queue. This means that an interrupt that arrives before interrupts are disabled is queued for the epoch just completed; an interrupt that arrives after that point is queued for the next epoch.

```
invoked when: Epoch Trap at Primary

input
  vmState : VMState;           -- virtual machine state at time of epoch trap

global var
  currentEpoch : integer;
  Outstanding : integer ⟶ VirtIntStruct*;

local var
  virtualInterrupt : VirtIntStruct; -- virtual interrupt information

with intrpts disabled do
  send ⟨"EpochNumberMsg", currentEpoch⟩ to Backup;
  currentEpoch := currentEpoch + 1;
  Outstanding[currentEpoch] := AllocateQueue();
  od
WaitStable(Backup);
while not empty(Outstanding[currentEpoch − 1]) do
  virtualInterrupt := dequeue(Outstanding[currentEpoch − 1]);
  vmState := DeliverVirtInt(vmState, virtualInterrupt);
  od
FreeQueue(Outstanding[currentEpoch − 1]);
ScheduleNextEpochTrap();
ResumeVirtMach(vmState);
```

Figure 2.4: Epoch-Boundary Processing at Primary

Once the backup has received all data for the epoch just concluded, the primary may deliver the interrupts that were saved for this epoch boundary; they are in Outstanding[currentEpoch−1]. The primary then frees the storage used by this queue, as it is no longer needed. Next, the primary uses ScheduleNextEpochTrap() to schedule the next epoch trap, thereby defining the next epoch boundary. Finally, the primary resumes execution of the virtual machine.

Backup Processing

Figure 2.5 depicts processing at the backup hypervisor when a message from the primary is available for receipt. When the backup receives a message of type "InputValueMsg" from the primary, it buffers the input instruction value m.inputVal on a queue of such values for the epoch specified in the message. If this is the first input instruction for the given epoch, no queue will yet exist, so one is first created.

Likewise, when the backup receives a message of type "InterruptMsg" from the primary, it buffers the virtual machine interrupt structure m.virtualInterrupt on a queue of such structures for the given epoch.

Comm (page 19) ensures that receive returns messages in the order sent, so all input instruction values and virtual machine interrupt structures are ordered as sent by the primary, so they may be returned to the virtual machine in the proper order.

When the hypervisor at the backup receives an epoch number message from the primary, it updates its record of the current epoch at the primary primaryEpoch. (This update may terminate a concurrent spin wait at the start of the epoch-boundary processing at the backup presented in Figure 2.7 and discussed below.)

```
invoked when: message available for receipt at Backup

input
  vmState : VMState;            -- virtual machine state

global var
  primaryEpoch : integer;
  Outstanding : integer ⟶ VirtIntStruct*;
  Values : integer ⟶ InputValue*;

local var
  m : ⟨type : "InputValueMsg", epochNum : EpochNumber, inputValue : InputValue⟩;
                                -- for incoming message "InputValueMsg"
  m : ⟨type : "InterruptMsg", epochNum : EpochNumber, virtualInterrupt : VirtIntStruct⟩;
                                -- for incoming message "InterruptMsg"
  m : ⟨type : "EpochNumberMsg", epochNum : EpochNumber⟩;
                                -- for incoming message "EpochNumberMsg"

receive(m);
case m.type of
  "InputValueMsg": if not defined(Values[m.epochNum]) then
                      Values[m.epochNum] := AllocateQueue();
                   Values[m.epochNum] := Values[m.epochNum] ⊕
                                m.inputValue;

  "InterruptMsg": if not defined(Outstanding[m.epochNum]) then
                     Outstanding[m.epochNum] := AllocateQueue();
                  Outstanding[m.epochNum] := Outstanding[m.epochNum] ⊕
                                m.virtualInterrupt;

  "EpochNumberMsg": primaryEpoch := m.epochNum;
esac
ResumeVirtMach(vmState);
```

Figure 2.5: Receive Processing at Backup

invoked when: Virtual Machine executes input instruction inputInstr at Backup

input
  vmState : VMState; – – virtual machine state at time of input instruction
  inputInstr : InputInstruction; – – input instruction attempted by Virtual Machine

global var
  currentEpoch : **integer**;
  Values : **integer** ⟶ InputValue*;

local var
  inputVal : InputValue; – – value returned by input instruction

```
inputVal := dequeue(Values[currentEpoch]);
vmState := SimInstruction(vmState, inputInstr, inputVal);
ResumeVirtMach(vmState);
```

Figure 2.6: Input Instruction Processing at Backup

Input instruction processing at the backup is presented in Figure 2.6. The appropriate input instruction value required for this input instruction and received from the primary will be the next value on the Values[currentEpoch] queue. The hypervisor can simply simulate the input instruction, returning that value, and resume the virtual machine.

Virtual machine execution at the backup can never be allowed to get ahead of virtual machine execution at the primary. Thus, we do not allow the backup to proceed into epoch i until the primary has completed epoch i. This is accomplished with the spin loop beginning the epoch-boundary processing presented in Figure 2.7.[2]

During this wait, the backup may continue to receive messages about virtual interrupts from the primary, causing the processing of Figure 2.5. The wait is terminated,

[2]Assume that, initially, the primary and backup's currentEpoch variable is -1 (as is the backup's primaryEpoch variable). The primary begins by incrementing it's currentEpoch and proceeding into epoch 0. The backup begins execution in this Epoch Trap code.

```
invoked when: Epoch Trap at Backup

input
  vmState : VMState;              -- virtual machine state at time of Epoch Trap

global var
  currentEpoch : integer;
  Outstanding : integer ⟶ VirtIntStruct*;
  Values: integer ⟶ InputValue*;

local var
  virtualInterrupt : VirtIntStruct; -- hardware interrupt information

while (primaryEpoch ≤ currentEpoch) do
  skip;
  od
while defined(Outstanding[currentEpoch] cand[a] not empty(Outstanding[currentEpoch]) do
  virtualInterrupt := dequeue(Outstanding[currentEpoch]);
  vmState := DeliverVirtInt(vmState, virtualInterrupt);
  od
FreeQueue(Outstanding[currentEpoch]);
FreeQueue(Values[currentEpoch]);
currentEpoch := currentEpoch + 1;
ScheduleNextEpochTrap();
ResumeVirtMach(vmState);
```

[a] **cand** denotes conditional and — the second term is not evaluated if the first term is **false**.

Figure 2.7: Epoch-Boundary Processing at Backup

as described above, when the backup receives and processes the "EpochNumberMsg" message, updating its primaryEpoch variable. The backup then delivers any buffered interrupts, increments its current epoch number, and begins the next epoch.

Our protocol requires that delivery of all virtual machine interrupts be handled by the hypervisor. Hypervisor intervention between interrupt arrival and virtual interrupt delivery is only possible if we assume:

A7: All virtual interrupts are implemented (i.e., not inherited) by the hypervisor.

This assumption is satisfied by the existence of the virtual machine and will be discussed in greater detail in the following chapter. This assumption is the level-specific instantiation of R1 (page 9) from Chapter 1.

2.2.6 Instruction Sequence Example

An example will illustrate the protocol for coordinating the primary and the backup. Consider the scenario depicted in Figure 2.8. The primary begins epoch i in state S.

During the epoch, an input instruction INPUT r is executed updating r (some part of the state of the virtual machine) with input instruction value v. A message with the value v is sent to the backup and the primary waits until it knows the message has been received before proceeding with the epoch. Termination of this wait is signified in Figure 2.8 by a dashed arrow from the backup to the primary. Also during the course of the epoch, interrupts a_j and a_{j+1} arrive at the primary, but are not delivered to the virtual machine. Instead, they are translated to virtual interrupts, buffered, and a copy of each virtual interrupt sent to the backup.

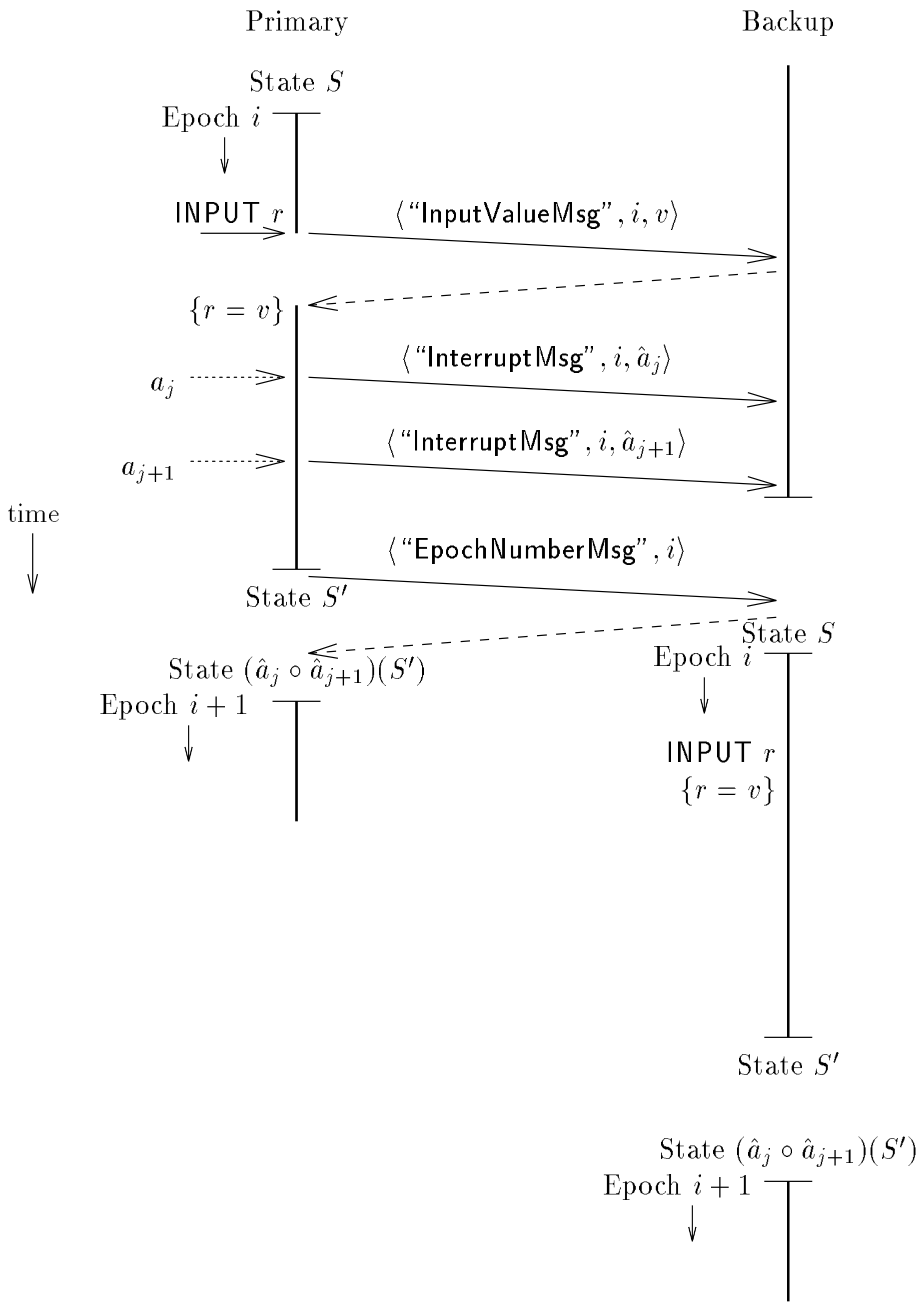


Figure 2.8: Instruction Sequence Example

At the end of the epoch, the primary is in state S', computed from the state S at the start of the epoch and based on the sequence of instructions comprising the epoch. The primary hypervisor sends the current epoch number i to the backup. Then the primary waits for all information that was sent to the backup to be received. After delivery of buffered virtual interrupts at the primary, the state of the primary is $(\hat{a}_j \circ \hat{a}_{j+1})(S')$. The primary then proceeds with epoch $i+1$.

When the hypervisor at the backup receives input instruction value v for epoch i from the "InputValuesMsg" message, it buffers it for later use. Likewise, when the backup receives the copy of interrupts a_j and a_{j+1} for epoch i from the primary, it buffers them as well.

Once the backup receives the "EpochNumberMsg" message for epoch i, it may begin epoch i. The backup also begins epoch i in state S. The backup returns the saved input instruction value v when input instruction INPUT r is encountered during the epoch. Regardless of any other activity, the backup will finish epoch i in state S'. The backup then delivers the buffered interrupts and begins epoch $i+1$, like the primary, in state $(\hat{a}_j \circ \hat{a}_{j+1})(S')$.

2.3 The Environment

Interactions between virtual machine replicas and the environment must be controlled so that the ensemble of replicas appears to the environment as if it were a single machine, even in the case of a failure. According to the I/O model given at the beginning of this chapter, the interaction between a virtual machine and the environment includes input instructions, interrupts, and output instructions. For example, an output instruction might be a disk read request that results in the transfer of a data block to memory followed by an interrupt indicating that the transfer completed. An input instruction might be a read of a hardware clock.

We now examine the interaction between the environment and the ensemble of virtual machine replicas with respect to the VFT protocol. There are three possible cases involving at most one failure — no failures, failure of the primary, and failure of the backup. The interaction when there are no failures and when the primary has failed are discussed below. The case where the backup fails simply requires that the primary no longer wait for the backup to receive information and is not discussed further.

2.3.1 No Failures

The primary is responsible for all interaction with the environment. Input instructions, output instructions, and resulting interrupts are handled by the hypervisor at the primary. The primary hypervisor allows input instructions to be executed and passes to the I/O controller every virtual machine output instruction. At the hypervisor of the backup, input instructions and output instructions are not passed along to the environment; this ensures that a backup does not interact with the environment.

According to the I/O model, the I/O controller will send completions to the hypervisor at the primary, since the hypervisor at the primary is the source of all requests it receives. Thus, given no failures, the ensemble of virtual machine replicas interacts with the environment as if it were a single machine, because the primary handles all such interaction.

Primary

invoked when: Virtual Machine executes output instruction virtualRequest

input

```
  vmState : VMState;              -- virtual machine state at time of request
  virtualRequest : HWRequest;     -- virtual request information
```

global var

```
  VirtualRequests : set of HWRequest;
```

local var

```
  hyperRequest : HWRequest;       -- hardware request information

VirtualRequests := VirtualRequests ∪ {virtualRequest};
hyperRequest := virtualToHyperRequest(virtualRequest);
hwRequest(hyperRequest);
ResumeVirtMach(vmState);
```

Backup

invoked when: Virtual Machine executes output instruction virtualRequest

input

```
  vmState : VMState;              -- virtual machine state at time of request
  virtualRequest : HWRequest;     -- virtual request information
```

global var

```
  VirtualRequests : set of HWRequest;

VirtualRequests := VirtualRequests ∪ {virtualRequest};
ResumeVirtMach(vmState);
```

Figure 2.9: Output Instruction Processing

Figure 2.9 shows the processing by a primary and backup in response to an output instruction by the virtual machine. At both primary and backup, the output instruction request virtualRequest is recorded in a set VirtualRequests of outstanding output instruction requests. The primary translates this virtual request into hyperRequest, an output instruction for the hardware. This request is submitted by executing hwRequest. Execution of the virtual machine is then resumed.

VirtualRequests is used when a failure occurs; we discuss this in Section 2.3.2. This set is kept current with the outstanding virtual requests by removing a virtual request from the set when its associated completion is delivered during epoch-boundary processing.

Hypervisor intervention at each output instruction is required for the primary to handle all interaction with the environment and for the backup to record the ongoing interaction. Thus we assume:

A8: All virtual machine output instructions are implemented by the hypervisor.

This assumption is the level-specific instantiation of R2 (page 9) from Chapter 1.

2.3.2 Failure of Primary

If the primary fails, then the backup must take over interacting with the environment. In addition, the backup must resolve any interaction ongoing at the time of failure. The interval beginning with the failure of the primary and ending when the backup has become the new primary is called *failover*.

When the primary fails, the backup knows the epoch during which the failure occurred from the last epoch number received from the primary. We call the epoch of the primary's failure the *epoch of failure*. Since, according to Comm, messages are received in the order sent, the backup must receive all virtual interrupt messages and input instruction value messages for epoch $\mathsf{primaryEpoch}$ (see Figure 2.5) and the primary must have completed that epoch. The epoch of failure is, therefore, $\mathsf{primaryEpoch} + 1$. The failure of the primary after completing epoch $\mathsf{primaryEpoch}$ and before beginning epoch $\mathsf{primaryEpoch} + 1$ is treated as a failure in epoch $\mathsf{primaryEpoch} + 1$. Once the interrupts for epoch $\mathsf{primaryEpoch}$ have been delivered by the backup, the state of the virtual machine at the backup is identical to the state of the virtual machine at the primary as of the beginning of the epoch of failure. The backup also has a record of the outstanding requests as of the beginning of the epoch of failure as stored in $\mathsf{VirtualRequests}$.

The failure of the primary, however, results in uncertainty at the backup about requests made at the primary during the epoch of failure and their corresponding completions. Consider the example shown in Figure 2.10. Suppose that the primary fails at some point during epoch i. Request A has been made prior to the beginning of epoch i, and epoch i is begun in state S. Suppose that during epoch i, an output instruction is executed resulting in request B. Further, suppose that epoch i does not include any input instructions. Later in the epoch, an output instruction would be executed resulting in request C. Before the primary fails, it receives interrupt a_A indicating completion of request A. The end of the epoch (if completed) would leave the virtual machine in state S'.

The hypervisor at the backup must cope with two difficulties:

1. Given the information available to it, the hypervisor at the backup has no way

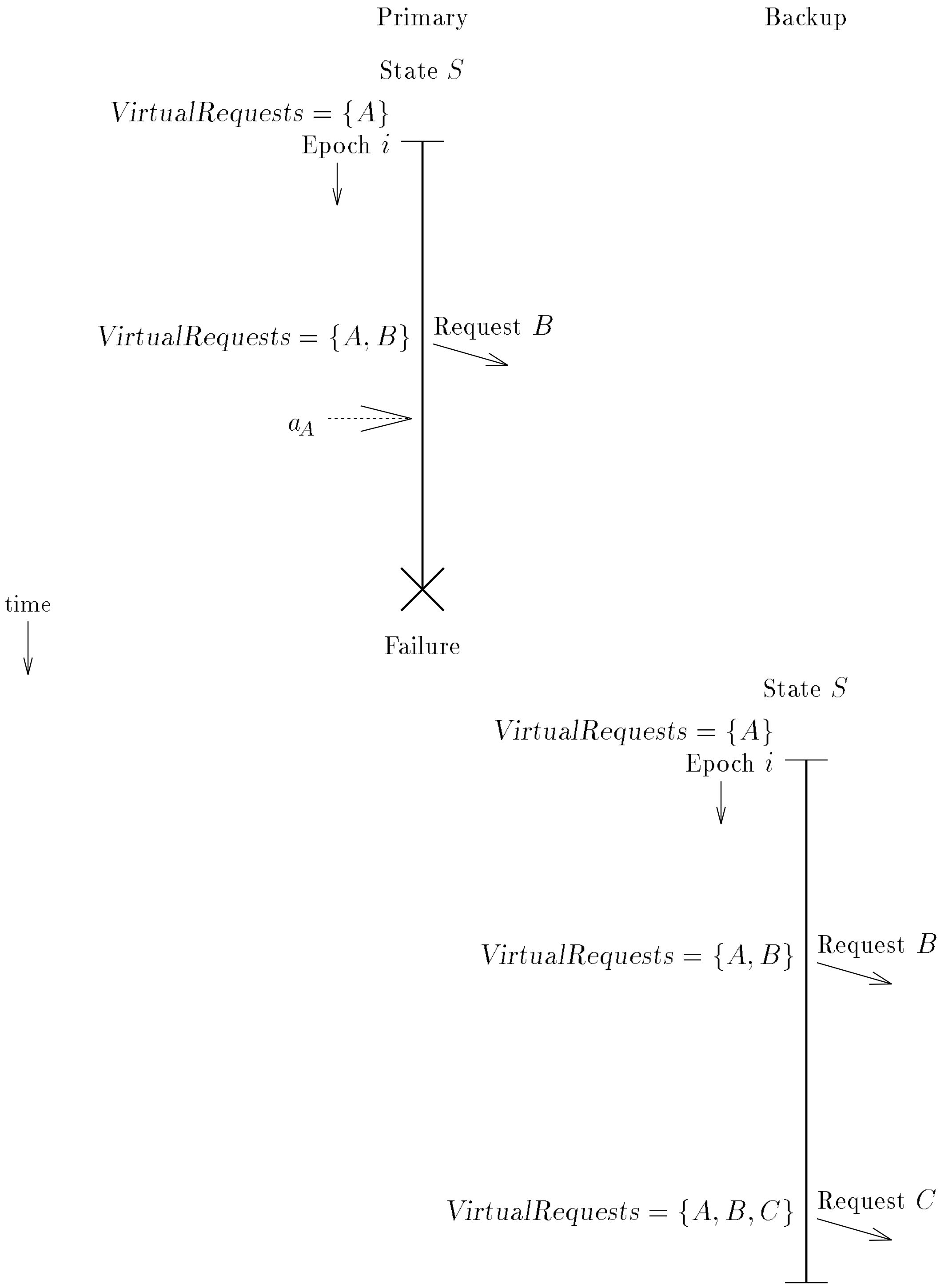


Figure 2.10: Failover Example 1

to determine how far the primary progressed in the epoch of failure. There are three possibilities:

(a) The primary failed prior to making request B, so there are no unrecorded outstanding requests at the time of the failure.

(b) The primary failed at some point between making requests B and C, so the recorded outstanding requests at the time of the failure include A and B.

(c) The primary failed late in the epoch and succeeded in making both requests B and C, so the recorded outstanding requests at the time of the failure include A, B, and C.

2. The backup has no way to determine whether all of the corresponding completions arrived at the hypervisor of the primary.

A naive solution to resolving the uncertainty about the primary's progress would be for the backup to take over as primary (and begin interacting with the environment) at the beginning of epoch i, the epoch of failure. However, since the backup might not have received the relayed interrupt a_A indicating completion of request A, the backup could not deliver the appropiate interrupt completion for request A. This violates IO2 (page 18) of the I/O model wherein each request terminates with a completion. Also, the backup would repeat request B, which had already been made by the primary. The backup virtual machine would not know the request had been repeated and, if the request were not idempotent, would violate the correctness criterion.

Another naive solution would be for the backup to wait until the end of the epoch of failure before taking over as primary. In this case, request C would never be seen by the environment, and there would be no completion for this request. This also violates the correctness criterion. Also problematic is delivering completion interrupts for requests A and B (submitted by the primary to the environment).

The ideal solution would be for each request to be passed along to the environment exactly once, so the backup would not repeat requests successfully made and completed by the primary. Unfortunately, the problem of the backup not knowing which requests were made by the primary (and their corresponding completions) at the time of failure is unsolvable.

The key to the solution employed by the VFT protocol is to exploit an uncertainty already inherent in interacting with a real environment. Elements of the environment are not always dependable. For example, when processing a request, a disk may experience a transient fault, leading to an uncertain outcome — either the request is processed or not, but it is not known which. Likewise, a packet transmission over a network may not arrive at its destination; network protocols are designed to detect this and retransmit such packets. These transient I/O faults are a normal consequence of interacting with an imperfect environment and are included in the I/O model. In particular, when a transient I/O fault occurs, the completion status for the request is a transient I/O error.

The VFT protocol uses transient I/O errors to move to the virtual machine some of the burden of managing a failover. The virtual machine decides the appropriate action to take during a failover. To accomplish this, the hypervisor at the backup returns "transient I/O error" completions for all requests still outstanding at the point when the backup takes over as the new primary. This newly promoted primary may then begin interacting with the environment.

To satisfy the correctness criterion, the point at which the backup takes over as the primary, called the *takeover point*, must be some point in the virtual machine computation that follows all requests the primary might have made. Whenever the hypervisor at the primary uses WaitStable to ensure that all messages have been received by the backup, the backup can replay the virtual machine computation that preceeded the WaitStable. Incocations of WaitStable occur at each epoch boundary and on each input instruction. By A2 (the determinism of non-input instructions), the backup can extend the virtual machine computation beyond the primary's last WaitStable until either the first input instruction not executed and relayed by the primary or the end of the epoch of failure, whichever comes first. This defines the takeover point and this point is guaranteed to follow all requests the primary might have made. The virtual machine computation performed by the primary is a prefix of the virtual machine computation performed by the backup.

Given a virtual request A, let function $\tilde{A}$ define the transformation in virtual machine state that occurs from delivering notification of a transient I/O error for request A. At the takeover point, the hypervisor at the backup delivers transient I/O error interrupts for all outstanding requests in VirtualRequests (regardless of their actual completion status). The revised processing for handling takeover at the end of the epoch of failure at the backup is given in Figure 2.11.

Since the primary may have been more than one epoch ahead of the backup, the condition of the **if** statement ensures that the takeover processing occurs at the end of the epoch of failure. Note in Figure 2.11 that the backup returns the true completion

```
invoked when: Epoch Trap at Backup

input
  vmState : VMState;              -- virtual machine state at time of epoch trap

global var
  currentEpoch : integer;
  Outstanding : integer ⟶ VirtIntStruct*;
  Values : integer ⟶ InputValue*;
  VirtualRequests : set of HWRequest;

local var
  virtualInterrupt : VirtIntStruct;  -- hardware interrupt information
  virtualRequest : HWRequest;        -- virtual request information

while (primaryEpoch ≤ currentEpoch and not failed(Primary)) do
  skip;
  od
while defined(Outstanding[currentEpoch]) cand not empty(Outstanding[currentEpoch]) do
  virtualInterrupt := dequeue(Outstanding[currentEpoch]);
  virtualRequest := InterruptToRequest(virtualInterrupt);
  vmState := DeliverVirtInt(vmState, virtualInterrupt);
  VirtualRequests := VirtualRequests − {virtualRequest};
  od
if (currentEpoch = primaryEpoch +1) then      -- takeover point
  while not empty(VirtualRequests) do
    vmState := DeliverErrorInt(vmState, virtualRequest);
    VirtualRequests := VirtualRequests − {virtualRequest};
    od
  FreeQueue(Outstanding[currentEpoch]);
  FreeQueue(Values[currentEpoch]);
  currentEpoch := currentEpoch + 1;
  Outstanding[currentEpoch] := AllocateQueue();
  ScheduleNextEpochTrap();
  BecomePrimary();
  ResumeVirtMach(vmState);
else
  FreeQueue(Outstanding[currentEpoch]);
  FreeQueue(Values[currentEpoch]);
  currentEpoch := currentEpoch + 1;
  ScheduleNextEpochTrap();
  ResumeVirtMach(vmState);
fi
```

Figure 2.11: Revised Backup Epoch-Boundary Processing

for any completions it has successfully received before the failure of the primary. In our example, if request A results in interrupt a_A, and $\hat{a}_A$ is sent successfully to the backup before the primary fails, then the backup would update the state of the virtual machine to $(\tilde{C} \circ \tilde{B} \circ \hat{a}_A)(\mathsf{vmState})$ before proceeding into epoch $i+1$.

The first input instruction after a failure for which the value returned by the primary has not been received by the backup defines the other possible takeover point. Figure 2.12 presents the revised input instruction processing to handle this takeover case. Note that once the backup has taken over as the primary, it begins interacting with the environment and must actually perform the input instruction and return a value to the virtual machine.

Once the hypervisor at the backup has taken over and has delivered the transient I/O error interrupts for any remaining outstanding requests, the virtual machine can decide the action to take and, if appropriate, retry the request. Interaction with the environment is then consistent with what the virtual machine would see if the transient I/O errors had actually occurred. And, the virtual machine at the backup has no way of knowing that the transient I/O error didn't occur.

If the environment could observe that a successful request was subsequently resubmitted by the virtual machine, the environment could detect that a failover had occurred and that the interaction is not consistent with that of a single fault-tolerant machine. Consider the case of request A and its completion a_A in the example of Figure 2.10. The environment has seen request A and delivered a successful completion. Suppose that $\hat{a}_j$ was not relayed to the backup (because of the timing of the primary's failure). The hypervisor at the backup will return a transient I/O error for request A. Further, suppose that in response to that error condition, the virtual

```
invoked when: Virtual Machine executes input instruction inputInstr at Backup

input
  vmState : VMState;                  -- virtual machine state at time of input instruction
  inputInstr : InputInstruction;      -- input instruction attempted by Virtual Machine

global var
  currentEpoch : integer;
  Values : integer ⟶ InputValue*;
  Outstanding : integer ⟶ VirtIntStruct*;
  VirtualRequests : set of HWRequest;

local var
  inputVal : InputValue;              -- value returned by input instruction
  virtualInterrupt : VirtIntStruct;   -- hardware interrupt information
  virtualRequest : HWRequest;         -- virtual request information

if (Values[currentEpoch] = ∅)         -- takeover point
  while defined(Outstanding[currentEpoch]) cand
        not empty(Outstanding[currentEpoch]) do
    virtualInterrupt := dequeue(Outstanding[currentEpoch]);
    virtualRequest := InterruptToRequest(virtualInterrupt);
    vmState := DeliverVirtInt(vmState, virtualInterrupt);
    VirtualRequests := VirtualRequests − {virtualRequest};
    od
  while not empty(VirtualRequests) do
    vmState := DeliverErrorInt(vmState, virtualRequest);
    VirtualRequests := VirtualRequests − {virtualRequest};
    od
  inputVal := hwInstruction(inputInstr);
  vmState := SimInstruction(vmState, inputInstr, inputVal);
  BecomePrimary();
  ResumeVirtMach(vmState);
else
  inputVal := dequeue(Values[currentEpoch]);
  vmState := SimInstruction(vmState, inputInstr, inputVal);
  ResumeVirtMach(vmState);
fi
```

Figure 2.12: Revised Backup Input Instruction Processing

machine at the backup retries request A. The environment would then see request A twice, even though it returned successful completion on the first instance. We preclude this by extending the I/O model as follows:

IO5: The environment cannot distinguish between a retry request of the virtual machine and a new request with the same operands.

In practice, this assumption is easily satisfied. In the case of a disk, a repeated write of some given data or read to the same disk block will leave the disk in the same state and — if successful — return the same completion status. Likewise for packet transmission and reception over a local network device.

2.3.3 Input Instruction Optimization

Our handling of input instruction values can be optimized. Suppose the primary did not send the InputValueMsg message when processing an input instruction but buffered the input instruction value and later sent a batch of input instruction values to the backup. The major advantage of this scheme would be reduced communication overhead: the primary would not have to wait (the WaitStable(Backup) in Figure 2.12) for a message to be received after each input instruction before continuing.

Since the primary must execute WaitStable(Backup) at every epoch boundary anyway, input instruction values could be sent at the end of each epoch. The primary would then incur only a single wait per epoch. Since the backup does not begin epoch i until after the primary has completed epoch i, sending the batch of input instruction values before (or with) the EpochNumberMsg message guarantees that the backup has all the information it requires to complete epoch i and end that epoch in the same state as the primary. Also, input instruction values tend to be small (generally a single 32-bit word per value), so a single message usually suffices for all the input instruction values returned during an epoch.

The drawback to this optimization is the new source of uncertainty it creates when a failure occurs. Consider the example shown in Figure 2.13. Suppose the primary fails during epoch i. Request A has been made prior to the beginning of epoch i, and epoch i is begun in state S. Suppose that during epoch i, the primary executes input instructions returning values x and y. Further, suppose that in epoch i an output instruction is executed resulting in request B. Note that the request B may be causally dependent on the input instruction value x returned from INPUT $r1$.

Once the primary fails, the backup must begin epoch i, but will not have received the input instruction values for epoch i. Starting epoch i in the same state S as the primary, the backup will reach the INPUT $r1$ instruction. Depending on the input instruction value the backup returns for the INPUT $r1$ instruction, the virtual machine at the backup may or may not make request B.

Previously, the virtual machine was guaranteed to get to request B by virtue of (i) the primary's WaitStable call following the send of the InputValueMsg at input instruction processing, and (ii) the determinism of the non-input instructions following that point. Under our new scheme, we can guarantee the backup can perform the same computation through all requests the primary might make before it fails only by requiring a request at the primary to flush the set of buffered input instruction values and call WaitStable before the request is issued to the environment.

This new scheme does not have as strong a property with respect to the computation executed at the primary and backup. The computation performed by the primary may no longer be a prefix of the computation performed by the backup. In

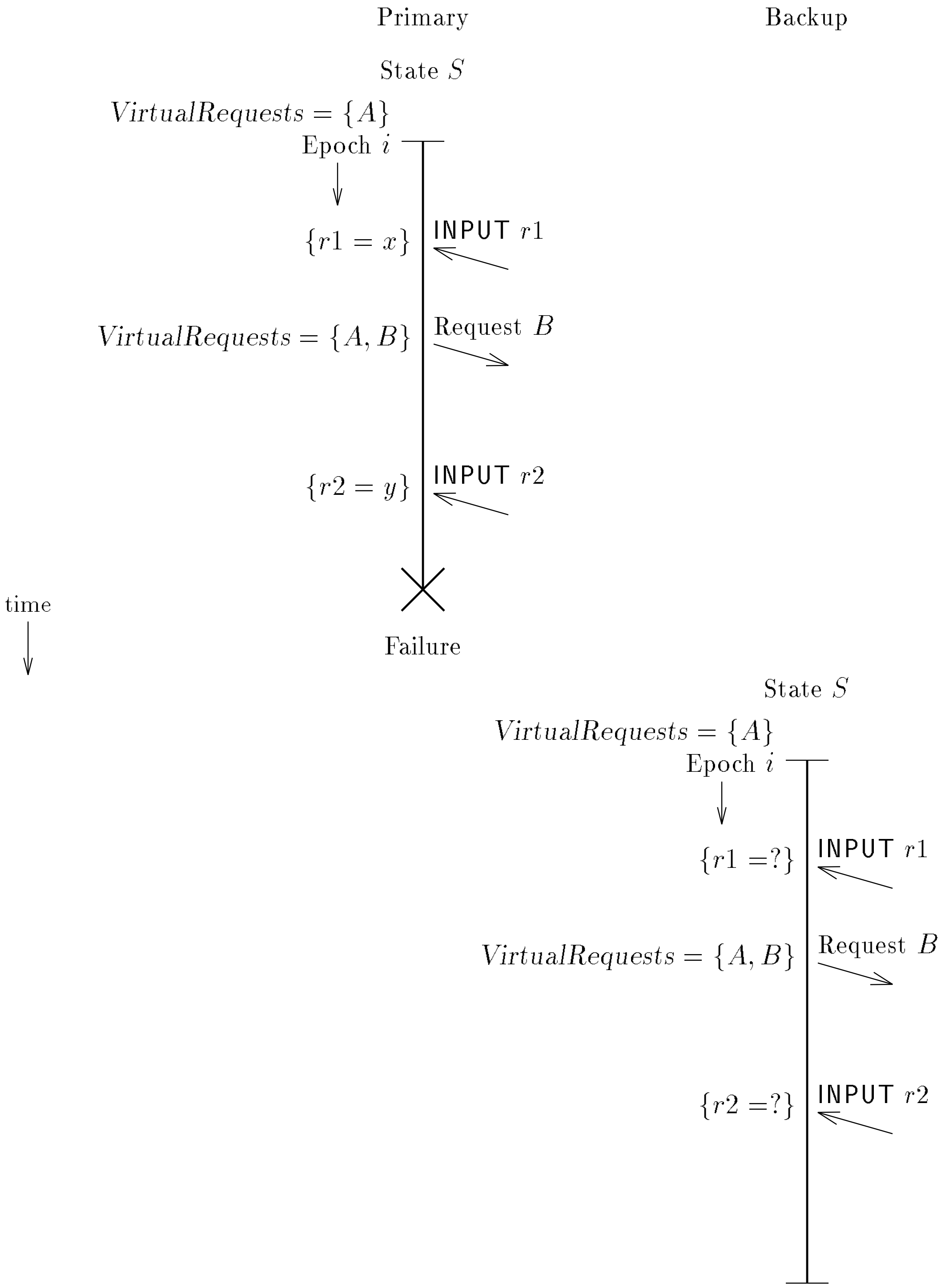


Figure 2.13: Failover Example 2

particular, the compuations may diverge at the first input instruction following the last request made by the primary in the epoch of failure. Again consider the example of Figure 2.13. The backup may return a different input instruction value than the primary did when it executes the INPUT $r2$ instruction. However, this divergence does not affect our correctness criterion. The value y returned at the primary cannot have become visible to the environment, so the backup may safely make its own non-deterministic choice.

Chapter 3

Virtualizing the HP Precision Architecture

The first step in building a prototype system to demonstrate the VFT protocol was to design and build a hypervisor. This chapter examines issues involved in implementing such a hypervisor for the HP Precision Architecture (HP–PA). We first discuss the general problem of implementing a hypervisor. We then apply this to the problem of implementing a hypervisor on the HP–PA. We conclude by discussing some of the problems unique to an HP–PA hypervisor.

3.1 Architecture of a Virtual Machine

Figure 3.1 shows levels and interfaces for a conventional computing system. The interface between the hardware and the operating system $\mathcal{L}_{HW}$ is the machine's architectural definition. This figure is similar to Figure 1.2, but shows how multiple

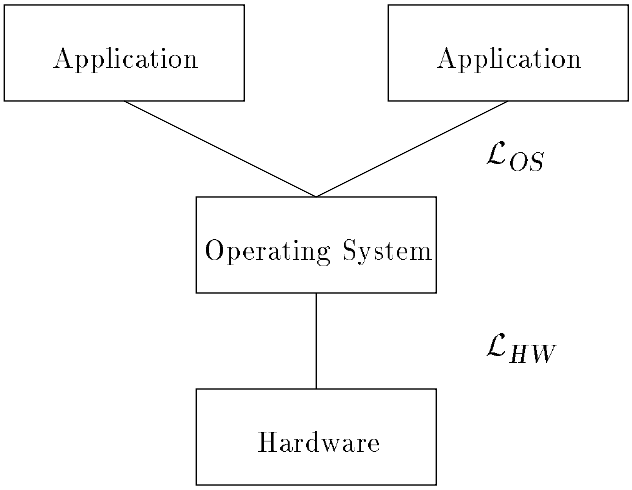


Figure 3.1: Conventional System

applications may use the same extended interface $\mathcal{L}_{OS}$ that is supported by the operating system.

When a hypervisor is interposed, the example architecture shown in Figure 3.2 is obtained. Here, multiple operating systems run above a single hypervisor. Each operating system may, in turn, support multiple applications. Note that each operating system may define its own extended interface ($\mathcal{L}_{OS_0}$ and $\mathcal{L}_{OS_1}$) for the applications running above it.

Each operating system and the software levels above comprises a virtual machine — an efficient, isolated replica of the real machine. VM_0 and VM_1 are the virtual machines in Figure 3.2. A hypervisor is a software layer with the following essential characteristic:

Interface Equivalence: The set of actions and the semantics of those actions are identical at $\mathcal{L}_{HW}$ and $\mathcal{L}_{VM}$.

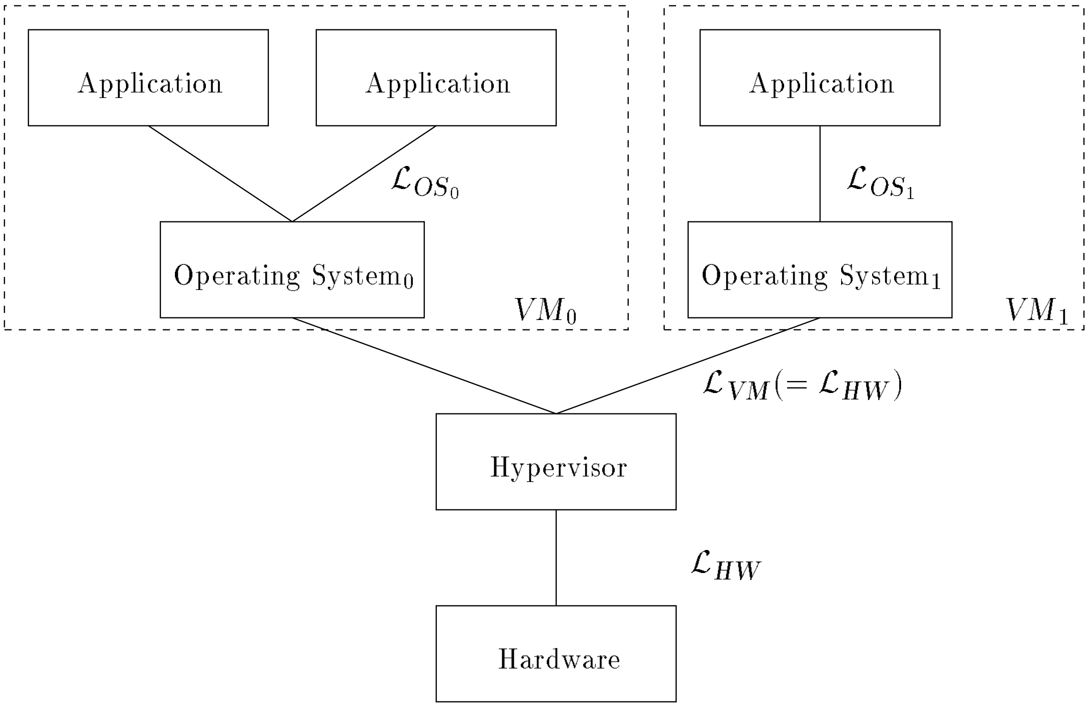


Figure 3.2: Virtual Machine Architecture

This characteristic is critical for building a transparent fault-tolerant system using the VFT protocol.

3.1.1 Interface Equivalence

If Interface Equivalence holds, then any program running on a virtual machine will behave identically, whether the hypervisor is present or not. This property thus provides the transparency we seek with VFT.

To allow realizable implementations of virtual machine architectures, the semantics of $\mathcal{L}_{HW}$ (and thus $\mathcal{L}_{VM}$) must be restricted slightly. These restrictions are necessary in order to support multiple virtual machines, where the processor and all resources of the computing system are multiplexed among the supported virtual machines and the hypervisor.

First, the semantics of no action of $\mathcal{L}_{HW}$ can depend on the execution time of that action. Programs that are sensitive to the timing of an instruction or sequence of instructions may no longer work in a virtual machine architecture because some actions may take longer to execute. Actions implemented by a hypervisor (as opposed to inherited by a hypervisor) typically take longer to execute when run in the virtual machine architecture than when run on the hardware directly. In addition, if multiple virtual machines are supported, execution of another virtual machine may be interleaved with the execution of the first virtual machine.

Second, the semantics of $\mathcal{L}_{HW}$ cannot define the number and extent of actual resources available on the system. As mentioned above, with a hypervisor, actual resources are multiplexed among the supported virtual machines and the hypervisor.

This multiplexing may cause the amount of a resource available to a given virtual machine to differ from what would be available with the hardware directly.

As a simple example of this restriction, suppose that the computing system of Figure 3.2 has a physical memory of size x. If the hypervisor itself occupied y of that memory, it might provide $(x-y)/2$ of physical memory to each of VM_0 and VM_1. (The hypervisor cannot provide a physical memory of size x to both VM_0 and VM_1.) If software running on VM_0 or VM_1 assumed the x of storage provided for by $\mathcal{L}_{HW}$, that assumption would be violated when run in the virtual machine architecture.

Note that these two restrictions are not significant:

- Software written for portability among a family of processors with the same architecture should not assume particular instruction timings anyway. This is because the members of such a family can exhibit a wide range of timing characteristics, depending on the speed and organization of the architecture implementation. Application programs should not make timing assumptions due to the effects of multiprogramming.

- Almost all modern computing systems are resource-configurable. Memory and resources of the environment vary from system to system. When a system is resource-configurable, other mechanisms are provided so that software can determine the number and extent of the resources. For example, instructions of $\mathcal{L}_{HW}$ may return this information. Thus, software need not assume a fixed set of resources.

3.1.2 Efficiency

One solution to building a hypervisor that satisfies Interface Equivalence is for the hypervisor to be an interpreter, written as software, for the architectural definition. In this case, all instructions of $\mathcal{L}_{VM}$ are implemented by the hypervisor. In fact, simulators are frequently written for new architectural definitions so that software may be tested before a hardware realization of the processor is available.

Simulation is much too slow to be useful for VFT (or for any production hypervisor). The practicallity of VFT is dependent on how it impacts performance. One component of this impact is the performance degredation due to execution in a virtual machine architecture.

To preclude this solution, we also require the following property of a virtual machine:

Efficiency: A virtual machine executes instructions on average at close to the speed of the hardware.

Efficiency can be achieved if, during execution, a large proportion of the instructions of $\mathcal{L}_{VM}$ are inherited from $\mathcal{L}_{HW}$ and thus executed directly by the hardware.

3.1.3 Example: The Need for Implementing Actions

To satisfy the Interface Equivalence and Efficiency criteria, a hypervisor should inherit most actions of $\mathcal{L}_{VM}$ from $\mathcal{L}_{HW}$. Some actions of $\mathcal{L}_{VM}$ must be implemented by the hypervisor, however. An example will show why. Suppose that an architectural definition provides a hardware register INT to indicate whether an interrupt is pending. Suppose further that $\mathcal{L}_{HW}$ includes an instruction

$$\mathsf{READINT}\ r$$

which copies INT into general purpose register r.

For $\mathcal{L}_{HW}$ and $\mathcal{L}_{VM}$ to be identical, the state of a virtual machine VM_i must include INT and $\mathcal{L}_{VM}$ must include READINT r. Moreover, the semantics of READINT r when executed by VM_i should detect the presence of a virtual interrupt pending for VM_i. If READINT r were inherited from $\mathcal{L}_{HW}$, then execution by VM_i would read the hardware INT register. Thus, a program running on VM_i could detect the presence of a hardware interrupt that may not translate to a virtual interrupt destined for VM_i. This violates Interface Equivalence, and a program might behave differently when run with the hypervisor as opposed to when run on the hardware directly.

3.1.4 Sensitive Actions

If, when inherited, an action might cause Interface Equivalence to be violated, then that action is defined to be *sensitive*. Consider the example of a machine's general purpose registers. As long as whenever a particular virtual machine VM_i is executing, the hardware registers have been loaded with VM_i's values, they reflect the state of the virtual machine and operations on those registers satisfy Interface Equivalence.

In the example above, the READINT r instruction is sensitive because the INT register does not reflect the virtual state. A virtual INT register must be part of the state of each supported virtual machine. And, the READINT r must be implemented by the hypervisor to copy the virtual INT register to general register r. By separating the virtual state from the hardware state for INT and implementing the instruction, Interface Equivalence can be satisfied.

3.1.5 Some Implications

It is worth noting a few implications of the above characterization of virtual machine architectures.

First, Interface Equivalence implies that a virtual machine may not detect the existence of the hypervisor or of other virtual machines. This is because such detection would have to occur by executing some sequence of actions of $\mathcal{L}_{VM}$. The same sequence of actions of $\mathcal{L}_{HW}$ would have to behave differently, thus violating Interface Equivalence. This forces an isolation between virtual machines and the hypervisor, and it provides a clue for implementing the hypervisor.

Second, Interface Equivalence implies that the hypervisor must control physical memory management and access to all elements of the environment. This is because physical memory and elements of the environment are shared by the multiple virtual machines. Since the virtual machines are isolated, they cannot cooperate to share system resources. Thus, resource sharing must be provided by the hypervisor. Hypervisor control of all access to elements of the environment is known as the *resource control* property in the virtual machine literature [MS70].

Resource control allows assumptions A3 (page 23), A7 (page 35) and A8 (page 40) to be satisfied. Recall, A3, A7 and A8 require the hypervisor to implement input instructions, output instructions, and virtual interrupts. Resource control subsumes these requirements.

Last, to support Interface Equivalence, all sensitive actions must be implemented by the hypervisor. The following sections will discuss standard mechanisms employed by hypervisors to implement actions. If these mechanisms do not allow <u>all</u> sensitive actions to be implemented, then the underlying architecture cannot be virtualized using the standard mechanisms.

It is important to note that not all architectures are virtualizable using standard mechanisms. One of the questions this chapter answers is whether the HP–PA architecture is virtualizable using them. And, if not, what new mechanisms and/or changes in the architectural definition (such as a subset of the HP–PA) can result in a virtualizable architecture for our prototype.

3.2 Implementing a Virtual Machine

3.2.1 Dual Mode Architecture

Most modern day processor architectures provide (at least) two distinct privilege modes — user and supervisor. When a machine is executing in *supervisor mode*, any instruction may be executed. When a machine is executing in *user mode*, however, a proper subset of the instructions are no longer available for execution. The set of instructions available only in supervisor mode are called *privileged instructions*. If execution of a privileged instruction is attempted in user mode, a *privilege trap* occurs and a handler is executed. The privilege trap handler may cause a transfer into supervisor mode for execution of the handler.

Instructions for accessing the environment (i.e. output instructions) and for maintaining isolation between address spaces on an architecture are generally privileged instructions. Use of these privileged instructions allows conventional operating systems to share the resources of the environment in a protected way. Most implementations of virtual machine architectures similarly use privilege modes to share resources of the environment between virtual machines and to isolate virtual machines from one another and the hypervisor. A typical virtual machine architecture is implemented as follows.

The hypervisor executes in supervisor mode; all other software, including the operating system of the virtual machine, is executed in user mode. The privilege mode is part of the state of the virtual machine. When the software of the virtual machine would be running in supervisor mode on a conventional architecture, it is run in a virtual supervisor mode on a virtual machine architecture, while the actual privilege mode is user.

The hypervisor receives control whenever there is a privilege trap because the virtual machine attempted to execute a privileged instruction. The hypervisor then simulates[1] those privileged instructions. The privilege trap, then, is the primary mechanism by which the hypervisor implements instructions of $\mathcal{L}_{VM}$.

Any action that may read or change the privilege mode must be sensitive. This is because if such an action were inherited, execution of that action would change the actual privilege mode and the hypervisor could no longer maintain Interface Equivalence. For example, the delivery of traps and interrupts may change the current privilege mode, so these actions are sensitive and must be implemented by the hypervisor.

Figure 3.3 illustrates the typical structure of a hypervisor. A dispatcher module serves as the single entry point. The dispatcher is entered on every trap and interrupt and is the top-level control module of the hypervisor. When entered, the dispatcher

[1]We use *simulate* to refer to the transformation in virtual machine state for an action implemented by (as opposed to inherited by) the hypervisor.

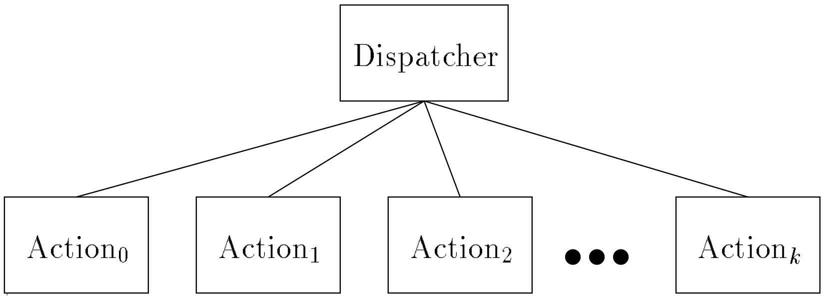


Figure 3.3: Typical Hypervisor Modules

must determine what action is simulated and which virtual machine is the destination for the effects of the action. The dispatcher then invokes the appropriate action module, which performs the required transformation on the virtual machine state. Thus, there would be an action module for each type of trap or interrupt and action modules for each privileged instruction.

3.3 HP–PA Hypervisor Design

The next two sections will discuss the hypervisor design for the HP–PA architecture, relating it to the techniques described above. The remaining sections discuss interesting problems related to building a hypervisor for the HP–PA.

3.3.1 HP–PA Privilege Levels

The HP–PA provides four privilege levels, numbered 0–3. Privilege level 0 is equivalent to supervisor mode. When executing at privilege level 0, all instructions may be executed. Privilege levels 1–3 are equivalent to user mode, since any attempt to execute a privileged instruction results in a privilege trap. Privilege levels 1–3 behave differently when coupled with the access control mechanisms of the HP–PA.

Table 3.1: Privilege Level Mapping

Virtual Privilege Level	Actual Privilege Level
0	1
1	2
2	3
3	3

HP–UX, a Unix-like operating system for the HP–PA, uses only two of the privilege levels provided by the HP–PA: 0 and 3. The portion of the operating system that must have access to the entire instruction set, the *kernel*, executes at privilege level 0. The remainder of the operating system and all applications execute at privilege level 3.

When our hypervisor is interposed, the actual privilege levels used by the virtual machine are mapped. The HP–UX kernel, run at virtual privilege level 0, is actually executed at privilege level 1. The remainder of the virtual machine, run at virtual privilege level 3, is actually executed at privilege level 3. The correspondence between virtual privilege level and actual privilege level in the presence of our hypervisor is given in Table 3.1.

3.3.2 Trap and Interrupt Control

On the HP–PA, delivery of hardware interrupts and traps is handled in a uniform mannner. Each interrupt and trap type is assigned an exception number. The delivery of a trap or interrupt causes the following sequence of actions by the hardware:

1. A (small) current context is saved in *exception context control registers.* It includes the current PC, the current PSW, and, if appropriate, a copy of the instruction causing a trap.

2. The current privilege level is set to 0.

3. A predetermined PSW of all zeroes is loaded. This PSW disables interrupts and address translation.

4. Execution begins at a physical location determined by the *interruption vector address control register* (IVA) and the exception number, E. In particular, PC := IVA + $(32 \cdot E)$.

In our hypervisor implementation, the dispatcher is the target for all traps and interrupts. This is accomplished by loading the address of the hypervisor's dispatcher into the IVA. Thus, any trap or interrupt causes the hypervisor to be executed when the trap or interrupt is delivered by the hardware. The dispatcher then determines which (if any) action module should be invoked.

The state of each virtual machine must include a virtual IVA, a virtual PSW, and virtual exception context control registers. The hypervisor uses these to simulate a virtual trap or virtual interrupt for the virtual machine. To deliver a trap or interrupt, the hypervisor follows the steps outlined above using these virtual counterparts.

The remainder of this chapter will discuss issues that are particularly interesting or problematic in building a hypervisor for the HP–PA.

3.3.3 Storage Management

Interface Equivalence implies that all features of the architectural definition are reflected in $\mathcal{L}_{VM}$. If $\mathcal{L}_{HW}$ includes storage management features to support conventional virtual memory, so must $\mathcal{L}_{VM}$. Architecural features for paging and/or segmentaion, translation look-aside buffers (TLB), and page or TLB miss traps fall into this category.

The HP–PA $\mathcal{L}_{HW}$ uses byte addressing to fetch instructions and data from memory. The byte addresses may be either absolute addresses or virtual addresses. *Absolute addresses* reference physical memory without address translation, while *virtual addresses* are translated to absolute addresses and are subject to protection and access rights checks before physical memory is accessed. Instruction fetch, LOAD, and STORE instructions reference memory.

Absolute addresses range from 0 to the extent of physical memory (max 2^{32}). This limit can be determined be executing instructions of $\mathcal{L}_{HW}$. Address translation is disabled and all address references are treated as absolute addresses when the PSW C and D bits are clear (C for Code (instruction) address translation and D for Data address translation).

The HP–PA uses 64-bit addresses. Virtual memory is structured as a set of segments, or *virtual spaces*, each containing 2^{32} bytes. During virtual address translation, a virtual space is selected by a space identifier contained in the upper 32 bits of the virtual address. The byte *space offset* within the space is specified by the lower 32 bits of the virtual address. (See Figure 3.4.)

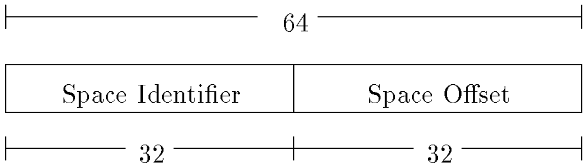


Figure 3.4: HP–PA Virtual Address as Space and Offset

Eight 32-bit *space registers*, s0 – s7, are used to maintain space identifiers and are used in virtual address calculations for both instructions and data. For instructions, a 64-bit PC is used. Some forms of the BRANCH instruction specify 3 bits identifying a space register whose contents are the desired space identifier. These forms of the BRANCH instruction load the upper 32 bits of the PC with the space identifier. Until another such branch is executed, the upper 32 bits of all updates of the PC are taken from the upper 32 bits of the current PC. Thus, when execution begins in a particular space, it remains in that space until an inter-space branch is executed.

Instructions that reference memory (LOADs and STOREs) either implicitly or explicitly specify 3 bits naming a space register. These instructions also have the normal collection of addressing modes for specifying a 32-bit space offset (immediate, register-direct, pre-decrement, post-increment, indexed, etc.). The 64-bit virtual address is constructed by concatenating the 32-bit space identifier from the specified space register with the 32-bit space offset.

For storage management purposes, the entire virtual address space is divided into pages of 4096 bytes in length. The *byte offset* into the page is specified by the least significant 12 bits of the virtual address. The upper 52 bits define the *virtual page number*. (See Figure 3.5.)

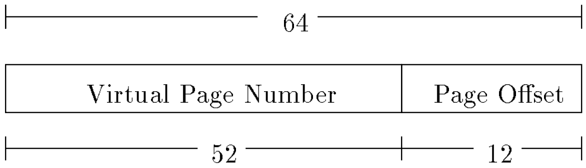


Figure 3.5: HP–PA Virtual Address as Page and Offset

Virtual addresses are translated into absolute addresses using a translation lookaside buffer (TLB). The TLB accepts a virtual page number and returns the corresponding *physical page number* if the translation is defined in the TLB. Otherwise, it generates a *TLB miss trap.* If a TLB miss trap does not occur, address translation is completed by simply concatenating the physical page number returned by the TLB with the page offset from the virtual address. This yields a 32 bit absolute address.

Since the TLB is not large enough to hold translations for all physical pages, all translations for memory resident pages are stored in a memory structure called the *page table.* On a TLB miss trap, it is the responsibility of the software to search the page table for the required translation. If the translation is not present in the page table, then the page is not resident in physical memory. Otherwise the translation is inserted into the TLB and the instruction is re-tried.

To satisfy Interface Equivalence, each virtual machine must have its own set of virtual space registers. Each virtual machine defines its own set of spaces using these virtual space registers. Memory reference instructions access the actual space registers directly during the address translation process. Thus, to provide virtual space registers the hypervisor must either (i) implement every instruction that accesses the space registers or (ii) control the hardware space registers so they reflect the virtual

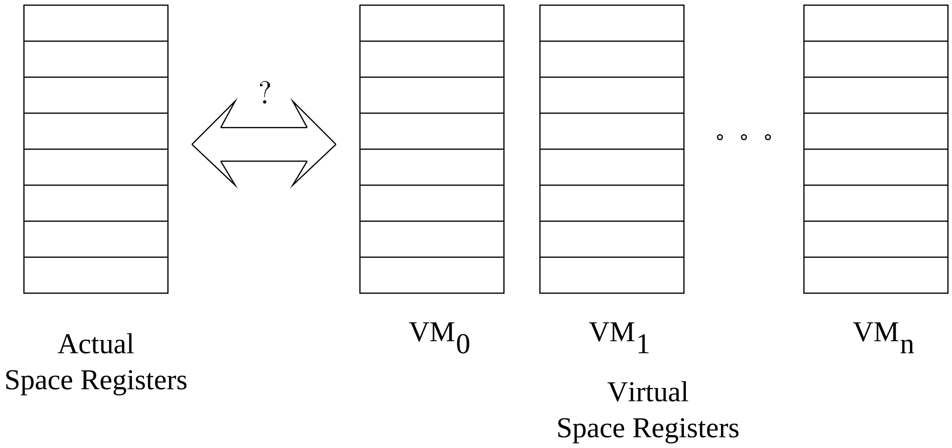


Figure 3.6: Mapping Actual Space Registers to Virtual Space Registers

machine state when a given virtual machine is executed. Since (i) violates Efficiency, (ii) was pursued.

The problem of using the actual space registers to provide virtual space registers for each virtual machine is depicted in Figure 3.6. One might attempt to solve this problem by simply multiplexing the set of hardware space registers between the virtual machines, just as is done for the general-purpose registers. Whenever VM_i becomes the active virtual machine, the hardware space registers would be loaded with the space identifiers appropriate to VM_i.

However, this solution fails to provide the required isolation between the hypervisor and the virtual machine. This is because the structure of the HP–PA memory system treats space identifier 0 as being special. Addresses in space 0 can have their virtual addresses be the same as their absolute addresses. Each virtual machine uses space 0 for the kernel, so that the virtual machine may operate with address translation enabled or disabled while executing in the kernel and have address references refer to the same physical memory locations. This use by the virtual machine conflicts with the hypervisor using absolute addresses and space 0 for its own purposes.

To solve this problem, a hypervisor might define a mapping between the virtual spaces defined by a virtual machine and a set of actual spaces defined by the hypervisor. Whenever a new space is defined by a virtual machine (by loading a space register), the hypervisor would load a new actual space into the given space register and maintain a mapping for each virtual machine between its virtual spaces and the actual spaces. This would allow the hypervisor to create a virtual space 0 for each virtual machine that did not conflict with its own address space.

With this solution, any action that reads or updates the space registers is sensitive and must be implemented by the hypervisor. The HP–PA instruction MFSP i, j (Move From SPace) copies space register i to general register j. The HP–PA instruction MTSP j, i (Move To SPace) updates space register i with the contents of general register j. Unfortunately, MFSP is not privileged and MTSP is not privileged if $i \in \{0, 1, 2, 3, 4\}$. Thus, these instructions cannot be implemented by the hypervisor. This means that the HP–PA may not be virtualized using the standard mechanisms.

Our solution to the space register problem was to support only a single virtual machine and allow the hypervisor and the kernel of the virtual machine to share the same address space. (See Figure 3.7.) Now, the hypervisor and the virtual machine do not conflict over use of absolute addresses and space 0 because they are in the same address space. This is perceived by the kernel as simply a larger kernel image, and the hypervisor looks to the kernel like another operating system driver. The

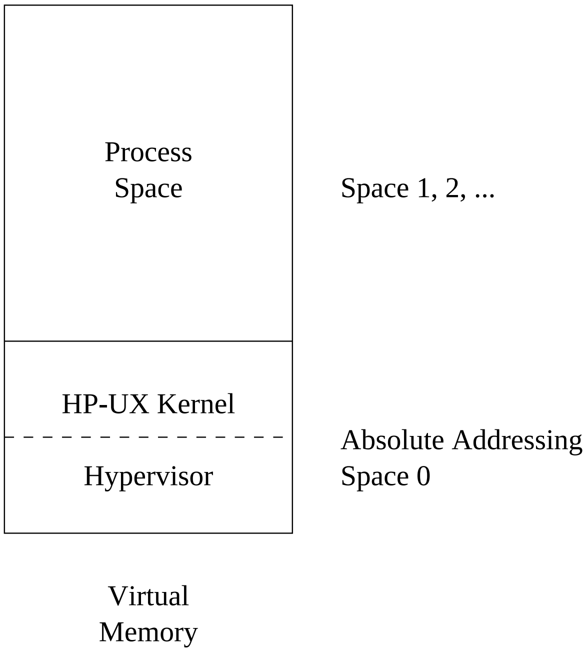


Figure 3.7: Hypervisor as Part of the Kernel

virtual machine's operating system can perform all storage management and let this storage management be reflected by the hardware features designed for that purpose.

There is still a logical separation between the hypervisor and the virtual machine kernel. The hypervisor still executes at privilege level 0 and the kernel at actual privilege level 1. All pages of the kernel (now both the hypervisor and the virtual machine kernel) are perminently resident in memory, so no portion of the hypervisor can be swapped out from main memory.

3.3.4 Access Control

The HP–PA provides a set of mechanisms that are used to provide a secure and protected execution environment for programs. These mechanisms are part of the address translation mechanism. Associated with each memory page is a set of *access rights*. These access rights determine what type of access (execute, read, or write) may be performed on the page. If the access is attempted but not permitted, an *access trap* results.

The access check for a page is, in part, based on the current privilege level. Suppose that the access rights for each page specify two privilege levels PL^T_{min} and PL^T_{max} for each access type T. [2]. PL^R_{min} and PL^R_{max} are used to specify the range of privilege levels that are permitted to perform a read access of the page, PL^W_{min} and PL^W_{max} for write access, and PL^X_{min} and PL^X_{max} for execute access. For example, if a page p had read access rights $PL^R_{min} = 0$, $PL^R_{max} = 1$, then an attempt to read page p with a privilege level of 0 or 1 would pass the privilege check. If the current privilege level were 2 or 3, then an attempt to read page p would result in an access trap.

Whenever address translation is enabled, access rights are checked. Thus, the access check is performed for almost every instruction execution, memory load, and memory store. To satisfy Efficiency, the majority of these instructions (and thus their access checks) must be inherited. To satisfy Interface Equivalence, the check must generate an access trap if and only if the access trap would occur when executing on the hardware directly. Thus, the access-rights check must be based on the mapped (actual) privilege levels, while ensuring that the virtual machine can only learn about the virtual privilege levels.

[2]This is a simplification of the actual privilege access rights structures used by the HP–PA. See [Hew90] for the actual access rights structures.

The access rights are checked during address translation. The access rights discussed above are stored in the TLB. Although the virtual machine manages memory, the instructions to insert entries into the TLB are privileged. Thus, the hypervisor can simulate TLB insert instructions and transform the access rights specified by a virtual machine TLB insert instruction. The access rights are transformed to reflect the actual privilege levels so that Interface Equivalence can be maintained.

As an example, suppose the virtual machine executes an insert TLB instruction for a kernel data page. The access rights specified by the instruction should include read and write access for the kernel, running at virtual privilege level 0 giving $PL^R_{min} = 0$, $PL^R_{max} = 0$ and $PL^W_{min} = 0$, $PL^W_{max} = 0$. The hypervisor simulates the TLB insert by translated it into an insert TLB instruction with $PL^R_{min} = 0$, $PL^R_{max} = 1$ and $PL^W_{min} = 0$, $PL^W_{max} = 1$. Virtual machine software executing at a virtual privilege level higher than 0 (actual privilege level higher than 1) will still generate an access trap when it attempts to perform a read or write to the kernel page. The virtual machine kernel, executing a virtual privilege level 0 (actual privilege level 1) can still access its own page. The mapping of PL^R_{min} is to actual privilege level 0, so the hypervisor may access the page whenever the kernel could.

The mapping given by Table 3.1 is applied to PL^R_{max}, PL^W_{max}, and PL^X_{max} for all insert TLB instructions. Note that virtual privilege levels 2 and 3 are both mapped to actual privilege level 3. The semantics of $\mathcal{L}_{VM}$ should cause an access trap if code running at virtual privilege level 3 attempted to write access a page whose PL^W_{max} specified virtual privilege level 2. In our scheme it would not. HP–UX uses only privilege levels 0 and 3, so this is not a problem for the prototype.

In summary, the access control mechanisms of HP–PA make every access to memory sensitive because each access depends on the actual privilege level. A solution sufficient for our prototype is possible by exploting knowledge of how the HP–UX operating system uses the HP–PA.

3.3.5 PROBE instructions

The HP–PA provides a PROBE instruction to determine if a read or write access to a given address would generate an access trap. The instruction specifies the type of access (R or W), an address a, and a privilege level PL. The instruction returns 1 in a destination register r if the access is permitted and returns 0 otherwise.

Clearly, this instruction is sensitive. A PROBE specifying $PL = 1$, R for a kernel page with virtual access $PL^R_{min} = 0$ and $PL^R_{max} = 0$ would return 0 when running without the hypervisor, but would return 1 when running with the hypervisor. Unfortuanately, the PROBE instruction is not privileged. Thus, the HP–PA may not be virtualized using the standard mechanisms.

Again our hypervisor implementation exploits the knowledge that HP–UX only uses privilege levels 0 and 3. Any PROBE instructions will only specify a PL of 0 or 3. Because we do not modify PL^R_{min} or PL^W_{min}, these return the same value as before. The change to PL^R_{max} is not a problem because we also exploit the relationship of $PL^R_{min} \leq PL^R_{max}$. (Likewise for PL^W_{max}.)

3.3.6 Branch and Link Instructions

In HP–PA all instructions are one (32-bit) word in length. All instructions in memory are word aligned. As noted above, all addresses are byte addresses. Thus, the two least significant bits of any instruction address are always zero. The PC stores the current instruction address and the HP–PA uses the two least significant bits of the PC to maintain the processor's current privilege level. These two bits are ignored when memory is accessed to fetch the next instruction.

With the current privilege level as part of the PC, any instruction that leaves the PC (and thus the privilege level) in the virtual machine state is sensitive. Consider the BL (Branch and Link) instruction of the HP–PA. The sequence:

```
        BL      ⟨target⟩, r
α:      ⟨next instruction⟩
```

causes execution to begin at $\langle target \rangle$ and stores PC + 4, the return address, in r. When this sequence is executed by a virtual machine running at virtual privilege level 0, the semantics of the instruction should be for $r = \alpha \mid 0$[3] complete with the two least significant bits equal to zero. However, when this instruction is executed with the hypervisor present, the result is $r = \alpha \mid 1$. (i.e. the least significant bit is set). The BL instruction is not a privileged instruction, so again, the HP–PA architecture may not be virtualized using standard mechanisms.

To address this problem, we analyzed the the use of the return address under HP–UX to see if the least significant 2 bits are ever actually used. We found that the return address is generally used by subroutines to branch back to the caller once a subroutine is complete. This use ignores the two least significant bits in r and constructs a new PC using the current privilege level.

Also note that all code executing at virtual privilege level 3 is executed at actual privilege level 3 when the hypervisor is present, so this code is uneffected. In addition,

[3] $a \mid b$ denotes the bitwise OR of a and b

any code written in a high level language simply stores the return address on the stack and branches using that value, so the current privilege level is never visible to the high level language code. Only the assembly language code of the kernel might be affected by these low order bits.

When executed at privilege level 0, the above code fragment gives a load-position independent way of determining the physical address of α. The HP–UX kernel uses this technique in only a very few places. With no other alternatives, the solution forced upon us was to modify the kernel to mask out the two least significant bits in those cases where this occurred.

3.3.7 Changing the Privilege Level

Any mechanism provided by $\mathcal{L}_{HW}$ to change the current privilege level is sensitive. Traps and interrupts provide one such mechanism, and they are handled as discussed above. An increase in privilege must also occur whenever an application process requires a service of the operating system (through a system call) that necessitates privileged instruction execution.

The HP–PA mechanism for increasing privilege level is the GATE instruction. The new privilege level is specified in a field of the TLB entry for the page that includes the GATE instruction. Thus, when HP–UX must change from privilege level 3 to privilege level 0 for a system call, it branches to a special page called a *gateway* page that has a 0 in the appropriate field of its TLB entry. The gate instruction also specifies a branch target and execution begins at that target at the new privilege level.

The hypervisor must ensure that a GATE instruction changing the current privilege level to 0 instead changes the virtual privilege level to 0 and the actual privilege level to 1. Since the GATE instruction is not privileged, this is accomplished by the hypervisor changing the PL_{gate} field when the TLB entry is inserted for the gateway page.

3.3.8 Memory Mapped I/O

The HP–PA uses memory-mapped I/O. Thus, control registers for I/O controllers are mapped to special memory locations. LOAD and STORE instructions are used to issue commands to the I/O controller and to initiate I/O.

To satisfy Interface Equivalence, such LOAD and STORE instructions must be implemented by the hypervisor. LOAD and STORE instructions are themselves not privileged, so another mechanism must be found to allow the hypervisor to implement these instructions.

The necessary mechanism is the access trap. Inserting entries in the TLB for the I/O memory pages is done by privileged instructions. This, then, is simulated by the hypervisor. The kernel of the virtual machine specifies access rights of $PL^R_{min} = 0$, $PL^R_{max} = 0$, $PL^W_{min} = 0$, and $PL^W_{max} = 0$ for the I/O pages. When these are inserted into the TLB, the hypervisor leaves PL^R_{max} and PL^W_{max} specified by the virtual machine kernel at 0. Then, when the kernel (executing at an actual PL of 1) attempts to manipulate an I/O controller to perform some I/O operation, an access trap occurs, causing the hypervisor to be entered. The hypervisor may then simulate the LOAD or STORE as appropriate to maintain Interface Equivalence.

Note that, if only a single virtual machine is supported and if the hypervisor has no requirement for accessing the environment itself, then the computing system resources corresponding to the environment (i.e. the I/O controllers) need not be shared. Interface Equivalence can then be satisfied by mapping the I/O controllers directly.

However, to satisfy assumptions A3 (page 23), A6 (page 27) and A7 (page 35) from Chapter 2, and to allow the hypervisor at the primary and the hypervisor at the backup to communicate with one another, the virtual machine cannot control environmental resources and still satisfy Interface Equivalence. Therefore, the technique of using access traps to implement memory mapped I/O LOAD and STORE instructions is employed by our hypervisor.

3.4 Virtualization Summary

In the preceeding sections, we have seen a number of instances where the HP–PA architecture is not virtualizable. We have surmounted these problems by:

1. Restricting the hypervisor to support a single virtual machine.

2. Using knowledge of how the HP–UX operating system uses the HP–PA, in particular the use of just two of the privilege levels.

3. Modifying the HP–UX kernel in the few cases of the branch and link problem.

This hypervisor is sufficient for building a prototype of the VFT system and investigating its performance.

Chapter 4

Prototyping the VFT Protocol

The VFT protocol presented in Chapter 2 was built using abstractions. The properties of the I/O Model and assumptions A1 – A8 of Chapter 2 are among these. Implementing the protocol and these abstractions using the HP Precision Architecture is the subject of this chapter.

We first describe the hardware on top of which our prototype was built. Given this, we had to solve a number of problems. How we addressed each of these problems is discussed in the remaining sections.

4.1 The Target Architecture

Figure 4.1 depicts the target architecture for a prototype fault-tolerant computing system. In addition to a virtual HP 9000/700 processor, the architecture includes a console connected through a serial line and a disk connected using a SCSI bus.

The console provides the operating system, HP–UX, with a means for communicating with an operator. The SCSI disk provides storage for HP–UX to use for

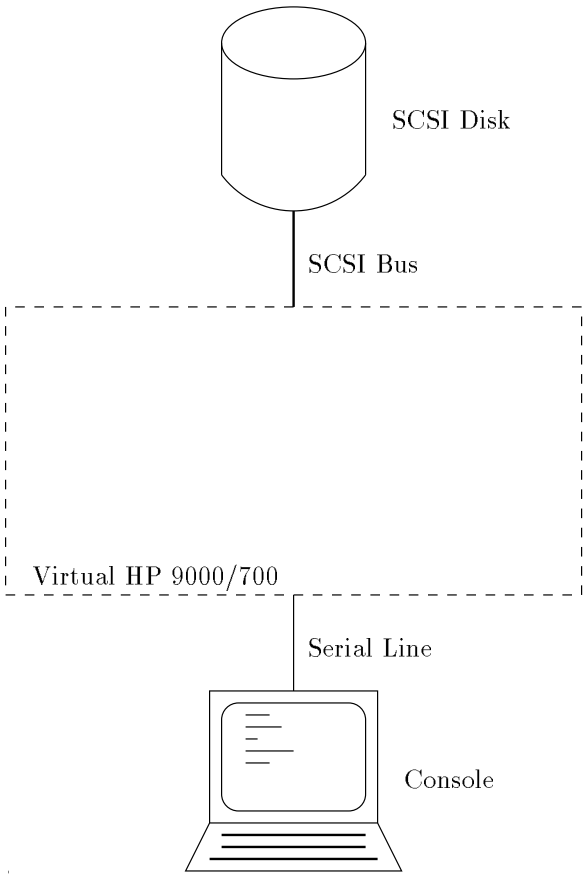


Figure 4.1: Target architecture for prototype

the root file system and for swap space. Although HP–UX can use more complex console devices, such as a graphics display, and can support diskless configurations over a network, our more-modest configuration was sufficient for evaluating the VFT protocol.

The target architecture is realized through the prototype of Figure 4.2. The disk is attached to both the primary and backup processors through a common SCSI bus. This allows the disk to satisfy property IO3 (page 19), wherein elements of the environment must be accessible to both the primary and the backup. A discussion of how the SCSI disk satisfies other elements of the I/O Model properties appears in Section 4.3.

The primary and backup processors are connected to an Ethernet. This allows primary to backup communication as required by Comm (page 19). The Ethernet also provides a means for implementing the serial console of the target architecture. When communication to the console is requested by the virtual machine, the hypervisor converts this communication into input and output to a *remote virtual console* residing on another processor attached to the Ethernet. This allows the remote virtual console to satisfy IO3 (where a single serial line would not). This structure also allows us to support primary and backup *hypervisor consoles* to use in querying hypervisor-specific state and controlling hypervisor operation.

4.2 Input Instructions

Input instructions cause interaction with the environment, and the semantics of $\mathcal{L}_{HW}$ with repect to those input instructions must be preserved across a failure. Two types

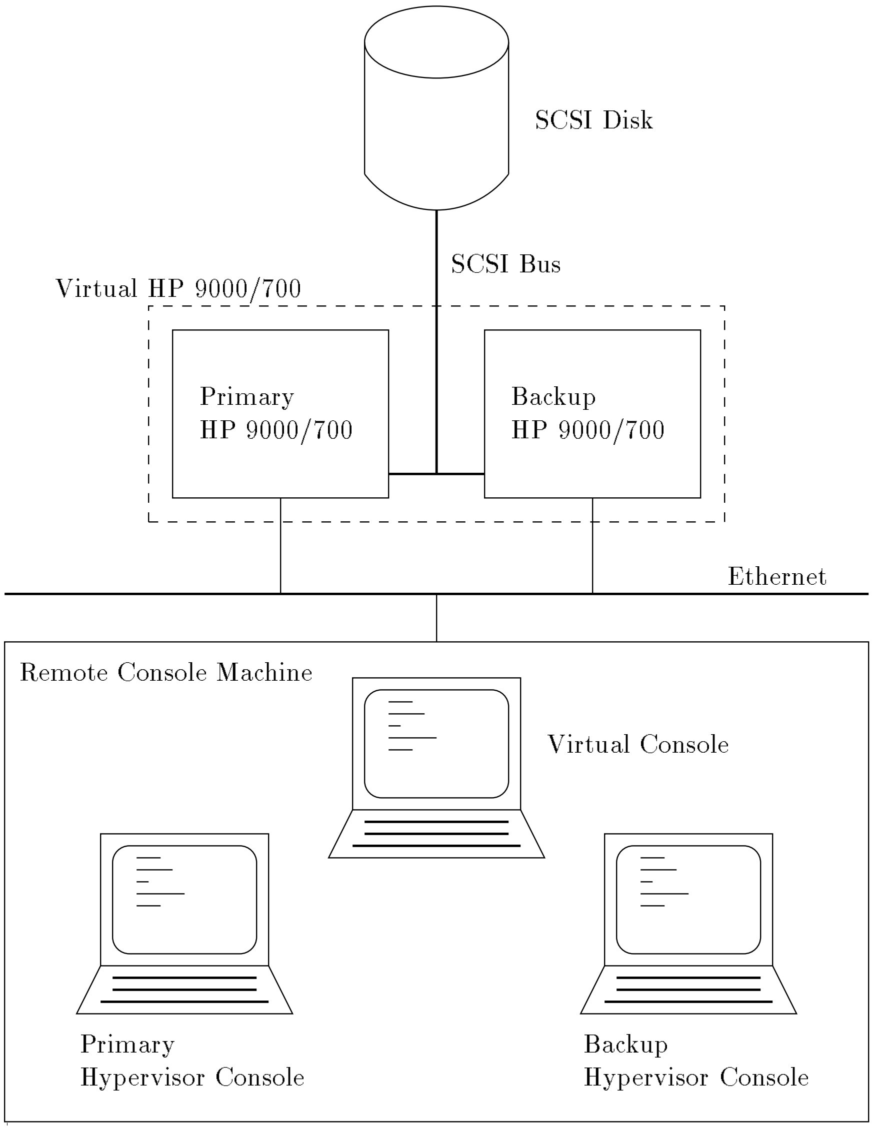


Figure 4.2: Actual Prototype Configuration

of input instruction of the HP–PA require special care in our prototype:

- Interval Timer Reads.
- I/O Controller Register Reads. These are LOAD instructions to the Memory Mapped I/O pages associated with the I/O controllers of the environment.

The interval timer will be discussed in detail next. The issue of reads of I/O controller registers are specific to the particular controller and will be discussed in Section 4.3.

4.2.1 The Interval Timer and Virtual Time

The interval timer of the HP–PA consists of two internal registers. One of the internal registers is continually counting up by 1 at a fixed rate (relative to real time). Reading the interval timer by executing a move from interval timer instruction, MFITMR r,[1] moves the value of this internal register to designated general register r. The other internal register contains a comparison value, which is set by executing a move to interval timer instruction, MTITMR r.[2] When the counter register and the comparison register contain identical values, an interrupt is generated.

The MTITMR instruction is a privileged instruction. The MFITMR instruction is only privileged if the PSW S bit[3] is set.

HP–UX uses the interval timer to implement a clock approximating real time. The interval timer is set to interrupt every 1/100 of a second. This clock serves as the basis for all time and clock abstractions in the HP–UX operating system. The interval timer interrupt used by HP–UX is called the *clock interrupt*.

[1] This is actually a varaint of the MFCTL (Move From ConTroL register) instruction.

[2] This is a variant of the MTCTL (Move To ConTroL register) instruction.

[3] The S stands for Secure interval timer.

Since reads of the interval timer are input instructions, the hypervisor must get control on every MFITMR instruction. Since the clock interrupt constitutes a souce of non-determinism in the primary, this interrupt must be delivered at the same point in the computation at the primary and backup, just as was done for completion interrupts. Also, since the backup gets its view of time from the primary, the backup must provide a consistent view of time after a failure.

The hypervisor can set the S bit in the PSW to make MFITMR instructions privileged, satisfying A3 (page 23) for interval timer reads. The interval timer counter register and the PSW S bit then become part of the virtual machine state that is not (necessarily) reflected by their hardware counterparts. This introduces the concept of *virtual time*, which is the time reflected through interaction with the interval timer (both reads and writes). The rate of virtual time relative to real time is discussed at the end of this section.

At the primary, reads of the interval timer reflect virtual time. A write to the virtual timer sets a virtual comparison register whose semantics are simulated by the hypervisor — when virtual time equals (or exceeds) the virtual comparison value, the hypervisor records the appropriate interrupt as pending. The interrupt is delivered at the next epoch boundary (prior to any outstanding completion interrupts). The primary also sends the virtual time ending the epoch to the backup. This epoch end virtual time is appended to the "EpochNumberMsg" message.

At the backup, reads of the interval timer return the same value returned by the primary by using the InputValues queue. A write to the interval timer is recorded in a variable associated with the virtual interval timer comparison value. At the end of each epoch, the backup compares the virtual interval timer comparison value with the epoch end virtual time received from the primary. If the value is exceeded, the backup simulates the clock interrupt at the same point as the primary.

If a failure occurs, the backup knows the virtual time for the end of the last epoch the primary successfully completed. If there have been any subsequent reads of the interval timer received from the primary, then the backup has an even more accurate view of virtual time. At the takeover point, the backup sets its own virtual time to the last virtual time received from the primary plus a delta determined by how much virtual time has elapsed since the last epoch boundary or interval timer read. In this way, the virtual time at the backup is consistent with the virtual time of the primary across a failure.

The rate of virtual time is controlled by the hypervisor at the primary. We have implemented two virtual time schemes in our hypervisor. In the first, virtual time runs at the same rate as real time. In this case, the actual interval timer can be used to provide values for interval timer reads and to generate interrupts to be later delivered to the virtual machine.

The second scheme allows virtual time to progress only while the virtual machine is running. The actual interval timer is read whenever the virtual machine is started (in ResumeVirtMach). Whenever the virtual machine is suspended and the hypervisor is entered through the dispatcher module, the actual interval timer is read again. The difference between these two time values is added to the current virtual time. Clock interrupts are simulated as described above. The advantages of the second scheme for performance modeling are discussed in Chapter 5.

4.3 The Environment: A SCSI Disk

The SCSI I/O controller (henceforth referred to as simply "controller") operates by executing *I/O programs* [NCR92]. These I/O programs reside in the processor's main memory and are provided by the HP–UX SCSI driver (henceforth referred to as 'driver'). The I/O program instructions are executed by the controller and allow manipulation of the SCSI bus lines enabling the I/O program to follow the SCSI bus protocol [ANS89], to transmit and receive data between the processor's main memory and the SCSI bus, and to interrupt the processor.

Given the I/O programs provided by the driver, a state diagram for the controller can be constructed. For simplicity of exposition, the state diagram shown in Figure 4.3 does not include initialization and error states[4]. A transition from one state to the next occurs due to:

- The driver commanding the controller to initiate an I/O program. Initiating an I/O program is successful only if the controller is not currently executing an I/O program. An I/O program is initiated by writing the address of the I/O program to a memory mapped controller register.

- A condition generated by the SCSI protocol. The driver detects such a transition through the I/O program, which executes an instruction that signals an interrupt. The type of interrupt is indicative of the originating condition and can be obtained by reading a memory mapped controller register. An I/O program interrupt instruction terminates the currently executing I/O program.

[4] These states are present in the actual SCSI operations and are handled by our hypervisor.

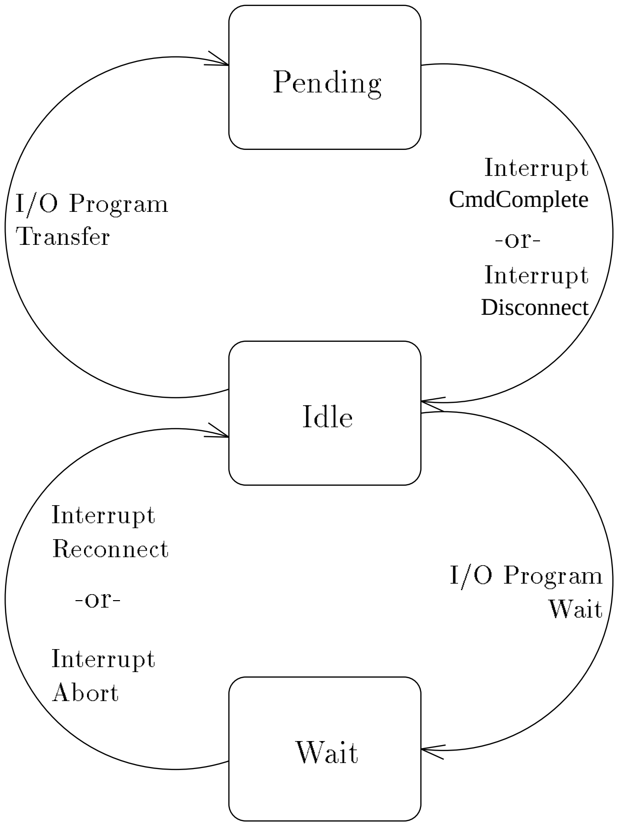


Figure 4.3: SCSI Controller State Diagram

The controller begins in the Idle state. Whenever the driver detects a transition into the Idle state, the driver checks its queue of submitted SCSI requests. A SCSI request is *submitted* when it is given to the driver by higher level software of the virtual machine. If there are submitted SCSI requests, the driver intializes the transfer parameters and initiates the Transfer I/O program. The write of the Transfer I/O program address to the controller register defines the output instruction for a request as required by IO1 (page 18). The controller transitions to the Pending state.

While in the Pending state, the Transfer I/O program is being executed by the controller. This I/O program establishes a connection with the SCSI disk. Once the connection has been established, the SCSI bus is busy until either (i) the request is complete, or (ii) the SCSI disk requests a disconnect. Both of these conditions cause a transition back to the Idle state and an interrupt is signalled to the processor.

The disconnect facility is provided by the SCSI protocol to allow concurrent operations between other host-controller/SCSI-device pairs. Say, for example, a disk read request would require significant seek time before the transfer could commence. The disk would disconnect for the seek time and then reestablish the connection for the data transfer.

When a request is complete, the controller signals completion of the request with a CmdComplete interrupt and transitions back to the Idle state. Whenever the controller issues the CmdComplete interrupt, it also makes the completion status of the request available in a memory-mapped controller register. The SCSI protocol provides a rich collection of error returns. The semantics of the SCSI error return CHECK CONDITION are roughly equivalent to our "transient I/O error". The CmdComplete interrupt and the CHECK CONDITION status return satisfy the requirements of IO2 (page 18). When the driver receives a CHECK CONDITION return on a SCSI request, it typically retries the request.

For a connection to be reestablished, the controller must be in the Wait state. Thus, whenever the controller enters the Idle state and the driver cannot initiate an outstanding request[5], the driver causes a transition to the Wait state by initiating the Wait I/O program.

When a SCSI device is ready to reestablish a connection, it makes a reconnect request to the controller over the SCSI bus. This causes the Wait I/O program to signal a Reconnect interrupt and also specifies the SCSI device. The driver may then resume execution of the appropriate Transfer I/O program.

If the driver receives a request that it can initiate and the controller is in the Wait state, it must force the controller into the Idle state before submitting the request. This is accomplished by writing to a controller register. The controller then exits the Wait state and signals an Abort interrupt. Once the Abort interrupt is delivered, the driver may initiate the new request.

IO3 (page 19) is satisfied by putting both the primary and backup on the same SCSI bus as the SCSI disk. IO5 (page 49) is also satisfied by the SCSI disk.

4.3.1 Failover Considerations

Delivering transient I/O errors for outstanding requests

The backup must deliver transient I/O error completions for all outstanding requests. As mentioned in the previous section, this may only occur on the CmdComplete

[5] The driver may not be able to initiate a request either because there are no outstanding requests or because the only outstanding requests are for SCSI devices that are already busy (and disconnected).

interrupt. If the state of the controller, as determined by the virtual machine at the backup, is Pending then this interrupt and its associated CHECK CONDITION return can be simulated by the hypervisor at the end of the epoch of failure. Note that this simulation by the hypervisor at the backup only involves transformations in the virtual machine state. The actual controller remains in it's initial Idle state.

If, however, the driver at the backup believes that the controller is in the Idle or Wait state at the takeover point, the hypervisor must simulate controller transitions back to the Pending state before it can simulate the CmdComplete for the outstanding request.

Suppose, for example, the driver at the backup believes that the controller is in the Idle state. The hypervisor must allow the driver to initiate the Wait I/O program, simulate a Reconnect interrupt for the SCSI device with the outstanding request, allow the driver to initiate the Transfer I/O program, and finally simulate the CmdComplete interrupt.

The problem becomes even more complex if multiple SCSI devices are supported on one SCSI I/O controller. A CmdComplete interrupt for only one device may be delivered at a time. Thus, the hypervisor must simulate SCSI controller transitions until all outstanding requests have received the CmdComplete interrupt. This, in effect, serializes the delivery of 'transient I/O error' completions for the outstanding requests. Only then can the hypervisor at the backup begin interacting with its SCSI controller.

Ensuring consistent state of the device and the controller

When the backup takes over as primary, the state of the disk and the SCSI controller must be consistent with the interaction seen by the backup virtual machine. The state of the disk is relative to the progress made by the primary virtual machine. The state of the controller at the backup is in its initial state, since all interaction with the controller at the backup has been suppressed.

For the state of the disk, we must ensure that there is no unresolved ongoing interaction between the controller at the primary and the disk. At the time of failure, the controller at the primary may have had a transfer in progress and been in the Pending state. Another possibility is that at the time of failure, the controller at the primary was in the Wait state after a disconnect. In the current hypervisor implementation, the backup takes over the SCSI address that the primary had when the backup takes over interaction with the SCSI disk. Upon takeover, the backup may find the disk attempting to reestablish the proper bus protocol phase or attempting to reestablish a connection. To avoid such problems, the hypervisor at the backup performs a SCSI Bus Reset, which is guaranteed to abort all outstanding requests for all devices on the SCSI bus.

For the state of the controller versus the state of the virtual machine, we must force the state of the controller to be consistent with the interaction as seen by the virtual machine at the backup. When the controller is in the Idle state, certain operating parameters and operating modes may be set or modified by reading and writing combinations of controller registers. These operating parameters have been passed through to the controller at the primary, but the controller at the backup is still in its initial state. At the takeover point, the controller at the backup must be updated with these operating parameters.

4.4 The Epoch Trap: HP–PA Recovery Counter

The VFT protocol, by assumption A6 (page 27) requires a mechanism to implement epoch traps. HP–PA provides an instruction countdown register, called a *recovery counter*, which can be used to satisfy A6.

The recovery counter is decremented by 1 during execution of each instruction, provided the R bit in the PSW is set. When the most significant bit of the recovery counter is set, a recovery counter trap occurs. The recovery counter decrement operation and conditional trap may be disabled by clearing the PSW R bit.

The VFT protocol loads the epoch length minus one into the recovery counter when preparing to begin a new epoch (in ScheduleNextEpochTrap()). Whenever execution of the virtual machine is resumed, the PSW R bit is set so that only virtual machine instructions are counted. When a trap or interrupt occurs, the fixed PSW loaded by the hardware has the R bit clear, so instructions associated with hypervisor execution are not counted. When the hypervisor simulates an instruction of $\mathcal{L}_{VM}$, the recovery counter is decremented.

Although the recovery counter is provided by the HP–PA, this feature is not used by HP–UX. To simplify the hypervisor, the recovery counter is not provided in $\mathcal{L}_{VM}$. Note, however, that reads and writes of the recovery and the PSW are sensitive and are privileged, so a virtual recovery counter could be implemented if required.

4.5 Communications

Our realization of the VFT protocol must provide reliable FIFO communication between the primary and backup hypervisors. As mentioned at the beginning of this chapter, the Ethernet is used as the communication medium. A standard sliding window protocol with sequence numbers and positive acknowledgments has been implemented to provide reliable FIFO messages above the Ethernet [Tan89].

Because the facilities of the operating system are not available to the hypervisor, the hypervisor must provide its own Ethernet driver to interface to the specific Ethernet I/O controller [Int92] used in the prototype workstations.

One complication introduced with primary to backup communication is the potential for the hypervisor to be reentered. Under normal operation, the hypervisor cannot be preempted. This is achieved by disabling hardware interrupts while the hypervisor is executing. There are, however, instances where the hypervisor must enable interrupts for communications purposes.

Consider epoch-boundary processing at the primary. In Figure 2.4 (page 30), the hypervisor calls WaitStable to wait until the backup has received all communication up to the "EpochNumberMsg" message. This requires enabling interrupts so that driver command completion and incoming packet reception interrupts (for acknowledgements) can be received and processed to allow exit from WaitStable. Any interrupts received will cause the hypervisor to be reentered. In addition to communications interrupts, this may include interrupts associated with devices in the environment. These device interrupts trigger additional primary to backup communication.

4.6 Flow Control

A practical realization of the VFT protocol cannot allow the primary to get too far ahead of the backup. For each epoch that the primary is ahead of the backup, the backup consumes storage because it must keep all "InterruptMsg", "InputValuesMsg", and "EpochNumberMsg" messages for all such epochs. These messages can be quite large, since they contain the entire data transformations for the input exceptions and input instructions they refer to. Thus, some form of flow control is required.

For the prototype, a simple flow control protocol was employed. The protocol is a producer/consumer double buffering protocol. In this instance, the primary is the producer and the backup is the consumer. Each "buffer" corresponds to an *epoch window*, a fixed number x of epochs. The backup statically allocates enough storage resources for $2x$ epochs. The backup informs the primary when an epoch window is available by sending a message. The backup sends this message only after completely finishing the last epoch in the given epoch window. The primary may not enter an epoch nor send any messages associated with an epoch if the backup has not indicated that the epoch window for that epoch is available.

This protocol guarantees that the primary never sends messages that the backup cannot handle due to lack of resources. All flow control decisions are made during epoch-boundary processing, simplifying the work in the hypervisor. The protocol has a fixed communications overhead cost per epoch, regardless of the workload.

4.7 The TLB Problem

Thus far, all the problems in building a hypervisor and mapping the VFT protocol to a realization on the HP–PA could be foreseen through a careful examination of the architectural definition [Hew90]. Once the prototype was built, we encountered a problem that could not be foreseen. The problem illustrates the impact of hidden virtual machine state in our assumptions.

As described in Section 3.3.3, all address translation occurs through the TLB. Given an instruction requiring a translation, if the requested translation is not present in the TLB, a TLB miss trap occurs. There are two subcases:

1. The page referred to is not in main memory; it must be retrieved from secondary storage.

2. The page is in main memory and its translation resides in the page table.

In the first case, the page is retrieved from secondary storage, the page table and TLB updated (through a TLB insert instruction), and then the trapped instruction may be reexecuted. In the second case, the TLB insert instruction is executed and the trapped instruction is reexecuted immediately.

The set of translations in the TLB is part of the state of the virtual machine. This state is partially hidden — it cannot be directly read by instructions of $\mathcal{L}_{VM}$. However, the contents of the TLB can be inferred by the sequence of TLB miss traps that occur.

Let *tlb* be the set of translations in the TLB. An

$$\textsf{insert}\ vaddr, paddr$$

Table 4.1: Initial TLB Contents

vaddr	*paddr*
a	$\hat{a}$
b	$\hat{b}$
c	$\hat{c}$
d	$\hat{d}$

operation adds the physical address translation *paddr* for virtual address *vaddr* to *tlb*. Generally, the insert operation results in some existing translation being removed from *tlb*. The particular translation that is removed is determined by a *TLB replacement policy*. A

translate *vaddr*

operation returns the translation *paddr* if the translation exists in *tlb* and results in a TLB miss trap otherwise. The translate operations are performed directly by the hardware during instruction execution.

By assumption A2, all non-input instructions must be deterministic in the state of the virtual machine. However, in the HP–PA, the insert operation is not deterministic. The HP–PA guarantees only that the inserted translation will be present in the TLB. It does not specify which translation is removed. If the replacement policy is not deterministic, then the insert operation is also not deterministic. Thus, the state of the TLB at the primary may differ from the state of the TLB at the backup after the same sequence of TLB reference and insert operations.

Consider a 4 line TLB at the primary and the backup starting in identical state given by Table 4.1. Suppose the virtual machine references virtual pages a, e, c in that order. That reference string results in TLB operations translate a, translate e,

Table 4.2: Divergent TLB Contents

Primary TLB		Backup TLB	
vaddr	*paddr*	*vaddr*	*paddr*
a	$\hat{a}$	a	$\hat{a}$
b	$\hat{b}$	b	$\hat{b}$
e	$\hat{e}$	c	$\hat{c}$
d	$\hat{d}$	e	$\hat{e}$

translate c. The first translation is successful at both the primary and backup and the contents of the lines of the TLB at both the primary and the backup remains the same. The **translate** e operation will cause a TLB miss trap at both the primary and backup virtual machines and will result in a subsequent **insert** e,$\hat{e}$ by both virtual machines. With a non-deterministic replacement policy, the hardware at the primary may remove the c translation and the hardware at the backup may remove the d translation. This results in a divergence of the TLB state at the primary and backup, violating A2 (page 22). The resulting TLB state is given in Table 4.2.

The problem manifests itself in the computation at a later point, when the **translate** c operation is eventually performed. A TLB miss trap will be delivered to the virtual machine at the primary, but the **translate** will be successful at the backup.

Our solution to this problem requires changing $\mathcal{L}_{VM}$ to be different from $\mathcal{L}_{HW}$, but changing it in a way that is consistent with current use by HP–UX. We exploit two properties of the HP–PA and HP–UX:

1. The TLB miss trap serves two functions:

(a) *page miss trap*: A TLB miss trap occurs when the referenced page is not in main memory (and thus is also not in the TLB).

(b) *translation miss trap*: A TLB miss trap occurs when the referenced page is in main memory, but its translation is not currently in the TLB.

These two functions can be distinquished by searching the page table.

2. When the TLB miss trap is functioning as a translation miss trap, the page table search and subsequent insert operation are transparent to the rest of the HP–UX kernel.

Our solution eliminates translation miss traps from $\mathcal{L}_{VM}$. It appears to the virtual machine as if the hardware is responsible for loading TLB entries directly from the page table for pages that are resident in main memory. The only TLB miss traps delivered to the virtual machine are then page miss traps. This solution does not preclude the virtual machine kernel from performing TLB insert operations directly. The virtual machine pre-inserts TLB entries following a page fetch operation.

The hypervisor is responsible for the page table search and TLB insert operations for translation miss traps. These traps are not delivered to the virtual machine. If, after a search of the page table, the hypervisor determines that the referenced page is not in main memory, the hypervisor simulates a TLB miss trap for the virtual machine.

The disadvantage of this solution is that it requires the hypervisor to know the format of the page table used by HP–UX and where the page table is located in memory.

This solution serves to take the state of the TLB out of the virtual machine state of the altered $\mathcal{L}_{VM}$. Thus, the TLB insert operations become deterministic in the state of the virtual machine and A2 is satisfied.

Chapter 5

Performance Analysis of the VFT Protocol

Measuring the performance of our prototype VFT system and then building mathematical models of the VFT protocols gives insight into the practicality of virtual fault-tolerance. Measurements were obtained for the various components of hypervisor overhead using our prototype. These measurements gave us an indication of the current performance characteristics and, based on knowledge of our VFT protocol implementation, we formulated models to predict effects of system parameters on performance. The models were validated using additional runs of the prototype. Finally, by formulating other models, performance could be predicted for architectural variants that we conjectured could lead to performance improvement.

Epoch length was the system parameter of paramount concern. We were interested in how different length epochs affected system performance. With short epochs, interrupts are not delayed significantly by hypervisor buffering but the number of epochs that pass during a given task — and associated overhead — is increased. With long epochs, fewer epochs pass but the hypervisor delays interrupt delivery.

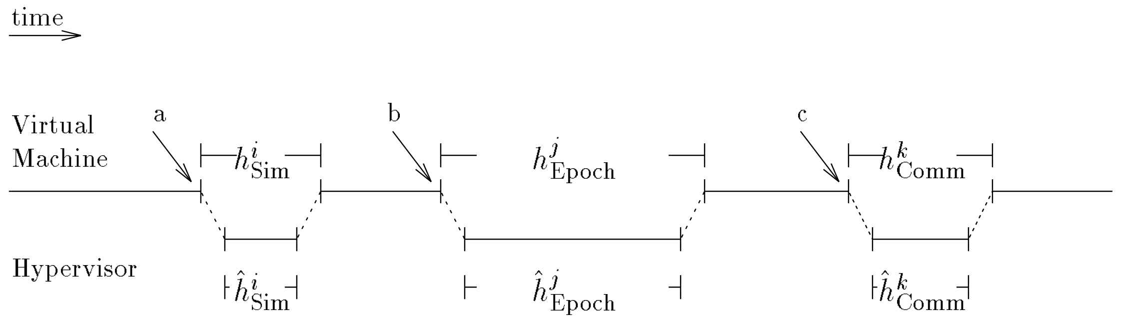
Normalized performance is the figure of merit. Let ET_C denote the elapsed time for a given workload when run on the bare hardware (the *control*), and let ET_H denote the elapsed time for the same workload when run with the VFT-enhanced hypervisor present. This workload is defined to have *normalized performance* of ET_H/ET_C. Thus, a normalized performance of 1.25 for a given workload indicates that virtual fault-tolerance adds 25% 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.

Two types of workload were selected so that we could understand the costs of the VFT protocol — a CPU-intensive workload and a Disk-I/O workload. The CPU-intensive workload allowed us to assess hypervisor overhead in the absense of I/O while keeping kernel activity, and its related virtualization cost, as low as possible. The Disk-I/O workload allowed us to measure the delays and assess the impact associated with delayed interrupt delivery for the reads and writes. In both cases, the workload involves a *dominant process* having known characteristics.

5.1 Common Execution Characteristics

5.1.1 Virtual Machine/Hypervisor Interaction

Operation of our system can be viewed as virtual machine execution alternating with hypervisor execution. The virtual machine executes until some event causes control to enter the hypervisor. At that point, the virtual machine is suspended and the



a. Virtual machine attempts a privileged instruction.
b. Virtual machine reaches an epoch boundary.
c. Communication interrupt arrives for the hypervisor.

Figure 5.1: Alternating Execution Cycle

hypervisor executes. The hypervisor processes the event; the virtual machine then resumes and the cycle may repeat. See Figure 5.1.

The events that cause the hypervisor to receive control are:

- **Sim:** Virtual machine instructions that must be simulated by the hypervisor. These include privileged instructions and memory accesses to I/O pages.

- **Int:** Interrupts for I/O that is initiated by the virtual machine. These events all cause the code of Figure 2.3 (page 28) to be executed.

- **Comm:** Interrupts resulting from communication between the hypervisors at the primary and backup. The processor's network controller generates an interrupt when each packet transmission completes and for each incoming packet received. (See Section 4.5, page 93.)

- **Epoch:** Epoch Traps. At the end of each epoch, the hypervisor performs the epoch-boundary processing of Figure 2.4 (page 30).

Figure 5.1 illustrates virtual machine execution being interrupted by three of the four event categories enumerated above, namely simulating a privileged instruction (labelled a), epoch-boundary processing (labelled b), and processing of a hypervisor communication interrupt (labelled c).

Let h^i_{A} denote time that elapses from the point where virtual machine execution is suspended until virtual machine execution is resumed for the ith occurrence of hypervisor execution due to event category A. This is the *hypervisor overhead* for the ith occurance of an event from category A. Further, let $\hat{h}^i_{\text{A}}$ denote the *hypervisor execution time* of the ith occurance of an event from category A.[1] Let n_{A} denote the number of incursions into the hypervisor for category A events during a run. For example, n_{Sim} is the number of instruction simulations performed by the hypervisor during a run, h^i_{Sim} is the hypervisor overhead for the ith instruction simulation, and $\hat{h}^i_{\text{Sim}}$ is the hypervisor execution time for the ith instruction simulation.

In our prototype, virtual time advances only while the virtual machine is executing. To accomplish this, the hypervisor excludes hypervisor execution time from the interval timer values seen by the virtual machine. Define the elapsed time of the virtual machine over the workload ET_V to be equal to ET_H minus the sum of the hypervisor execution times over the run.[2]

[1] $\hat{h}^i_{\text{A}}$ excludes the delays associated with entering and exiting the hypervisor from h^i_{A}.

[2] Thus, ET_V corresponds to the (virtual) time presented to the virtual machine by the hypervisor over a run.

By maintaining virtual time in this manner, ET_V approximates ET_C for a CPU-intensive workload. This implies that the time-related activities of the operating system kernel should be approximately the same when a CPU-intensive workload is executed with or without the hypervisor present. In particular, the number of clock interrupts and their associated kernel processing should be the same, simplifying the performance analysis below.

5.1.2 Experimental Setup and Obtaining Measurements

To measure the elapsed time of a workload running on the bare hardware, a control environment was set up. The control environment consisted of an unaltered version of HP–UX 8.05 on an HP–720 computer. The operating system was configured as a stand-alone system with no network interface, thereby eliminating unsolicited I/O from the network.

For each workload, the dominant process of the workload was executed 20 times; each such execution consituted a *control run*. The arithmetic mean of the elapsed time for these 20 control runs was used as an approximation of ET_C for the given workload.

To obtain measurements for the various components comprising hypervisor overhead, the dominant process of each workload was executed 20 times with the hypervisor present. Each execution defines a *measurement run*. The arithmetic mean of the true elapsed time as well as the elapsed time seen by the virtual machine over the 20 measurement runs was used as an approximation of ET_H and ET_V respectively for the given workload. The number VI of virtual machine instructions executed for the workload was also recorded for each run, and the arithmetic mean over the set of measurement runs was used to approximate it. A fixed epoch length[3] of 4096 virtual machine instructions was used for all measurement runs. We refer to this as the *measurement epoch length*.

Let $\hat{\Sigma}_{\mathrm{A}}$ denote the sum of the hypervisor execution time for event category A during a measurement run. During each measurement run, n_{A} and $\hat{\Sigma}_{\mathrm{A}}$ are recorded for event category A, where A is one of Sim, Epoch, Comm, or Int. The arithmetic mean over the set of 20 measurement runs is used to approximate the respective sample values. We can approximate $\hat{h}_{\mathrm{A}}$, the average hypervisor execution time for event category A, for a particular workload as follows:

$$\hat{h}_{\mathrm{Sim}} : \frac{\hat{\Sigma}_{\mathrm{Sim}}}{n_{\mathrm{Sim}}} \tag{5.1}$$

$$\hat{h}_{\mathrm{Epoch}} : \frac{\hat{\Sigma}_{\mathrm{Epoch}}}{n_{\mathrm{Epoch}}} \tag{5.2}$$

$$\hat{h}_{\mathrm{Comm}} : \frac{\hat{\Sigma}_{\mathrm{Comm}}}{n_{\mathrm{Comm}}} \tag{5.3}$$

$$\hat{h}_{\mathrm{Int}} : \frac{\hat{\Sigma}_{\mathrm{Int}}}{n_{\mathrm{Int}}} \tag{5.4}$$

5.1.3 Approximating Hypervisor Entry/Exit Overhead

For any incursion into the hypervisor, the time from suspension of the virtual machine until its subsequent resumption, h^i_{A}, includes all the overhead of entering and exiting the hypervisor as well as the processing time within the hypervisor. Hypervisor timing measurements, however, can only measure the hypervisor execution time, $\hat{h}^i_{\mathrm{A}}$. Consider the detailed example of a hypervisor incursion depicted in Figure 5.2.

[3] The current hypervisor implementation constrains the maximum epoch length to 8192 virtual machine instructions.

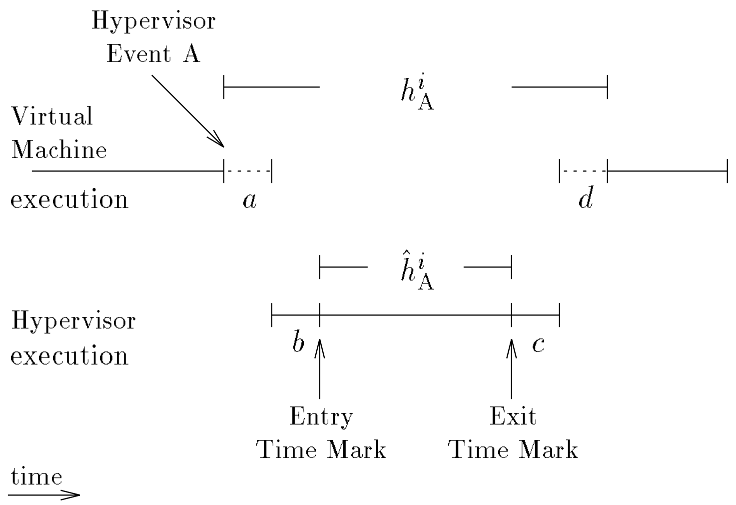


Figure 5.2: Hypervisor Entry/Exit Overhead

There are numerous sources of overhead associated with entering and exiting the hypervisor. The segment labelled a in the figure represents hardware delay associated with buffering and delivering an interruption. This includes saving the interruption context and invoking the appropriate handler. Segment b represents prologue code in the hypervisor, where portions of the virtual machine context are saved and a stack frame built for the subsequent hypervisor execution. At this point, an entry time mark (as read from the interval timer) can be recorded and is. Once hypervisor processing is complete, an exit time mark is taken and the difference from the entry time mark is computed, yielding the hypervisor execution time $\hat{h}^i_{\mathrm{A}}$ for this hypervisor incursion. The virtual machine stack and context are restored and the hypervisor is exited, as represented by segment c. Segment d represents the additional hardware delay associated with restarting the virtual machine. This would include delays in refilling the instruction pipeline and delays for higher data and instruction cache-misses as the virtual machine resumes execution.[4] The hypervisor entry/exit overhead is the sum of these four delays, or $a + b + c + d$.

Let h_{Entry} denote the average hypervisor entry/exit overhead per any hypervisor incursion. If we assume a fixed workload that is consistent between the control runs and the measurement runs, then the virtual time spent during a measurement run will exceed the time spent in a control run by the sum of the hypervisor entry/exit overhead $(a + b + c + d)$ over the run[5]. Thus, we can approximate h_{Entry} by

$$h_{\text{Entry}} : \frac{ET_V - ET_C}{n_{\text{Sim}} + n_{\text{Epoch}} + n_{\text{Comm}} + n_{\text{Int}}} \tag{5.5}$$

Our ability to do this calculation is another advantage to maintaining the elapsed virtual time ET_V by excluding hypervisor execution time.

By assuming that h_{Entry} is independent of the hypervisor-event category and is, on average, fixed over the workload, we approximate the average hypervisor overhead for each event category by including h_{Entry}:

$$h_{\text{Sim}} : h_{\text{Entry}} + \hat{h}_{\text{Sim}} \tag{5.6}$$

$$h_{\text{Epoch}} : h_{\text{Entry}} + \hat{h}_{\text{Epoch}} \tag{5.7}$$

$$h_{\text{Comm}} : h_{\text{Entry}} + \hat{h}_{\text{Comm}} \tag{5.8}$$

$$h_{\text{Int}} : h_{\text{Entry}} + \hat{h}_{\text{Int}} \tag{5.9}$$

[4]Although this is shown as a single delay before the virtual machine resumes execution, the delay is actually spread out over the initial virtual machine execution.

[5]The hypervisor execution time has already been excluded from the virtual time over the run.

5.2 CPU-Intensive Workload

5.2.1 The Measurements

Our first investigations concerned a CPU-intensive workload. A CPU-intensive workload was chosen to allow us to, as much as possible, quantify the overhead due to epoch-boundary processing and thus to explore the effect of epoch length on overhead. When a CPU-intensive workload is executing, the cost of virtualization, which is due to kernel processing, is reduced[6]. Since there is no I/O, the overhead due to processing virtual machine related I/O is eliminated (i.e. n_{Int} should be 0). Remaining hypervisor overhead is entirely due to epoch-boundary processing and to hypervisor communication.

The dominant process chosen for the CPU-intensive workload was the Dhrystone 2.1 benchmark; its primary loop executed 1 million times. The process was assigned a high real-time priority to reduce (unpredictable) interference from any Unix daemon processes. As mentioned previously, for the measurement runs, the measurement epoch length was set to 4096 virtual machine instructions; both the control and measurement experiments were repeated 20 times.

Table 5.1 shows the mean, standard deviation, coefficient of variation, and the 99% confidence interval for the elapsed times of the control and measurement exper-

[6]Note that the cost of virtualization is not eliminated entirely. Even when the dominant process is a CPU-intensive task, the kernel is entered for every clock interrupt to execute the HP–UX scheduler.

Table 5.1: CPU Intensive Workload Core Measurements

Term	Arithmetic Mean	Standard Deviation	Coefficient of Variation	Conf. Interval ($\alpha = 0.01$)
ET_C (μsec.)	8875187.7500	34.02766	3.83402E-06	19.59905
ET_H (μsec.)	57687293.4531	2698.44829	4.67772E-05	1554.23634
ET_V (μsec.)	10846419.0940	1188.62475	1.09587E-04	684.61708
VI (instr.)	428273651.4000	1167.33651	2.72568E-06	672.35560
n_{Entry} (count)	225248.0000	135.44818	6.01329E-04	78.01465
h_{Entry} (μsec.)	8.7514	0.00121	1.37759E-04	0.00069

Table 5.2: CPU Intensive Workload Sim Measurements

	Sim			
Term	Arithmetic Mean	Standard Deviation	Coefficient of Variation	Conf. Interval ($\alpha = 0.01$)
n_{Sim} (count)	108498.1500	133.34967	1.22905E-03	76.80596
$\hat{\Sigma}_{\text{Sim}}$ (μsec.)	690948.8629	840.31581	1.21618E-03	484.00015
$\hat{h}_{\text{Sim}}$ (μsec.)	6.3683	0.00021	3.25160E-05	0.00012
h_{Sim} (μsec.)	15.1197	0.00124	8.24963E-05	0.00072

iments and the number of virtual machine instructions for the CPU-intensive workload. This table also includes the value of h_{Entry} as calculated with equation (5.5).[7]

Tables 5.2 through 5.4 show the measurement information for each of the hypervisor-event categories. Note that there is no table for the Int event category since, as predicted for the CPU-intensive workload, $n_{\text{Int}} = 0$ for all measurement runs.

[7]By assuming ET_C to be a constant, we calculate h_{Entry} for all of the measurement runs and include its standard deviation, coefficient of variation, and 99% confidence interval. We perform this sensitivity analysis for all our calculated values to demonstrate that the variations do not combine adversely as we combine measured values.

Table 5.3: CPU Intensive Workload Epoch Measurements

Epoch				
Term	Arithmetic Mean	Standard Deviation	Coefficient of Variation	Conf. Interval ($\alpha = 0.01$)
n_{Epoch} (count)	104558.2000	0.41039	3.92501E-06	0.23638
$\hat{\Sigma}_{\text{Epoch}}$ (μsec.)	45361112.3515	1721.40714	3.79490E-05	991.48594
$\hat{h}_{\text{Epoch}}$ (μsec.)	433.8360	0.01636	3.77110E-05	0.00942
h_{Epoch} (μsec.)	442.5874	0.01680	3.79598E-05	0.00968

Table 5.4: CPU Intensive Workload Comm Measurements

Comm				
Term	Arithmetic Mean	Standard Deviation	Coefficient of Variation	Conf. Interval ($\alpha = 0.01$)
n_{Comm} (count)	222269.7000	3.31027	1.48930E-05	1.90663
$\hat{\Sigma}_{\text{Comm}}$ (μsec.)	6006797.3545	420.91893	7.00738E-05	242.43840
$\hat{h}_{\text{Comm}}$ (μsec.)	27.0248	0.00185	6.83749E-05	0.00106
h_{Comm} (μsec.)	35.7762	0.00263	7.36371E-05	0.00152

There is remarkably little variation in either the measured values or the calculated values; the largest coefficient of variation was less than 0.13% giving us confidence in the soundness of using their average values.

Our key metric, normalized performance, for this CPU-intensive workload with an epoch length of 4096 virtual machine instructions was a disappointing 6.50 — the overhead of the hypervisor and VFT protocols was very high. The largest component of this overhead was epoch-boundary processing. With our approximation of h_{Epoch} at 442.59 μsec and n_{Epoch} at 104558, epoch boundaries added approximately 46 seconds to the 8.8 seconds required for executing the benchmark of 4.2×10^8 instructions.

5.2.2 Building the Model

Given the above measurements, we can propose a model for normalized performance parameterized by epoch length EL. We first predict the elapsed time for the workload when the hypervisor is present; call this prediction $T_{\mathrm{CPU}}(EL)$. Our predicted normalized performance of the CPU-intensive workload, NP_{CPU}, is then given by

$$NP_{\mathrm{CPU}}(EL) : \frac{T_{\mathrm{CPU}}(EL)}{ET_C} \tag{5.10}$$

$T_{\mathrm{CPU}}(EL)$ is modelled as the sum of the elapsed time for the control and the sum of the hypervisor overheads over the workload. The hypervisor overheads in the model are all dervied relative to the primary hypervisor. This means that we are assuming that the backup virtual machine maintains pace with the primary, and thus that the primary does not incur delays possible in the flow control protocol presented in Section 4.6.

Hypervisor Overhead for Sim

We assume that the hypervisor overhead for simulating privileged instructions for the CPU-intensive workload is independent of epoch length. This assumption is justified as follows. During the CPU-intensive workload, privileged instructions are executed only during clock interrupt processing by the kernel. The number of such interruptions is determined by ET_C for control runs and by ET_V for measurement runs. Since virtual time progresses only when the virtual machine is executing, ET_V approximates ET_C and the number of clock interrupts over the workload is approximately the same both with and without the hypervisor present. Therefore, the number of clock interrupts should be independent of epoch length. And, the hypervisor overhead for simulating privileged instructions is independent of epoch length.

From this, we further assume that, on average, the time spent simulating privileged instructions is constant over the workload. Thus, the hyperivsor overhead for simulating instructions over the workload would be approximated by the term $n_{\mathrm{Sim}} \cdot h_{\mathrm{Sim}}$.

Hypervisor Overhead for Epoch

The hypervisor overhead for epoch-boundary processing is dependent on the number of epochs in the workload. For the CPU-intensive workload, we assume that the number of virtual machine instructions required to execute the workload is fixed, and independent of epoch length. The number of epochs over the workload is then given by $\frac{VI}{EL}$. If we assume that, on average, the time spent during epoch-boundary processing is constant, regardless of epoch length, then the hypervisor overhead for epoch-boundary processing over the workload is approximated by the term $\frac{VI}{EL} \cdot h_{\text{Epoch}}$.

Hypervisor Overhead for Comm

Assuming no communication errors, there are two Comm events for each message sent from the hypervisor at the primary to the hypervisor at the backup — one for the communcation interrupt upon completion of the transmit command by the network controller and another for the communication interrupt when the corresponding acknowledgement arrives from the backup. Likewise, there are two Comm events for each message sent from the hypervisor at the backup to the hypervisor at the primary — the first is the communication interrupt when the packet from the backup arrives at the primary and the second is the communication interrupt upon completion of the transmit of the acknowledgement from the primary to the backup.

Recall from Figure 2.4 (page 30) that on each epoch boundary, the hypervisor at the primary sends an epoch marker to the hypervisor at the backup. However, the hypervisor communication processing for the interrupts associated with these messages occurs during the epoch-boundary processing and has already been accounted for in h_{Epoch}. Thus, we should not include these interrupts again.

The backup sends messages to the primary for the purposes of flow control. (See Section 4.6, page 94.) One message is sent every WS epochs, where WS is the size of the epoch window used in the flow control algorithm[8]. If we assume that all communication interrupts associated with the flow control messages occur during virtual machine execution (i.e. they are not processed while the hypervisor is al-

[8] $WS = 16$ in the current prototype.

ready executing), and assume that, on average, hypervisor communication interrupt processing is constant, regardless of epoch length, then the hypervisor overhead for hypervisor communication processing over the workload is approximated by the term $2 \cdot \frac{\frac{VI}{EL}}{WS} \cdot h_{\text{Comm}}$.

Hypervisor Overhead for Int

For the CPU-intensive workload, we assume the number of I/O interrupts for the virtual machine to be zero, so the hypervisor overhead for processing such interrupts is also zero.

5.2.3 Putting it together

Combining these overhead terms with the elapsed time for the control experiment, we arrive at our model for the elapsed time with the hypervisor present:

$$T_{\text{CPU}}(EL) : ET_C + n_{\text{Sim}} \cdot h_{\text{Sim}} + \frac{VI}{EL} \cdot h_{\text{Epoch}} + 2 \cdot \frac{\frac{VI}{EL}}{WS} \cdot h_{\text{Comm}} \tag{5.11}$$

In Figure 5.3, we graph (5.10) for epoch lengths in the range of 1024 virtual machine instructions through 32768 virtual machine instructions. Although our current implementation cannot support epoch lengths greater than 8192, the longer epoch lengths are achievable. When there are 32768 virtual machine instructions in an epoch, a normalized performance of 1.84 is predicted by the model — a substantial improvement from the normalized performance of 6.50 observed at the measurement epoch length.

Also indicated on the graph of Figure 5.3 are measurements we made using our prototype for epoch lengths 1024, 2048, 4096, and 8192. The measurements agree well (all within 0.35%) with what equation (5.10) predicts, validating $NP_{\text{CPU}}(EL)$ of

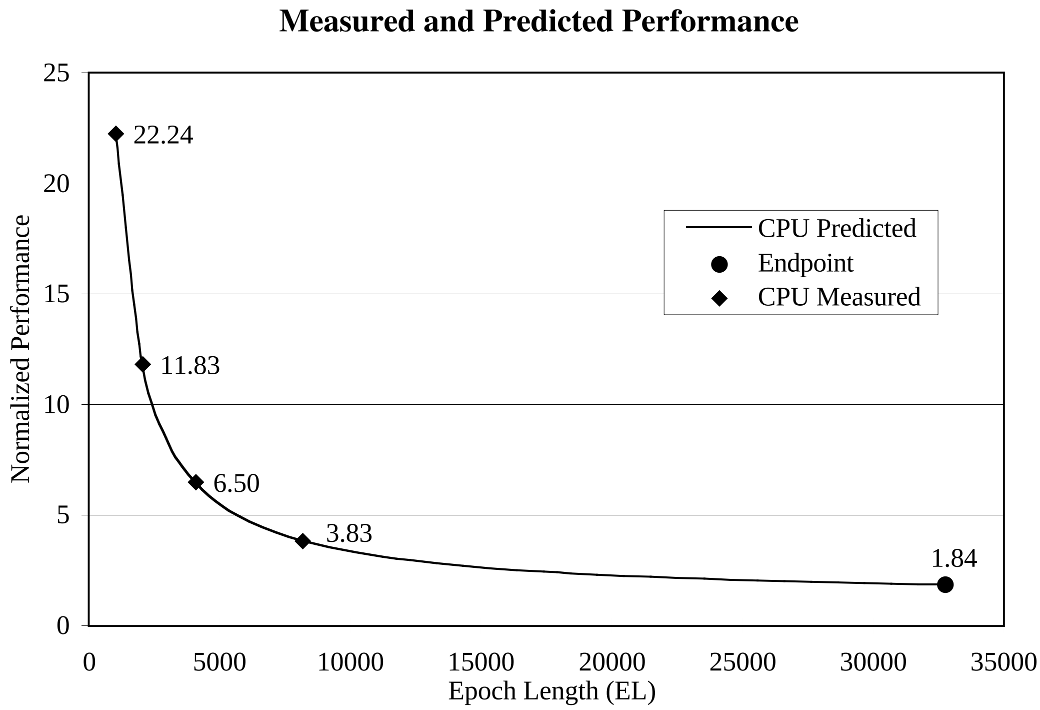


Figure 5.3: Measured and Predicted Performance of the CPU Intensive Workload

equation (5.10) and $T_{\mathrm{CPU}}(EL)$ of equation (5.11) for predicting performance of this workload.

The upper limit on epoch length is determined by the clock interrupt rate of the virtual machine. Since all interrupts, including the clock interrupt, must be delivered at epoch boundaries, the frequency of epoch boundaries must be at least as great as the frequency of clock interrupts expected by the virtual machine. For our prototype system running on HP–UX, the clock rate is 100 Hz and the actual instruction execution rate is roughly 50 MIPS. If we assume that virtual time is running at the same rate as real time, and accounting for epoch-boundary processing and the expected number of instruction simulations per epoch for the CPU intensive workload, we arrive at an upper limit for epoch lengths of approximately 385,000 instructions. Our predicted performance for that length epoch is 1.24. 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, our VFT protocols for replica-management are responsible for only 6% overhead above that required for implementing a virtual machine manager.

By far, the dominant cost in our model is epoch-boundary processing time. Longer epochs can significantly reduce this overhead. However, long epochs cause interrupt delivery to be delayed. In the following section, we examine the sensitivity of interrupt delivery delay to epoch length by examining Disk I/O-intensive workloads. Then, in Section 5.4, we develop models for alternative designs as alternatives for reducing the epoch-boundary processing overhead.

5.3 Disk-I/O 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 delay of deferring the delivery of every virtual machine I/O completion interrupt until the next epoch boundary. Third is the overhead associated with transferring the result of an input operation from the primary's hypervisor to the backup's hypervisor. The primary virtual machine must not proceed until this data has been received by the backup's hypervisor (per the WaitStable(Backup) of Figure 2.4, page 30).

When analyzing an I/O-intensive workload involving a disk, care must be taken to ensure that read or write requests of the dominant process actually propagate 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.

5.3.1 The Measurements

To simplify the process of obtaining measurements, we divide the Disk-I/O workload into a write-intensive workload and a read-intensive workload. The two workloads are very similar and are presented together in the discussion that follows.

The dominant process for the write-intensive workload is a benchmark that repeatedly writes 8192 byte blocks[9] to a disk file. The 16 megabyte file is preallocated

[9]This size matches the block transfer size used by HP–UX to and from the SCSI disk used in the prototype.

to prevent I/O operations by HP–UX during the workload's execution for allocating and mapping the file on the disk. In each *write cycle* of the benchmark, a disk block is randomly selected, a write is issued, and then the write completion is awaited. This is iterated 2048 times. For each write cycle, the *write CPU processing interval* begins with the point of delivery of the completion interrupt to the virtual machine from the previous write operation and ends with the memory-mapped I/O instruction to initiate the next write operation.

The dominant process for the read-intensive workload is analagous. A preallocated 16 megabyte disk file is again used. In each *read cycle* of the benchmark, a disk block is randomly selected, a read of a 8192 byte block is issued, and then the read completion is awaited. This is iterated 2048 times. For each read cycle, the *read CPU processing interval* begins with the point of delivery of the completion interrupt to the virtual machine from the previous read operation and ends with the memory-mapped I/O instruction to initiate the next read operation.

Figure 5.4 (a) depicts the write cycle of the write-intensive workload and Figure 5.4 (b) depicts the read cycle of the read-intensive workload. Note that the write and read cycles are defined in terms of I/O operations by the virtual machine. Thus, for example, if a read issued by the read benchmark did not result in a read by the virtual machine (as would be the case if the desired disk block were currently cached by the kernel), the virtual machine would remain in the read CPU processing interval.

For the measurement runs, the measurement epoch length was again set to 4096 virtual machine instructions; both the control and measurement experiments were repeated 20 times.

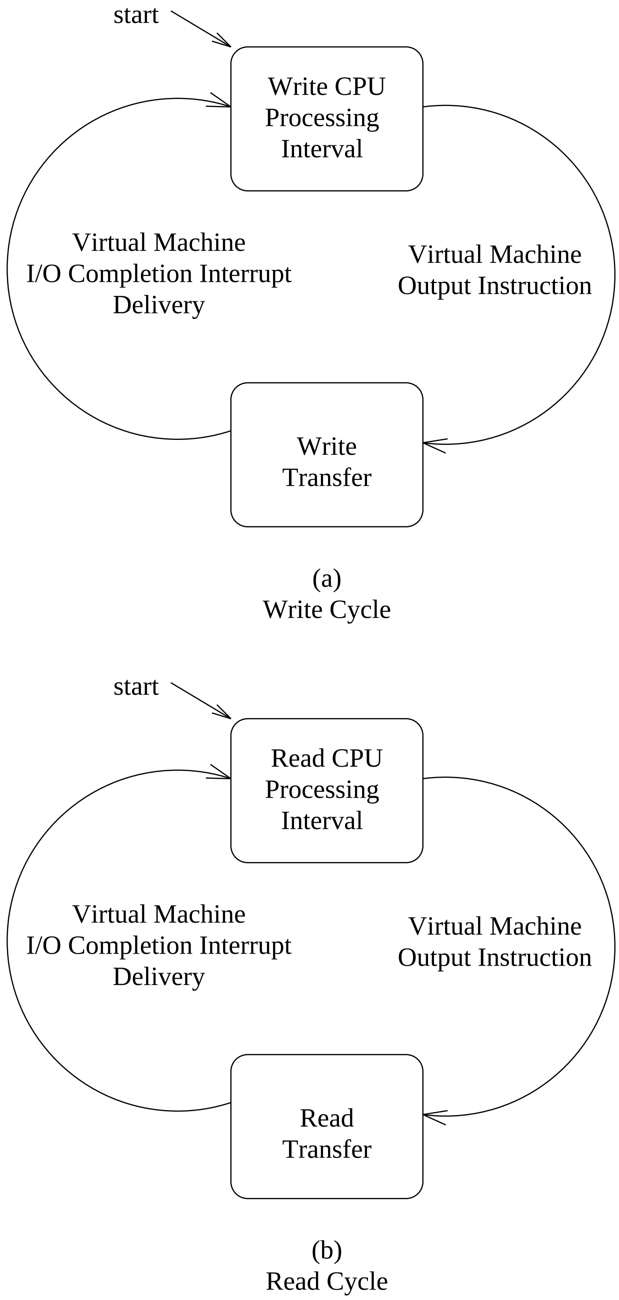


Figure 5.4: Write and Read Cycle for Disk-I/O Workload

Table 5.5: Disk-I/O Workload Core Measurements

Write-Intensive Workload				
Term	Arithmetic Mean	Standard Deviation	Coefficient of Variation	Conf. Interval ($\alpha = 0.01$)
n_W (count)	2049.3500	0.48936	2.38788E-04	0.28186
ET_C (μsec.)	54036250.0000	175564.91527	3.24902E-03	101120.84535
ET_H (μsec.)	90321808.6385	139918.22819	1.54911E-03	80589.27658
Read-Intensive Workload				
Term	Arithmetic Mean	Standard Deviation	Coefficient of Variation	Conf. Interval ($\alpha = 0.01$)
n_R (count)	1730.0500	0.22361	1.29249E-04	0.12879
ET_C (μsec.)	42679050.0000	418877.12996	9.81458E-03	241262.38101
ET_H (μsec.)	86617765.3356	628493.41428	7.25594E-03	361995.93326

Table 5.5 shows the mean, standard deviation, coefficient of variation, and 99% confidence interval for the elapsed time of the control and measurement experiments for both the write and read-intensive workloads. The table also shows the number of reads n_R and writes n_W actually issued from the virtual machine to the SCSI disk for the workloads.

The normalized performance was 1.67 for the write-intensive workload and 2.03 for the read-intensive workload. The worse performance in the read case is due to the transfer of the data blocks read from the hypervisor at the primary to the hypervisor at the backup. There is again remarkably little variation in the measured values, especially considering a disk I/O workload[10]; the largest coeffecient of variation was

[10]We would expect higher variation in both the control and measurements runs of the Disk-I/O intensive workload than the CPU-intensive workloads. Operations to the disk have higher latency and involve the mechanical operation of the disk itself. In addition, the caching of disk blocks implemented by the operating system kernel may perform differently from one run to the next depending on initial conditions and the policies involved.

less than 1%. Note that the number of writes issued by the virtual machine closely matches the 2048 writes requested by the write benchmark, while the number of reads issued by the virtual machine is only 85% of the 2048 reads requested by the read benchmark. The remainder of the read requests apparently were satisfied by the caching implemented by the operating system.

To make the analysis tractable and to isolate the I/O-specific overheads involved, we separate the measurements and overheads associated with the write and read CPU processing intervals over the workload from the I/O transfer and interrupt delivery overheads. In the control experiment, we expect the CPU processing interval to be very short and the majority of the time to be spent performing transfers. However, we expect a relatively high percentage of the instructions to concern I/O. These instructions will be privileged and, therefore, must be simulated by the hypervisor. Thus in the measurement runs, the CPU processing intervals should be much longer.

In the following sections, we partition the time during each cyle of the write or read benchmarks into the time spent in the CPU processing interval, the time spent during data transfer, and the delay in delivering interrupts to the virtual machine.

To help the reader distinguish between terms that apply to the entire run of the workload from the terms that apply to a single read or write cycle, we use uppercase term names to denote the former and lower case term names to denote the latter.

CPU Processing Interval Measurements

Let VI^W denote the total number of virtual machine instructions executed during the write CPU processing intervals over the write workload. Likewise, let VI^R denote the total number of virtual machine instructions executed during the read

CPU processing intervals over the read workload. These counts are recorded during each measurement run. We approximate the average number of virtual machine instructions executed during each CPU processing interval, vi^W for the write CPU processing interval and vi^R for the read CPU processing interval, as follows:

$$vi^W \quad : \quad \frac{VI^W}{n_W} \tag{5.12}$$

$$vi^R \quad : \quad \frac{VI^R}{n_R} \tag{5.13}$$

We next approximate the average elapsed time spent in the write CPU processing interval and the read CPU processing interval when run on the bare hardware, denoted et_C^W and et_C^R respectively. We assume that the rate of instruction execution $IRate$ during the CPU processing intervals is constant. This rate was obtained from measurements in the CPU-intensive workload — in particular, the number VI of virtual machine instructions required to complete the workload and the elapsed time ET_C of the control to execute that many instructions. Thus $IRate$ is approximated by VI/ET_C from the CPU-intensive workload. The average CPU processing interval is then given by:

$$et_C^W \quad : \quad \frac{vi^W}{IRate} \tag{5.14}$$

$$et_C^R \quad : \quad \frac{vi^R}{IRate} \tag{5.15}$$

Table 5.6 presents these measurements and calculated values.

Let N_{A}^W denote the number of hypervisor incursions due to event category A during write CPU processing intervals over a measurement run of the write-intensive workload. Further, let $\hat{\Sigma}_{\mathrm{A}}^W$ denote the sum of the hypervisor execution time for event category A during write CPU processing intervals over a measurement run

Table 5.6: Disk-I/O Workload CPU Processing Interval Core Measurements

Write-Intensive Workload				
Term	Arithmetic Mean	Standard Deviation	Coefficient of Variation	Conf. Interval ($\alpha = 0.01$)
VI^W (count)	36724587.7500	22923.96074	6.24213E-04	13203.60783
vi^W (count)	17920.1152	10.55504	5.89005E-04	6.07943
et_C^W (μsec.)	371.3616	0.21873	5.89005E-04	0.12599
Read-Intensive Workload				
Term	Arithmetic Mean	Standard Deviation	Coefficient of Variation	Conf. Interval ($\alpha = 0.01$)
VI^R (count)	38505173.4500	358925.85324	9.32150E-03	206731.99792
vi^R (count)	22256.6763	206.74498	9.28912E-03	119.07975
et_C^R (μsec.)	461.2289	4.28441	9.28912E-03	2.46771

of the write-intensive workload. Similarly, N_{A}^R denotes the sum of the number of hypervisor incursions due to event category A during read CPU processing intervals and $\hat{\Sigma}_{\mathrm{A}}^R$ denotes the sum of the hypervisor execution time for event category A during read CPU processing intervals over a measurement run of the read-intensive workload.

During each measurement run of the write-intensive workload, N_{A}^W and $\hat{\Sigma}_{\mathrm{A}}^W$ are recorded for event category A, where A is one of Sim, Epoch, or Comm.[11] Similarly, N_{A}^R and $\hat{\Sigma}_{\mathrm{A}}^R$ are recorded for each measurement run of the read-intensive workload. The arithmetic mean over the set of 20 measurement runs is used to approximate the respective sample values.

Let n_{A}^W represent the average number of hypervisor incursions due to event category A during each write CPU processing interval of the write-intensive workload and

[11]The Int event category is also measured and will be discussed in detail when interrupt delivery delay is approximated.

$\hat{h}^W_{\mathrm{A}}$ denote the average hypervisor execution time for event category A during each write CPU processing interval of the write-intensive workload. We can approximate these values for event categories Sim, Epoch, and Comm as follows:

$$n^W_{\mathrm{Sim}} \;:\; \frac{N^W_{\mathrm{Sim}}}{n_W} \tag{5.16}$$

$$n^W_{\mathrm{Epoch}} \;:\; \frac{N^W_{\mathrm{Epoch}}}{n_W} \tag{5.17}$$

$$n^W_{\mathrm{Comm}} \;:\; \frac{N^W_{\mathrm{Comm}}}{n_W} \tag{5.18}$$

$$\hat{h}^W_{\mathrm{Sim}} \;:\; \frac{\hat{\Sigma}^W_{\mathrm{Sim}}}{N^W_{\mathrm{Sim}}} \tag{5.19}$$

$$\hat{h}^W_{\mathrm{Epoch}} \;:\; \frac{\hat{\Sigma}^W_{\mathrm{Epoch}}}{N^W_{\mathrm{Epoch}}} \tag{5.20}$$

$$\hat{h}^W_{\mathrm{Comm}} \;:\; \frac{\hat{\Sigma}^W_{\mathrm{Comm}}}{N^W_{\mathrm{Comm}}} \tag{5.21}$$

Then, by using the value of h_{Entry} obtained from the CPU-intensive workload and by assuming that h_{Entry} is independent of the hypervisor-event category and remains, on average, fixed over the workload, we approximate the average hypervisor overhead during each write CPU processing interval h^W_{A} for each event category A by including h_{entry}:

$$h^W_{\mathrm{Sim}} \;:\; h_{\mathrm{Entry}} + \hat{h}^W_{\mathrm{Sim}} \tag{5.22}$$

$$h^W_{\mathrm{Epoch}} \;:\; h_{\mathrm{Entry}} + \hat{h}^W_{\mathrm{Epoch}} \tag{5.23}$$

$$h^W_{\mathrm{Comm}} \;:\; h_{\mathrm{Entry}} + \hat{h}^W_{\mathrm{Comm}} \tag{5.24}$$

The calculation of measurement values for the read CPU processing intervals of the read-intensive workload proceeds analagously:

$$n^R_{\mathrm{Sim}} \;:\; \frac{N^R_{\mathrm{Sim}}}{n_R} \tag{5.25}$$

$$n^R_{\text{Epoch}} \;:\; \frac{N^R_{\text{Epoch}}}{n_R} \tag{5.26}$$

$$n^R_{\text{Comm}} \;:\; \frac{N^R_{\text{Comm}}}{n_R} \tag{5.27}$$

$$\hat{h}^R_{\text{Sim}} \;:\; \frac{\hat{\Sigma}^R_{\text{Sim}}}{N^R_{\text{Sim}}} \tag{5.28}$$

$$\hat{h}^R_{\text{Epoch}} \;:\; \frac{\hat{\Sigma}^R_{\text{Epoch}}}{N^R_{\text{Epoch}}} \tag{5.29}$$

$$\hat{h}^R_{\text{Comm}} \;:\; \frac{\hat{\Sigma}^R_{\text{Comm}}}{N^R_{\text{Comm}}} \tag{5.30}$$

$$h^R_{\text{Sim}} \;:\; h_{\text{Entry}} + \hat{h}^R_{\text{Sim}} \tag{5.31}$$

$$h^R_{\text{Epoch}} \;:\; h_{\text{Entry}} + \hat{h}^R_{\text{Epoch}} \tag{5.32}$$

$$h^R_{\text{Comm}} \;:\; h_{\text{Entry}} + \hat{h}^R_{\text{Comm}} \tag{5.33}$$

Tables 5.7 through 5.9 present the measurement information for each of the hypervisor-event categories in the write and read CPU processing intervals of the Disk-I/O Workloads.

Transfer Time Measurements

We next approximate the average data transfer time for a single cycle of the write- and read-intensive workloads. From the control runs, we have the elapsed time for the entire benchmark. Using the approximatated time for each write CPU processing interval et^W_C we subtract the time for all write CPU processing intervals to obtain the sum of the write transfer times over the write-intensive workload X^W:

$$X^W : ET_C - (n_W \cdot et^W_C) \tag{5.34}$$

Table 5.7: Disk-I/O Workload CPU Processing Interval Sim Measurements

Write-Intensive Workload				
Term	Arithmetic Mean	Standard Deviation	Coefficient of Variation	Conf. Interval ($\alpha = 0.01$)
N^{W}_{Sim} (count)	1855016.2000	1627.009999	8.77087E-04	937.11563
$\hat{\Sigma}^{W}_{\text{Sim}}$ (μsec.)	11612284.0485	10230.512678	8.81008E-04	5892.51041
n^{W}_{Sim} (count)	905.1730	0.785744	8.68059E-04	0.45257
$\hat{h}^{W}_{\text{Sim}}$ (μsec.)	6.2599	0.000070	1.12333E-05	0.00004
h^{W}_{Sim} (μsec.)	15.0113	0.000070	4.68450E-06	0.00004
Read-Intensive Workload				
Term	Arithmetic Mean	Standard Deviation	Coefficient of Variation	Conf. Interval ($\alpha = 0.01$)
N^{R}_{Sim} (count)	1583173.1000	2362.533228	1.49228E-03	1360.75797
$\hat{\Sigma}^{R}_{\text{Sim}}$ (μsec.)	9913328.2720	14878.248267	1.50083E-03	8569.48576
n^{R}_{Sim} (count)	915.1025	1.375766	1.50340E-03	0.79241
$\hat{h}^{R}_{\text{Sim}}$ (μsec.)	6.2617	0.000104	1.66751E-05	0.00006
h^{R}_{Sim} (μsec.)	15.0131	0.000104	6.95490E-06	0.00006

Table 5.8: Disk-I/O Workload CPU Processing Interval Epoch Measurements

Write-Intensive Workload				
Term	Arithmetic Mean	Standard Deviation	Coefficient of Variation	Conf. Interval ($\alpha = 0.01$)
N^W_{Epoch} (count)	6190.5000	4.740642	7.65793E-04	2.73049
$\hat{\Sigma}^W_{\text{Epoch}}$ (μsec.)	2905201.5901	3160.611633	1.08791E-03	1820.43046
n^W_{Epoch} (count)	3.0207	0.002253	7.45960E-04	0.00130
$\hat{h}^W_{\text{Epoch}}$ (μsec.)	469.3000	0.413500	8.81100E-04	0.23817
h^W_{Epoch} (μsec.)	478.0514	0.413500	8.64971E-04	0.23817
Read-Intensive Workload				
Term	Arithmetic Mean	Standard Deviation	Coefficient of Variation	Conf. Interval ($\alpha = 0.01$)
N^R_{Epoch} (count)	7048.5000	87.487443	1.24122E-02	50.39050
$\hat{\Sigma}^R_{\text{Epoch}}$ (μsec.)	3359731.5031	38381.909885	1.14241E-02	22106.98628
n^R_{Epoch} (count)	4.0742	0.050439	1.23803E-02	0.02905
$\hat{h}^R_{\text{Epoch}}$ (μsec.)	476.6648	0.677601	1.42155E-03	0.39028
h^R_{Epoch} (μsec.)	485.4162	0.677601	1.39592E-03	0.39028

Table 5.9: Disk-I/O Workload CPU Processing Interval Comm Measurements

Write-Intensive Workload				
Term	Arithmetic Mean	Standard Deviation	Coefficient of Variation	Conf. Interval ($\alpha = 0.01$)
N^W_{Comm} (count)	207640.9000	514.106675	2.47594E-03	296.11213
$\hat{\Sigma}^W_{\text{Comm}}$ (μsec.)	5880689.6638	13813.745468	2.34900E-03	7956.35971
n^W_{Comm} (count)	101.3204	0.253049	2.49751E-03	0.14575
$\hat{h}^W_{\text{Comm}}$ (μsec.)	28.3215	0.005146	1.81689E-04	0.00296
h^W_{Comm} (μsec.)	37.0728	0.005146	1.38800E-04	0.00296
Read-Intensive Workload				
Term	Arithmetic Mean	Standard Deviation	Coefficient of Variation	Conf. Interval ($\alpha = 0.01$)
N^R_{Comm} (count)	187271.8500	2372.992299	1.26714E-02	1366.78212
$\hat{\Sigma}^R_{\text{Comm}}$ (μsec.)	5255491.9038	64442.020967	1.22618E-02	37116.93550
n^R_{Comm} (count)	108.2465	1.372036	1.26751E-02	0.79026
$\hat{h}^R_{\text{Comm}}$ (μsec.)	28.0636	0.012819	4.56796E-04	0.00738
h^R_{Comm} (μsec.)	36.8150	0.012819	3.48210E-04	0.00738

Table 5.10: Disk-I/O Workload Transfer Time Measurements

Write-Intensive Workload				
Term	Arithmetic Mean	Standard Deviation	Coefficient of Variation	Conf. Interval ($\alpha = 0.01$)
X^W (μsec.)	53275200.1193	475.057448	8.91705E-06	273.62079
x^W (μsec.)	25996.1466	6.288066	2.41885E-04	3.62176
Read-Intensive Workload				
Term	Arithmetic Mean	Standard Deviation	Coefficient of Variation	Conf. Interval ($\alpha = 0.01$)
X^R (μsec.)	41881100.7388	7438.081572	1.77600E-04	4284.14239
x^R (μsec.)	24208.0296	5.929909	2.44956E-04	3.41547

and similarly to obtain the sum of the read transfer times over the read-intensive workload X^R:

$$X^R : ET_C - (n_R \cdot et_C^R) \tag{5.35}$$

The average write and read transfer times, x^W and x^R respectively, are approximated using the above and the number of transfers measured:

$$x^W \ : \ \frac{X^W}{n_W} \tag{5.36}$$

$$x^R \ : \ \frac{X^R}{n_R} \tag{5.37}$$

These four derived measurements are presented in Table 5.10. On average, 26 msec. are required per write transfer and 24.2 msec. are required per read transfer, given the random access pattern of the benchmark. These numbers are consistent with the transfer times expected to and from a SCSI disk.

Deferred Interrupt Delivery Delay Measurements

To measure and then model the overhead due to processing interrupts for the virtual machine, we must distinguish between interrupts for the virtual machine that require minimal information to be sent from the hypervisor at the primary to the hypervisor at the backup from those interrupts for the virtual machine that require a significant amount of information to be transferred. In particular, the interrupts associated with the intermediate states of a SCSI transfer (See Section 4.3, page 86) or with completion of a disk write operation only require status information to be transferred from the primary to the backup. The interrupt associated with the completion of a disk read operation, however, requires that the result of the read be sent from the primary to the backup. Let Int_{min} represent the hypervisor-event category associated with processing an interrupt for I/O initiated by the virtual machine with minimal state to be transferred and Int_{max} represent the hypervisor-event category associated with processing a completion interrupt for I/O initiated by the virtual machine for a disk read request.

We record $n_{\text{Int}_{min}}$, $\hat{\Sigma}_{\text{Int}_{min}}$, $n_{\text{Int}_{max}}$, $\hat{\Sigma}_{\text{Int}_{max}}$ for each measurement run in the write- and read-intensive workloads and use the arithmetic mean over the set of 20 measurement runs to approximate their respective sample values. These are combined to approximate the average hypervisor execution time for these event categories:

$$\hat{h}_{\text{Int}_{min}} \;:\; \frac{\hat{\Sigma}_{\text{Int}_{min}}}{n_{\text{Int}_{min}}} \tag{5.38}$$

$$\hat{h}_{\text{Int}_{max}} \;:\; \frac{\hat{\Sigma}_{\text{Int}_{max}}}{n_{\text{Int}_{max}}} \tag{5.39}$$

Table 5.11: Disk-I/O Workload Int_{min} Measurements

Write-Intensive Workload				
Term	Arithmetic Mean	Standard Deviation	Coefficient of Variation	Conf. Interval ($\alpha = 0.01$)
$N^W_{\text{Int}_{min}}$ (count)	8098.5000	10.10731	1.24805E-03	5.82155
$\hat{\Sigma}^W_{\text{Int}_{min}}$ (μsec.)	157565.0319	150.42115	9.54661E-04	86.63869
$n^W_{\text{Int}_{min}}$ (count)	3.9517	0.00491	1.24420E-03	0.00283
$\hat{h}^W_{\text{Int}_{min}}$ (μsec.)	19.4561	0.01915	9.84332E-04	0.01103
Read-Intensive Workload				
Term	Arithmetic Mean	Standard Deviation	Coefficient of Variation	Conf. Interval ($\alpha = 0.01$)
$N^R_{\text{Int}_{min}}$ (count)	4599.0500	9.09294	1.97713E-03	5.23730
$\hat{\Sigma}^R_{\text{Int}_{min}}$ (μsec.)	35845.7289	103.17764	2.87838E-03	59.42765
$n^R_{\text{Int}_{min}}$ (count)	2.6583	0.00520	1.95806E-03	0.00300
$\hat{h}^R_{\text{Int}_{min}}$ (μsec.)	7.7941	0.00886	1.13757E-03	0.00511

Tables 5.11 and 5.12 present the derived Int measurements for the write- and read-intensive workloads. During the measurement runs of the write-intensive workload, there were no disk reads submitted by the virtual machine. Thus Table 5.12, which details measurements for Int_{max}, only includes information for the read-intensive workload.

For the read-intensive workload, each read operation requires a transfer of the disk block data from the hypervisor at the primary to the hypervisor at the backup. This transfer begins when the disk read I/O completion interrupt arrives at the primary. The transfer overlaps virtual machine execution and then, at the subsequent epoch boundary, the hypervisor at the primary waits until the disk block data and the subsequent epoch marker have been received by the hypervisor at the backup.

Table 5.12: Disk-I/O Workload Int_{max} Measurements

Read-Intensive Workload				
Term	Arithmetic Mean	Standard Deviation	Coefficient of Variation	Conf. Interval ($\alpha = 0.01$)
$N^R_{\text{Int}max}$ (count)	1709.7500	5.61834	3.28606E-03	3.23602
$\hat{\Sigma}^R_{\text{Int}max}$ (μsec.)	310535.4052	1066.63552	3.43483E-03	614.35445
$n^R_{\text{Int}max}$ (count)	0.9883	0.00326	3.30350E-03	0.00188
$\hat{h}^R_{\text{Int}max}$ (μsec.)	181.6262	0.10917	6.01067E-04	0.06288

We are interested in determining the time spent by the hypervisor at the primary during epoch boundary processing waiting for all interrupt data and the subsequent epoch marker to be received by the hypervisor at the backup. We measure the time, within epoch-boundary processing, starting with the send of the epoch marker and ending with the release of the WaitStable(Backup), distinguishing between two types of epoch-boundary processing:

1. Those epoch boundaries that do not follow the epoch of arrival of a disk read I/O completion interrupt.

2. Those epoch boundaries that do follow the epoch of arrival of a disk read I/O completion interrupt.

The difference of the average of these two measurements yields the additional waiting time due to the transfer of the result of the disk read.

Let WtRT represent the event associated with *stability-wait* round trip from the send of the epoch marker to the termination of the WaitStable in the nominal case.[12]

[12]Note that h_{WtRT} is a component of h_{Epoch} and we must be careful not to count this time twice in the model.

Let WtRTIO represent the event associated with stability-wait round trip at the epoch boundary following a disk read I/O completion interrupt. We record n_{WtRT}, Σ_{WtRT}, n_{WtRTIO}, Σ_{WtRTIO} for each measurement run in the read-intensive workload and use the arithmetic mean over the set of 20 measurement runs to approximate their respective sample values. These are combined to approximate the average elapsed time for these event categories and for the additional delay h_{Wait} for the read-intensive workload:

$$h_{\mathrm{WtRT}} : \frac{\Sigma_{\mathrm{WtRT}}}{n_{\mathrm{WtRT}}} \tag{5.40}$$

$$h_{\mathrm{WtRTIO}} : \frac{\Sigma_{\mathrm{WtRTIO}}}{n_{\mathrm{WtRTIO}}} \tag{5.41}$$

$$h_{\mathrm{Wait}} : h_{\mathrm{WtRTIO}} - h_{\mathrm{WtRT}} \tag{5.42}$$

Table 5.13 presents the measured and calculated values for the stability-wait samples. The additional wait delay in the read case is a disappointing 7.2 msec per read operation. During this time, the full 8192 byte block read by the primary is transferred to the backup over the Ethernet communication link, which requires 9 data packets plus acknowledgements. It may be possible to tune the sliding window protocol employed in the prototype to reduce this overhead, but doing so was regarded as beyond the scope of this study.

5.3.2 Building the Model

Given the above measurements, we can postulate a model for normalized performance of the Disk-I/O workloads parameterized by epoch length EL. For each write cycle of the write-intensive workload, we predict (i) the elapsed time spent in the write CPU processing interval $t^{W}_{\mathrm{CPU}}(EL)$ and (ii) the delays associated with defer-

Table 5.13: Disk-I/O Workload Stability-Wait Measurements

Read-Intensive Workload				
Term	Arithmetic Mean	Standard Deviation	Coefficient of Variation	Conf. Interval ($\alpha = 0.01$)
n_{WtRT} (count)	7048.5000	87.48744	1.24122E-02	50.39050
Σ_{WtRT} (μsec.)	3067368.4704	35142.29161	1.14568E-02	20241.05005
h_{WtRT} (μsec.)	435.1854	0.63529	1.45982E-03	0.36591
n_{WtRT1} (count)	1730.0500	0.22360	1.29249E-04	0.12879
Σ_{WtRT1} (μsec.)	13227533.3368	8938.24839	6.75731E-04	5148.19964
h_{WtRT1} (μsec.)	7645.7522	5.30006	6.93204E-04	3.05270
h_{Wait} (μsec.)	7210.5668	5.34580	7.41385E-04	3.07905

ring interrupts for each transfer $d^W(EL)$. These are added to our approximation of the average elapsed time for the transfer. If we assume that these terms are, on average, constant over the workload, the predicted normalized performance of the write-intensive workload is given by:

$$NP_W(EL) : \frac{n_W \cdot \left(t^W_{\mathrm{CPU}}(EL) + d^W(EL) + x^W\right)}{ET_C} \tag{5.43}$$

Similarly, for each read cycle of the read-intensive workload, we predict (i) the elapsed time spent in the read CPU processing interval $t^R_{\mathrm{CPU}}(EL)$ and (ii) the delays associated with deferring interrupts for each transfer $d^R(EL)$ These are added to our approximation of the average elapsed time for the transfer. If we assume that these terms are, on average, constant over the workload, the predicted normalized performance of the read-intensive workload is given by:

$$NP_R(EL) : \frac{n_R \cdot \left(t^R_{\mathrm{CPU}}(EL) + d^R(EL) + x^R\right)}{ET_C} \tag{5.44}$$

CPU Processing Interval

Each CPU processing interval is essentially the same as a portion of a CPU-intensive workload. Any differences can be attributed to the particulars of the number of virtual machine instructions in the interval and the number and mix of instructions that must be simulated — information that is given in the measurements above. By assuming that the measurements are, on average, constant over the CPU processing interval and independent of epoch length, we predict the average execution time of a single write CPU processing interval parameterized by epoch length

$$t^W_{\text{CPU}}(EL) : et^W_C + \left(n^W_{\text{Sim}} \cdot h^W_{\text{Sim}}\right) + \left(\frac{vi^W}{EL} \cdot h^W_{\text{Epoch}}\right) + \left(2 \cdot \frac{\frac{vi^W}{EL}}{WS} \cdot h^W_{\text{Comm}}\right) \qquad (5.45)$$

and for a single read CPU processing interval parameterized by epoch length

$$t^R_{\text{CPU}}(EL) : et^R_C + \left(n^R_{\text{Sim}} \cdot h^R_{\text{Sim}}\right) + \left(\frac{vi^R}{EL} \cdot h^R_{\text{Epoch}}\right) + \left(2 \cdot \frac{\frac{vi^R}{EL}}{WS} \cdot h^R_{\text{Comm}}\right) \qquad (5.46)$$

Note the similarity in the form of these equations to equation (5.11).

Deferred Interrupt Delivery Delay

Recall from Section 4.3, page 86 that, for each transfer between a SCSI disk and the HP machine, three interrupts occur between issuing the input or output request and the delivery of the CmdComplete interrupt: the Disconnect interrupt, the Reconnect interrupt, and the CmdComplete interrupt. Each will incur a delay due to interrupt delivery being deferred until the next epoch boundary when the hypervisor is present.[13]

[13]It may be argued that the Disconnect interrupt delay is concurrent with the seek of the SCSI disk and thus would not differ between the control and measurement runs of the workload. However, since this is not certain, we felt that a more conservative prediction that includes the Disconnect interrupt delay would be more appropriate.

For the write-intensive workload, all three interrupts require minimal state transfer from the hypervisor at the primary to the hypervisor at the backup. Let $d_{\mathrm{Int}_{min}}$ denote this delay. Thus the interrupt delivery delay for each write cycle of the write-intensive workload is given by:

$$d^W(EL) : 3 \cdot d_{\mathrm{Int}_{min}}(EL) \tag{5.47}$$

In the read-intensive workload, the Disconnect and Reconnect interrupts require minimal state transfer while the CmdComplete interrupt requires significant state transfer from the hypervisor at the primary to the hypervisor at the backup. Let $d_{\mathrm{Int}_{max}}$ denote the delay for the read operation I/O completion. Then, the interrupt delivery delay for each read cycle of the read-intensive workload is given by:

$$d^R(EL) : 2 \cdot d_{\mathrm{Int}_{min}}(EL) + d_{\mathrm{Int}_{max}} \tag{5.48}$$

In modelling $d_{\mathrm{Int}_{min}}$ and $d_{\mathrm{Int}_{max}}$, we must include all delays from interrupt arrival to interrupt delivery that are due to the hypervisor (and thus would not be incurred in the control runs). The sequence of events from interrupt arrival to interrupt delivery is the following:

1. The hardware saves the interrupt context and the hypervisor interrupt handler is invoked.

2. The hypervisor interrupt handler corresponding to Int_{min} or Int_{max} is executed. During this time, the transfer of state from the hypervisor at the primary to the hypervisor at the backup begins.

3. The hardware restores the interrupt context.

4. The virtual machine resumes execution. We assume that, on average, the virtual machine will execute one half an epoch to the next epoch boundary.

5. Epoch-boundary processing is performed. At the end of the epoch-boundary processing, the primary must wait until information sent to the backup is stable. When the virtual machine is resumed, the interrupt will then be delivered.

First note that control runs of the workload still receive the interrupt and incur the hardware delay in saving interrupt context and delivering the interrupt to the machine. Likewise, the hardware delays associated with resuming interrupted execution after interrupt processing are incurred during the control experiments. We assume that these delays in the control are approximated by h_{Entry} and thus use $\hat{h}_{\text{Int}_{min}}$ as the first component of $d_{\text{Int}_{min}}$ and $\hat{h}_{\text{Int}_{max}}$ as the first component of $d_{\text{Int}_{max}}$.

Once the hypervisor at the primary has handled the interrupt, it must wait until the next epoch boundary to deliver the interrupt to the virtual machine. This delay is dependent on epoch length EL. We assume a constant rate of instruction execution and also assume that, on average, the interrupt must be deferred by one half of an epoch. This delay, then, is modelled by $\frac{EL}{2 \cdot IRate}$.

The interrupt is delivered to the virtual machine upon completion of the epoch boundary processing. By including the epoch-boundary processing term, we obtain:

$$d_{\text{Int}_{min}}(EL) : \hat{h}_{\text{Int}_{min}} + \frac{EL}{2 \cdot IRate} + h_{\text{Epoch}} \tag{5.49}$$

In the case of an I/O completion interrupt for a disk read operation, there is additional overhead for transferring the result of the disk read from the hypervisor at the primary to the hypervisor at the backup. For the measurement epoch length of 4096 virtual machine instructions, the delay for delivering a disk read I/O completion interrupt is given by $\hat{h}_{\mathrm{Int}max} + \frac{4096}{2 \cdot IRate} + h_{\mathrm{Epoch}} + h_{\mathrm{Wait}}$. This sum approximates the time required to reliably communicate the result of the disk read from the primary to the backup, and this communication time should be independent of epoch length. (It depends on the size of the disk block, and the underlying communications infrastructure, which are assumed to remain fixed for the read-intensive workload.)

When the epoch length varies, the interrupt delivery delay for a disk read I/O completion interrupt will not differ from the transfer communication time unless the remainder of the epoch is so long that it's execution time completely hides the communication time. Recall the measurements from the previous section that show h_{Wait}, which is the dominant component of the communication time calculated above, is approximately 7.2 msec. Execution time for half an epoch at the epoch length upper bound is only around 4 msec. Thus, for practical purposes, the interrupt delivery delay for a disk read I/O completion interrupt is independent of epoch length and is given as follows:

$$d_{\mathrm{Int}max} : \hat{h}_{\mathrm{Int}max} + \frac{4096}{2 \cdot IRate} + h_{\mathrm{Epoch}} + h_{\mathrm{Wait}} \qquad (5.50)$$

5.3.3 Putting It Together

In Figure 5.5, we graph equations (5.43) and (5.44) for epoch lengths in the range of 1024 virtual machine instructions through 32768 virtual machine instructions. When there are 32768 virtual machine instructions in an epoch, a normalized performance of 1.98 is predicted for the read-intensive workload and a normalized performance of 1.63 is predicted for the write-intensive workload.

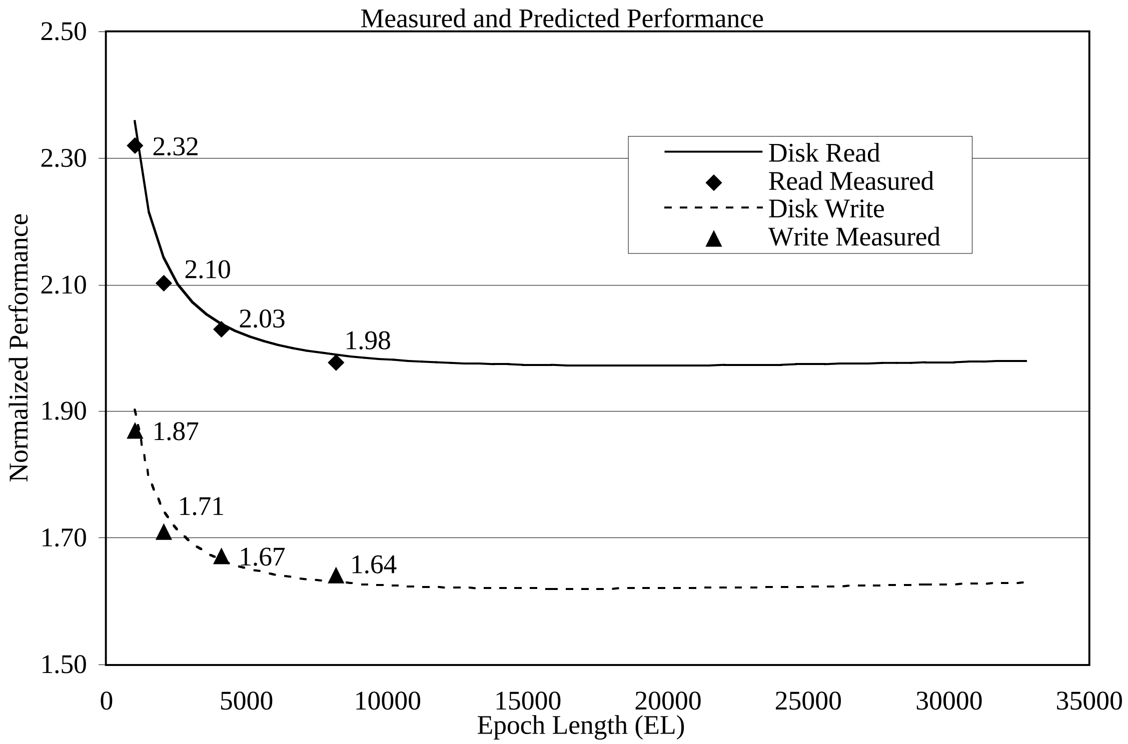


Figure 5.5: Measured and Predicted Performance of the Disk-I/O Workloads

Also indicated on the graph of Figure 5.3 are measurements we made using our prototype with epoch lengths of 1024, 2048, 4096, and 8192. The measurements are each within 1.9% of what is predicted by NP_W and NP_R.

As with the CPU-intensive workload, longer epochs lead to better normalized performance. This is because, in our models, $t^W_{\mathrm{CPU}}(EL)$ and $t^R_{\mathrm{CPU}}(EL)$ dominate. But another trend is also visible. Increases to epoch length EL cause $d^W(EL)$ and $d^R(EL)$ to increase, due to longer delays before interrupts are delivered. This 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 $t^W_{\mathrm{CPU}}(EL)$ and $t^R_{\mathrm{CPU}}(EL)$ terms would ameliorate the normalized performance. Normalized performance for the Disk-I/O workloads never goes as low as for the CPU-intensive workload, because of the high percentage of hypervisor-simulated instructions in the CPU processing interval.

5.4 Alternative Designs

The predominant overhead for the replica-coordination protocols comes from epoch-boundary processing. We have seen that using long epochs reduces the number of epochs required for a fixed workload and thus reduces the overall epoch-boundary processing overhead. Alternatively, we can seek designs that reduce the cost at each epoch boundary.

Within epoch-boundary processing, the largest overhead comes from the WaitStable, where the hypervisor at the primary must await acknowledgements for all messages previously sent to the hypervisor at the backup. This suggests searching for ways to either speed up the latency associated with the communication or to find an alternative design that obviates the need for the wait.

In the next section, we partition epoch-boundary processing into component overhead terms and arrive at values for these terms using measurement runs of the CPU-intensive workload. The remaining sections propose designs that attempt to reduce the overhead for epoch-boundary processing. We then build models for these alternative designs.

build models that use these measurements and are designed to reduce epoch-boundary processing.

5.4.1 Epoch-Boundary Processing Decomposition

Epoch-boundary processing at the hypervisor of the primary can be decomposed into the following key activities:

1. Prologue activity. This includes all activity from the Entry time mark until the setup required to send the epoch marker.

2. Communication setup. This includes all interaction with the network controller to send the epoch marker and ends when the network packet for the epoch marker "hits the wire".

3. Epoch Marker transmission time. This is the period from the end of (2) until the epoch marker packet is received by the controller at the backup.

4. Backup epoch marker processing time. This is the period from the end of (3) until the ackowledgement packet "hits the wire".

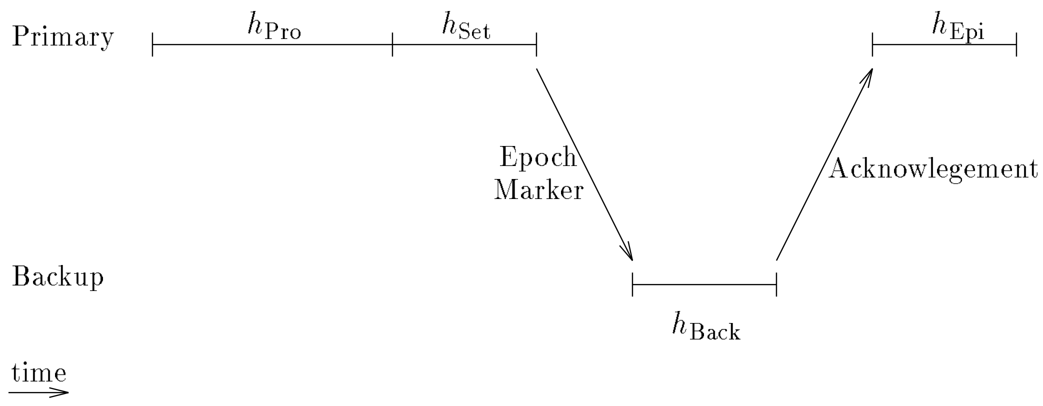


Figure 5.6: Epoch-Boundary Processing Decomposition

5. Acknowledgement transmission time. This is the period from the end of (4) until the acknowledgement packet is completely received by the network controller at the primary.

6. Epilogue activity. This includes all hypervisor processing at the primary from the end of (5) until the Exit time mark of epoch-boundary processing.

Let h_{Pro} denote the average execution time of item (1) over the CPU-intensive workload, h_{Set} denote the average execution time of item (2) over the CPU-intensive workload, h_{Trans} denote the average elapsed time of the sum of items (3) and (5), h_{Back} denote the average execution time of (4), and h_{Epi} denote the average execution time of (6). Figure 5.6 depicts this epoch-boundary processing decomposition.

Let Σ_{Pro}, Σ_{Set}, and Σ_{Epi} denote the sum of the hypervisor execution time for prologue processing, communication setup, and epilogue processing during a measurement run. During each measurement run Σ_{Pro}, Σ_{Set}, and Σ_{Epi} are recorded. The arithmetic mean over the set of 20 measurement runs is used to approximate the respective sample values. Dividing each by the number of epochs n_{Epoch} yields approximations for the individual average execution times:

$$h_{\text{Pro}} \;:\; \frac{\Sigma_{\text{Pro}}}{n_{\text{Epoch}}} \tag{5.51}$$

$$h_{\text{Set}} \;:\; \frac{\Sigma_{\text{Set}}}{n_{\text{Epoch}}} \tag{5.52}$$

$$h_{\text{Epi}} \;:\; \frac{\Sigma_{\text{Epi}}}{n_{\text{Epoch}}} \tag{5.53}$$

The average communication transmission time h_{Trans} can be approximated using the transmission rate of the prototype's Ethernet as the communications medium and the size of the epoch marker and acknowlegement packets (including Ethernet preamble and postamble bits). If we let S_{EM} denote the size of the epoch marker packet, S_{Ack} denote the size of the acknowledgement packet, and R_{Enet} denote the transmission rate of Ethernet (at 10 Megabits per second), then h_{Trans} is approximated by:

$$h_{\text{Trans}} : \frac{S_{\text{EM}} + S_{\text{Ack}}}{R_{\text{Enet}}} \tag{5.54}$$

Since all the terms in this approximation are constants, no measurement runs are required to obtain h_{Trans}, whose value is 131.48 μsec. based on the packet sizes used in the prototype implementation.

We obtain an approximation of h_{Back} by using our approximation of the hypervisor execution time of epoch boundary processing $\hat{h}_{\text{Epoch}}$ and subtracting the other components of epoch boundary processing as approximated above:

$$h_{\text{Back}} : \hat{h}_{\text{Epoch}} - \left(h_{\text{Pro}} + h_{\text{Set}} + h_{\text{Trans}} + h_{\text{Epi}}\right) \tag{5.55}$$

Table 5.14 shows the mean, standard deviation, coefficient of variation, and the 99% confidence interval for the terms discussed above.

Table 5.14: Epoch-Boundary Processing Component Measurements

Term	Arithmetic Mean	Standard Deviation	Coefficient of Variation	Conf. Interval ($\alpha = 0.01$)
Σ_{Pro} (μsec.)	2618325.3342	69.21060	2.64332E-05	39.86351
h_{Pro} (μsec.)	25.0416	0.00065	2.60678E-05	0.00038
Σ_{Set} (μsec.)	4421598.6698	97.52565	2.20567E-05	56.17225
h_{Set} (μsec.)	42.2880	0.00090	2.12793E-05	0.00052
Σ_{Epi} (μsec.)	1467813.4932	125.87602	8.57575E-05	72.50133
h_{Epi} (μsec.)	14.0385	0.00120	8.57906E-05	0.00069
h_{Back} (μsec.)	220.9863	0.01653	7.48132E-05	0.00952

5.4.2 Faster Communication

One alternative for reducing the time spent in epoch-boundary processing is to speed up the communication between the hypervisor at the primary and the hypervisor at the backup. This in turn would lead to better normalized performance. Consider, for example, the use of an ATM link between the primary and backup machines. Since the ATM link has a transmission rate of 155 Mbps as compared with the Ethernet's 10 Mbps, one would expect the transmission time to be greatly reduced and for performance to improve.

Let $Tr(R)$ denote the transmission time of the epoch marker packet plus the transmission time of the acknowledgement packet for a communication medium whose transmission rate is R. The transmission time is given by:

$$Tr(R) : \frac{S_{\mathrm{EM}} + S_{\mathrm{Ack}}}{R} \tag{5.56}$$

Epoch-boundary processing is then parameterized by the communication medium transmission rate:

$$h_{\text{Epoch}}(R) : h_{\text{Pro}} + h_{\text{Set}} + Tr(R) + h_{\text{Back}} + h_{\text{Epi}} \tag{5.57}$$

Equation 5.11 from the CPU-intensive workload is then modified to include the communication rate parameter R:

$$T_{\text{CPU}}(EL, R) : ET_C + n_{\text{Sim}} \cdot h_{\text{Sim}} + \frac{VI}{EL} \cdot h_{\text{Epoch}}(R) + 2 \cdot \frac{\frac{VI}{EL}}{WS} \cdot h_{\text{Comm}} \tag{5.58}$$

as is the definition of normalized performance:

$$NP_{\text{CPU}}(EL, R) : \frac{T_{\text{CPU}}(EL, R)}{ET_C} \tag{5.59}$$

The result is a model of normalized performance parameterized by both epoch length and communication rate. This model is a generalization of the CPU-intensive model presented at the beginning of this chapter.

In Figure 5.7, we graph equation (5.59) for two values of communication transmission rate R, one at the Ethernet rate of 10 Mbps and one at the ATM transmission rate of 155 Mbps, and over epoch lengths in the range of 1024 virtual machine instructions through 32768 virtual machine instructions.

One can see some improvement, especially at the shorter epoch lengths. For example, at 4096 instructions per epoch, the Ethernet has a predicted normalized performance of 6.45 while the ATM has a predicted normalized performance of 5.00 — an improvement of 22.46%. At 32768 instructions per epoch, the Ethernet normalized performance is predicted to be 1.84 and ATM normalized performance is predicted to be 1.66, corresponding to a 9.83% improvement. At the epoch length upper bound,

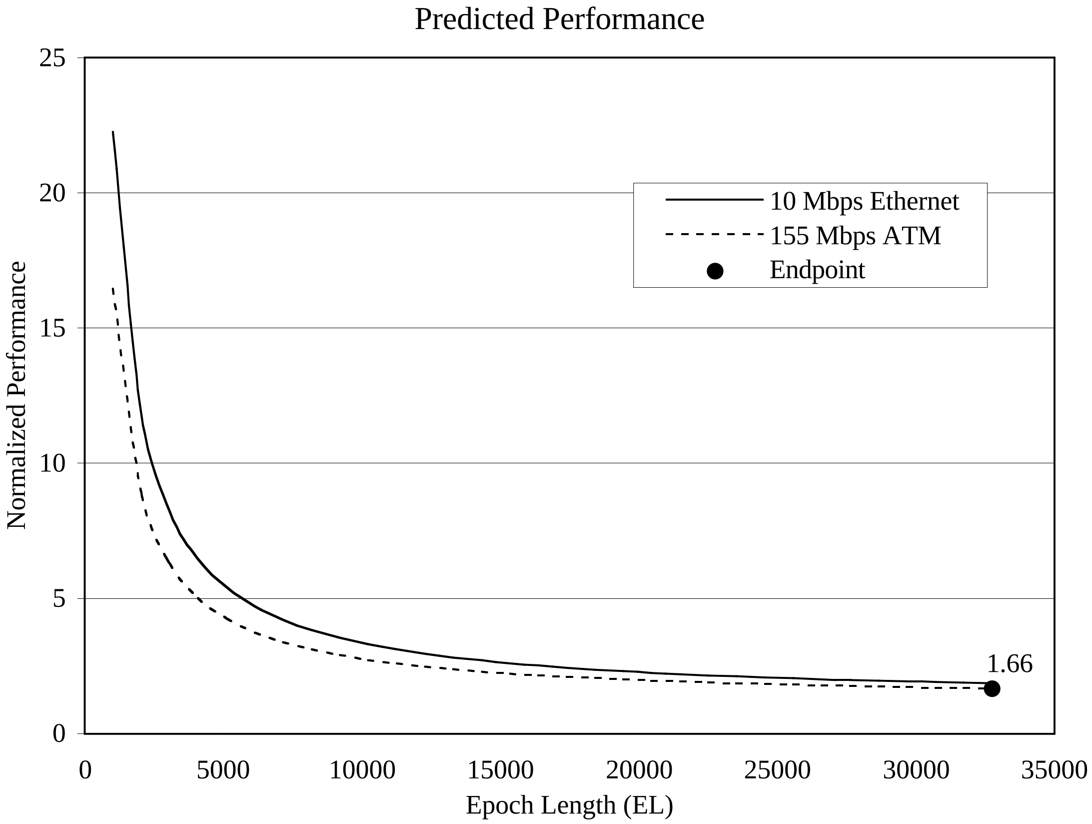


Figure 5.7: Faster Communication

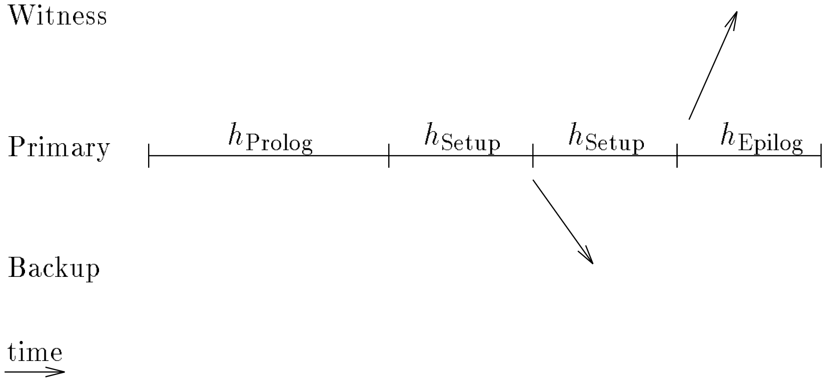


Figure 5.8: Witness Model Epoch Boundary Processing

we saw that the Ethernet has a predicted normalized performance of 1.24. The ATM, as predicted by the model, improves this to 1.23.

5.4.3 Witness Processor

There would be considerable gain in performance if we could eliminate the need to wait for the acknowledgement from the backup at each epoch boundary. Suppose all communication from the hypervisor at the primary were sent to two desinations — the hypervisor at the backup and to a *witness processor*. If the witness processor and its link fail independently from the primary and the backup then, assuming at most one failure, all messages are guaranteed to get to either the backup or the witness. This design avoids the need during epoch-boundary processing to wait for data to be received by the backup.

Hypervisor Overhead for Epoch

The witness-processor design eliminates the overhead terms h_{Back} and h_{Trans} from the hypervisor overhead at epoch-boundary processing. The requirement to communicate to both a backup and a witness, however, requires additional interraction with the network controller to set up the additional transmission. Figure 5.9 depicts the activity required at epoch-boundary processing. Note that this depiction assumes that there is no overlap between setting up the communication to the backup and setting up the communication to the witness, even though the same epoch marker packet is being sent to both. Thus, this is a pessimistic model of the revised epoch-boundary processing in the witness-processor design.

In this model, the epoch-boundary processing term is approximated by:

$$h_{\text{Epoch}} : h_{\text{Pro}} + (2 \cdot h_{\text{Set}}) + h_{\text{Epi}} \tag{5.60}$$

As in the CPU-intensive model, the hypervisor overhead for epoch-boundary processing is dependent on the number of epochs in the workload, given by $\frac{VI}{EL}$. The hypervisor overhead for epoch-boundary processing over the workload is approximated by the term $\frac{VI}{EL} \cdot h_{\text{Epoch}}$.

Hypervisor Overhead for Sim

The hypervisor overhead for simulating instructions is the same in the witness-processor design as in the CPU-intensive workload, or $n_{\text{Sim}} \cdot h_{\text{Sim}}$. (See Section 5.2.2, page 112.)

Hypervisor Overhead for Comm

In the witness-processor model, the overhead for processing hypervisor communication interrupts associated with the epoch marker messages is no longer hidden by the epoch-boundary processing. This is because the WaitStable call would no longer be executed and these interrupts would occur during virtual machine execution.

Recall that, assuming no communication errors, there are two Comm events for each message sent from the hypervisor at the primary to the hypervisor at the backup — one for the communcation interrupt upon completion of the transmit command by the network controller and another for the communication interrupt when the corresponding acknowledgement arrives from the backup. Likewise, there are two Comm events for each message sent from the hypervisor at the backup to the hypervisor at the primary — the first is the communication interrupt when the packet from the backup arrives at the primary and the second is the communication interrupt upon completion of the transmit of the acknowledgement from the primary to the backup.

The hypervisor at the primary sends two epoch markers per epoch boundary in the witness-processor design. Since there are $\frac{VI}{EL}$ epochs in the CPU-intensive workload and there are two Comm events for each such message, this contributes $4 \cdot \frac{VI}{EL} \cdot h_{\text{Comm}}$ to the hypervisor overhead for Comm.

Also, the backup sends messages to the primary for the purposes of flow control. This contribution to the hypervisor overhead for Comm is the same as the CPU-intensive workload, or $2 \cdot \frac{\frac{VI}{EL}}{WS} \cdot h_{\text{Comm}}$. (See Section 5.2.2, page 113.)

The Witness-Processor Model

Combining these overhead terms with the elapsed time for the control experiment, we arrive at our revised model for the elapsed time with the hypervisor present:

$$T_{\text{CPU}}(EL) : ET_C + n_{\text{Sim}} \cdot h_{\text{Sim}} + \frac{VI}{EL} \cdot h_{\text{Epoch}} + 4 \cdot \frac{VI}{EL} \cdot h_{\text{Comm}} + 2 \cdot \frac{\frac{VI}{EL}}{WS} \cdot h_{\text{Comm}} \quad (5.61)$$

Normalized performance is defined as before:

$$NP_{\text{CPU}}(EL) : \frac{T_{\text{CPU}}(EL)}{ET_C} \quad (5.62)$$

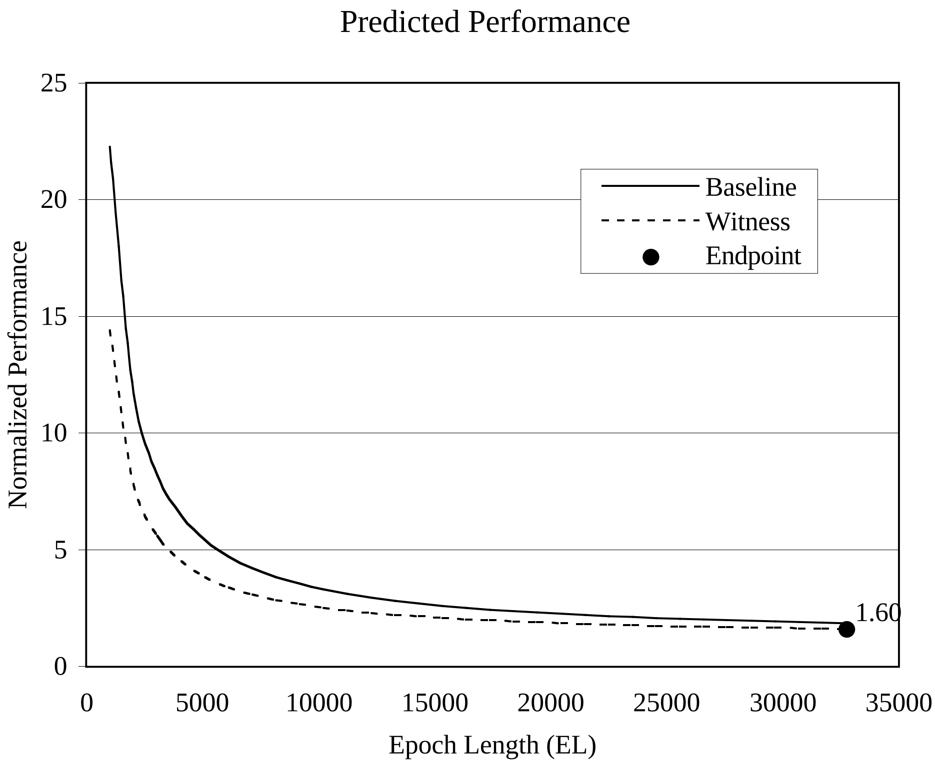


Figure 5.9: Predicted Performance of the Witness-Processor Model

In Figure 5.9 we graph equation (5.62) for epoch lengths in the range of 1024 virtual machine instructions through 32768 virtual machine instructions. Normalized performance for the original CPU-intensive workload model is also shown for comparison. At 4096 instructions per epoch, the original model has a predicted normalized performance of 6.45 while the witness-processor design has a predicted normalized performance of 4.48 — an improvement of 30.51%. At 32768 instructions per epoch, the normalized performance of the original model is predicted to be 1.84 and normalized performance of the witness-processor model is predicted to be 1.60, corresponding to a 13.35% improvement. At the upper bound for epoch length, the witness-processor model is predicted to be 1.22.

The drawback to the witness-processor design is the requirement of the additional hardware it entails. However, the capabilities required of the witness processor are very limited and it may be possible to use the existing hardware of an intelligent disk subsystem to act as a witness processor.

5.4.4 I/O Stability

Another improvement in performance results from appreciating that it is not strictly necessary for the hypervisor at the primary to await the acknowledgments by calling WaitStable during epoch-boundary processing. Such a design would defer the stability-wait required for correctness for as long as possible.

Suppose (i) a message sent by the hypervisor at the primary is not delivered to the hypervisor at the backup and (ii) the hypervisor at the primary passes through epoch-boundary processing without waiting. Thus, it becomes possible that an interrupt *Int* has been delivered to the virtual machine at the primary but, if there is a lost message, might never be delivered to the virtual machine at the backup. The computation of the virtual machine at the backup, 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 virtual machine at the backup — whatever they may be — are consistent with what could be observed were there a single processor. Thus, it suffices that the WaitStable be invoked prior to any output instruction by the virtual machine at the

Table 5.15: Normalized Performance of Original and Revised Protocol

Epoch Length	Workload					
	CPU-Intense		Write-Intense		Read-Intense	
	Old	New	Old	New	Old	New
1024	22.24	11.67	1.87	1.70	2.32	1.92
2048	11.83	4.49	1.71	1.66	2.10	1.76
4096	6.50	3.21	1.67	1.66	2.03	1.72
8192	3.83	2.20	1.64	1.64	1.98	1.70

primary, since output instructions are the only way in which the state of a virtual machine is revealed to the environment. Call this I/O stability.

The modifications to the protocol of Chapter 2 are straightforward. First, the WaitStable(Backup) is removed from the epoch-boundary processing depicted in Figure 2.4 (page 30). Second, before an output instruction is issued to the environment (see Figure 2.9, page 39), WaitStable(Backup) is called by the hypervisor at the primary.

We performed these modifications to the prototype and re-ran our experiments for the CPU-intesive workload, the read-intensive workload, and the write-intensive workload. Table 5.15 presents the normalized performance for each of these workloads at epoch lengths of 1024, 2048, 4096, and 8192 virtual machine instructions. The normalized performance given is the arithmetic mean of 20 measurement runs for each case. The column labelled "Old" refers to the original protocol and "New" refers to the I/O stability modified protocol.

As expected, the normalized performance improves significantly when the WaitStable is removed from the epoch-boundary processing. The effect is most pronounced in the CPU-intensive workload, because its normalized performance is most affected by the delay at epoch boundaries. In the Disk-I/O intensive workloads, some of the delay at an epoch boundary is simply displaced to the I/O operation in the next read or write cycle of the workload.

5.4.5 Variable Length Epochs

From the CPU-intensive workload, we saw that the predicted normalized performance of the system when the epoch length was long was very good, and thus long epochs are a desirable objective. From the Disk-I/O intensive workloads, we have the objective of keeping the interrupt delivery delay as short as possible. Combining these objectives leads naturally to the hybrid concept of *variable-length epochs.*

Consider a modified VFT protocol that defines the epoch length to be the epoch length upper bound EL^{max} unless some termination condition occurs, causing the epoch to be terminated early.[14] For simplicity, we only consider the termination condition of an interrupt destined for the virtual machine arriving at the primary.

CPU-Intensive Workload

Analyzing the performance impact of variable length epochs on the CPU-Intensive workload is straightforward. In the CPU-intensive workload, there are no interrupts to be delivered to the virtual machine. Thus, all epochs will reach the default of the epoch length upper bound. Normalized performance for variable length epochs for the CPU-intense workload, then, is assumed to be the same as the fixed length of EL^{max} virtual machine instructions per epoch. Table 5.16 summarizes the normalized performance for the three models presented in this chapter.

[14]For the purpose of this section, we are assuming $EL^{max} = 385000$ based on the analasis of Section 5.2.3 (page 116).

Table 5.16: Normalized Performance for the CPU Workload with Variable-Length Epochs

Model	Normalized Performance
Base CPU-Intensive Workload Model	1.24
155 Mbps ATM Communication Model	1.23
Witness-Processor Model	1.22

The results are very good with the largest component of overhead now attributable to the cost of virtualization (0.18 of the values listed in Table 5.16).

Disk-I/O Intensive Workloads

The effects of variable length epochs on the Disk-I/O intensive workloads are seen in (i) the write and read CPU processing intervals and (ii) the deferred interrupt delivery delay.

During each write (read) CPU processing interval, no interrupts arrive to be delivered to the virtual machine. The time spent in the write (read) CPU processing interval, then, is given by $t^{W}_{\mathrm{CPU}}(EL^{max})$ ($t^{R}_{\mathrm{CPU}}(EL^{max})$) from equation (5.45) (equation (5.46)).

In the varaible epoch length design, many of the same sequence of events must occur after arrival of an interrupt before it can be delivered to the virtual machine. These include the hardware save of context, the execution of the hypervisor interrupt handler, and epoch boundary processing. In the case of an I/O completion interrupt for a read operation, the data must still be transferred to the backup.

In the case of a minimal state transfer interrupt, the delay term $d_{\mathrm{Int}_{min}}$ (see equation (5.49)) is now approximated as follows:

$$d_{\mathrm{Int}_{min}}(EL) : \hat{h}_{\mathrm{Int}_{min}} + h_{\mathrm{Epoch}} \tag{5.63}$$

The delay for a disk read I/O completion interrupt $d_{\mathrm{Int}max}$ remains the same, since it represents the transfer time and stability wait for the data block transferred from the primary to the backup. This time is not affected by having variable length epochs. Its approximation is repeated here:

$$d_{\mathrm{Int}max} : \hat{h}_{\mathrm{Int}max} + \frac{4096}{2 \cdot IRate} + h_{\mathrm{Epoch}} + h_{\mathrm{Wait}} \tag{5.64}$$

Each write cycle still has three minimal state transfer interrupts, so the deferred interrupt delivery delay is approximated as follows:

$$d^W : 3 \cdot d_{\mathrm{Int}_{min}} \tag{5.65}$$

Similarly, each read cycle has two minimal state transfer interrupts and one read I/O completion interrupt, so the deferred interrupt delivery delay is approximated as follows:

$$d^R : 2 \cdot d_{\mathrm{Int}_{min}} + d_{\mathrm{Int}max} \tag{5.66}$$

Combining these with the approximations for the write cpu processing interval and the write transfer time yields the following approximation of normalized performance of the write-intensive workload under the variable epoch length design:

$$NP_W : \frac{n_W \cdot \left(t^W_{\mathrm{CPU}}(EL^{max}) + d^W + x^W\right)}{ET_C} \tag{5.67}$$

Table 5.17: Normalized Performance for the Disk-I/O Workloads with Variable-Length Epochs

Workload	Normalized Performance
Write-Intensive Workload	1.57
Read-Intensive Workload	1.92

Similary, the normalized performance of the read-intensive workload under the variable epoch length design is given as follows:

$$NP_R : \frac{n_R \cdot \left(t^R_{\mathrm{CPU}}(EL^{max}) + d^R + x^R\right)}{ET_C} \tag{5.68}$$

Substituting the measurements from Section 5.3.1 in the above approximations, we arrive at scalar values for the predicted normalized performance of the write- and read-intensive workloads. These are given in Table 5.17.

Chapter 6

Conclusion

6.1 Relation to Other Work

Virtual Fault-Tolerance distinguishes itself from other approaches to building fault-tolerant computing systems by:

(1) Selecting the hypervisor as the level for replica coordination.

(2) Being transparent relative to the architectural definition of the target processor.

Any fault-tolerant system must select a level for replica coordination and an interface to serve as the replication management interface[1]. Given this interface, the system must identify input instructions and input exceptions of that interface and ensure that (i) the same inputs are disseminated to all non-faulty replicas in the same order, and (ii) levels above the replication management interface behave deterministically in response to these inputs. Likewise, output instructions of the replication management interface must be identified and coordinated.

[1] Refer to Chapter 1 for definitions of the terminology used in this chapter.

If the replication management interface selected is an existing interface, then levels above the interface do not need to be modified. The solution is transparent relative to that interface.

6.1.1 Hardware-Level Replica Coordination

When replica coordination is performed at the hardware level, the replication management interface is at or below the architectural definition of the processor. This shares the same transparency properties as virtual fault-tolerance. It also shares the disadvantage (noted in Chapter 1) that failures in higher levels (i.e. the software) are not addressed.

Clearly, hardware replica coordination requires appropriate hardware. Thus, a design and/or reengineering cost is incurred for each new realization of the architectural definition. Because designing this hardware takes time, support for fault-tolerance necessarily lags behind the cost/performance curve. This is true both for new architectural definitions and for new realizations of existing architectures.

Despite the engineering and time-to-market costs, manufacturers continue to design and sell processors that implement replica coordination in hardware.

Stratus Computer Systems

The Stratus line of computer systems [Web92,Sie91] have the interface to the system bus as the replication management interface. All system components, including the system bus and the processors accessing the bus, are synchronized by a common (redundent) clocking scheme. This clocking scheme ensures that all system components run in lock-step.

In the Stratus system, four identical processors are grouped into a primary pair and a secondary pair. All four processors run from the same clock, and inputs from the system bus[2] are presented at the same time to all processors. Assuming determinsitic execution from the same inputs, all non-faulty processors will request the same output to the system bus at the same clock cycle. If a hardware comparison of the bus outputs from the primary processor pair reveals a difference, a failure is assumed to have occurred and the primary processor-pair is blocked from driving the bus, allowing the secondary processor-pair to take over. Since all processors use the same clock, interrupts are delivered to all processors at the same time.

Other components attached to the system bus, such as memory modules and I/O device controllers, are similarly replicated in the Stratus "pair and spare" architecture.

DEC VAXft 310

DEC's entry into fault-tolerant computing, the VAXft 310 [BB92], selects a slightly higher internal hardware interface as the replication management interface. In the VAXft 310, the main memory for a processor is included as part of each replica (called a *zone*). The memory controller for DMA operations to and from I/O devices becomes the replication management interface.

During system startup, each zone of a VAXft 310 operates independently to reach a point where the state of the processor/main memory subsystem is identical to that of the other zone. Within each zone, two CPUs execute instructions from a single common memory. A pair of CPUs allows CPU failures to be detected. ECC checks

[2]The system bus is also duplexed.

identify the failure of the memory in a zone. Once the zones have reached the same starting state, they run in lock-step by synchronizing their clock oscillators in a phase-lock loop.

DMA input and output are communicated between zones over a crosslink communication channel. DMA input and interrupts from I/O devices are presented at the same time and in the same order by virtue of the synchronized clock. DMA output from both zones is available unless a failure has been detected and the CPU/main memory subsystem of one zone is off-line. If a failure is detected during a DMA transfer, then the DMA is reissued from the non-faulty zone. The crosslink channel allows all I/O devices to be available to both zones.

Tandem Multi-processor

Tandem Computers has designed a fault-tolerant multi-processor system [CMJ90] that, like the Stratus design, uses the system bus interface as the replication management interface. The two designs differ in how the replicas synchronize for receiving input from the system bus and how outputs are selected by other components on the system bus (e.g. memory modules).

In the Tandem design, processors execute asynchronously, each using its own clock. A processor that is ready to perform its next memory read or write is stalled until all other processors reach the same point in their instruction stream. Likewise, interrupt delivery is stalled until the next of these synchronization points is reached.

Unlike the Stratus pair and spare design where, on output, pairs of replicas are used for fault detection and the spare pair is used in the event of a failure, the Tandem system allows outputs from all replicas to be received by the memory modules (over independent buses). Once the outputs are received, the memory modules use majority voting on the value and address to update memory.

6.1.2 Operating System Level Replica Coordination

There are two primary advantages to providing replica coordination within an operating system. First, doing so avoids the hardware design/reengineering costs, and thus no cost is incurred in tracking the price/performance for new hardware realizations of a given architecture. Second, by being at or below the operating system interface, transparency is achieved relative to the operating system interface, thereby supporting existing and new applications for a given operating system interface.

Mature operating systems are invariably complicated. Thus, identifying an appropriate replication management interface satisfying the requirements outlined at the beginning of this chapter can be quite difficult. In addition, this reengineering effort must be repeated for every operating system supported by a given architectural definition, and perhaps even each version change of the operating system.

NetWare Operating System

In the NetWare Operating System [MMP94,MPN92], an interface internal to the operating system has been selected as the replication management interface. The traditional operating system functionality for I/O is put into a separate lower level called the *I/O engine*. The rest of the operating system, called the *O.S. engine*, and the levels above become the replica[3] and the interface to the I/O engine is the replication management interface.

[3]The replica is called a *mirrored server engine*.

The NetWare approach structures each of the replicas as a state machine. The only input to a replica is through the I/O engine, and the replica must behave deterministically in response to these inputs. Ensuring determinism is simplified because NetWare does not allow preemption of applications. The NetWare operating system is structured in a primary/backup scheme, with the primary performing all interaction with the environment and communicating all inputs from the I/O engine to the backup.

A network routing protocol is employed to advertise reachability of the primary server engine. In the event of a failure, the backup uses the routing protocol to coerce client systems to switch connection endpoints.

TARGON/32

The TARGON/32 fault-tolerant computer system [GBBW86,BBG$^+$89] by Nixdorf Computer[4] transforms the Unix operating system interface as seen by programmers into a corresponding message-passing interface, where messages contain requests for operating system services. For example, a `read` system call to a disk file would be transformed into a `read` request message to a (fault-tolerant) file server process. It is this underlying message-passing interface that is the replication management interface in this scheme.

The Unix processes above the (transformed) system call interface form the replicas, structured in primary/backup pairs. The outputs from replicas and inputs to replicas are the request and reply messages that define the transformed system call

[4] Formerly known as the Auragen 4000 system of Auragen Systems Corporation.

interface. Processes are assumed to behave deterministically in response to all input messages.

Every message is sent along a custom-designed duplexed bus. This bus guarantees atomic 3-way delivery to the sender's backup, to the receiver, and to the receiver's backup. This guarantee means that the backup will process the same input messages in the same order as the primary.

In TARGON/32, the backup process does not perform any true computation. The backup simply supports enough operating system infrastructure to queue input messages received by its primary counterpart and to keep a count of messages output by its primary.[5] To enable instantiation of a backup upon a failure of its primary, TARGON/32 periodically performs a checkpoint of the state of the primary. Called a *sync* operation, the state of the primary is preserved through an innovative use of a virtual memory paging server (which must be a fault-tolerant process in its own right). The sync operation also allows the backup stub to discard messages in its input queue and to clear its output message count.

When a failure occurs, the backup image is initialized to the most recent checkpoint. Queued input messages may then be delivered to the recovering process. During this recovery period, any output message is discarded and the output message counter decremented until the output message counter is zero. At this point, the promoted backup may interact with the environment.

In TARGON/32, the only asynchronous events allowed are messages corresponding to Unix signals. The arrival of such a message at the primary causes processing

[5]The backup knows about all input/output messages to/from its primary through the atomic 3-way message delivery mentioned above.

to stop and an immediate sync operation between the primary and backup. This ensures that signals are handled at the same time in the instruction sequence at the primary and the backup without violating the Unix semantics of immediate delivery of signals.

Efficient server and I/O processes require the use of either kernel supported multithreading or asynchronous I/O operations. In TARGON/32, neither are supported. To overcome this limitation, Unisys has designed a system [Gle94] that is the intellectual successor to TARGON/32. The Unisys extension allows the general handling of asynchronous external events by recording the time of delivery of the asynchronous event relative to the instruction execution at the primary[6]. This recorded information is used to deliver the same asynchronous event to the backup at the same point in its instruction stream in the event of a failure.

6.1.3 Application Level Replica Coordination

Defining a replication management interface above the operating system may either be left to the application programmer, or an API (Application Programming Interface) may be defined that serves as the replication management interface. If the task is left to the application programmer, then similar problems must be solved by the programmers of every application. Moreover, all of these programmers must be aquainted with the nuances of replica coordination. If, on the other hand, the replication management interface uses a new or given API, application programmers are constrained to use the given interface and programming abstractions. These abstractions are not going to be suitable for all applications.

[6] In particular, a memory reference count register is used.

SIFT/August Systems

SIFT (Software Implemented Fault-Tolerance) [WLG+78] implements replica coordination in a real-time executive operating system and application software layer that executes above a custom hardware platform. The system was originally designed for fault-tolerance in flight control for fly-by-wire aircraft. August Systems, which grew out of the effort, extended the system to other process control environments. The system defines its own replication management interface; it is the inputs and outputs from cyclic real-time tasks.

SIFT uses (actually, introduced) the state machine approach to replica coordination (see page 6). Inputs to replicas are sensor values (with the sensors themselves replicated as necessary) or output from other fault-tolerant tasks. The majority value of each input is used by all (correct) replicas for its next real-time cycle. Ensuring identical ordering of inputs to all replicas is achieved by exploiting the well-defined input/output patterns of a process control system in conjunction with a fixed non-preemptive rate-monotonic scheduling of replica execution.

Tandem Non-Stop System

Tandem's Non-Stop series of computer systems [Bar81] uses a request/reply message passing scheme to serve as the replication management interface. The Tandem system employs a primary/backup approach with checkpoints of global data and message inputs under application control at the primary. In the event of a failure, the checkpoints of global data are used to put the backup in a reasonable state and checkpoints of message inputs are used to guarantee processing of the same inputs in the same order. Sequence numbers on request and reply messages are used so that, during failover, non-idempotent operations will not be executed twice. The system uses a custom duplexed bus for message communication.

Isis

The Isis system [BvR94,Bir93] is a toolkit that provides mechanisms and paradigms for applications programmers to use in building distributed and fault-tolerant systems. In particular, Isis provides a communication substrate for delivering messages to a group of processes in the same causally consistent order. In a typical usage scenario, this message-passing substrate is the replication management interface.

On the output side, a client can configure itself to receive the first output, the majority output, or all outputs, depending on its requirements and the failure model. By using these mechanism to coordinate message inputs and outputs, users can design systems that support either the state machine approach or a primary/backup approach to replica coordination.

6.1.4 Other Related Work

If processes in a distributed system behave deterministically in response to the messages received from other processes in the distributed system, then a process can be recovered after a failure by checkpointing the process state and recording all message interactions.

The PUBLISHING system [PP83] was designed to perform checkpointing and record message interactions in a transparent way by monitoring a broadcast communication medium (such as Ethernet). After a failure, processes are restarted at their last checkpoint and are delivered messages that were recorded since that checkpoint.

During process restart, duplicate output messages are suppressed by the system. The system relies on having a single point, the broadcast communication medium, from which to view all communication in that it allows complete knowledge of the set of input and output messages at a globally consistent state. From this knowledge, it is sufficient to suppress repeated output messages to bring the recovered process to a state that is consistent with the state of other processes in the system.

The general problem, in which there is not a single globally consistent view of the communication between processes, is significantly more difficult. Message logging techniques provide methods for ensuring that the state of a recovered process is consistent with the states of the other processes in the system in this general case. Work in [Alv96] formalizes the spectrum of message logging solutions and derives new optimal solutions.

Recent work by Elnozahy [Eln95] proposes using a recovery counter in a number of different domains as a means of tracking non-determinism in the execution of operating system kernels and in user programs. Some of the domains include performance monitoring, kernel debugging, and support for fault-tolerant system building.

6.2 Discussion

In this dissertation, we have presented a new approach to building a fault-tolerant system. The approach interposes a hypervisor between the operating system and the hardware levels of a computing system. Replica coordination is then implemented by this new level. Because the replication management interface is identical to the architectural definition, the approach appears transparent to the replicated virtual machines executing above the hypervisor.

The two major goals of the research were:

(1) To take the concepts of the approach and instantiate them in a prototype.

(2) To analyze the performance of this prototype and assess the practicality of the approach.

These goals have been achieved.

Building a prototype realization included:

- Extending the protocols to handle synchronous interaction with the environment.

- Redefining the definition of the architectural interface exported by the hypervisor to HP-UX to make it possible to virtualize. This had to be accomplished transparently to HP-UX.

- Adapting the I/O Model to the memory mapped I/O of the PA-RISC and the specifics of the SCSI interface.

The prototype has shown, by its existance, that the theory does map to practice. It has also shown that the recovery counter is useful.

Our performance analysis reveals that, for sufficiently long epochs (or for variable-length epochs), a CPU-intensive workload achieves a normalized performance of 1.25 and that a Disk I/O intensive workload achieves a normalized performance of under 2.0. This establishes that the approach is feasible.

There are a number of directions for further investigation. The current hypervisor implementation of replication management has a significant performance cost, but there are clear paths to reduce that cost. In particular, support for longer epochs and for variable-length epochs should be implemented in the hypervisor.

Reducing or eliminating the virtualization cost is another important direction, especially for the performance of the input/output workloads. From implementing our prototype, we learned exactly which instructions and exceptions must be intercepted by the replica coordination software in order to support the fault-tolerance protocol[7]. One could envision some type of macro expansion or replacement of just these instructions that avoids the privilege trap cost incurred for all privileged instructions that virtualization requires. Other approaches to be investigated include selecting a micro-kernel as the level for replica coordination and applying some of the same techniques as used in virtual fault-tolerance.

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 virtual machine without significantly delaying its execution.

[7] Distinguish these instructions from those that are intercepted solely to support virtualization.

Bibliography

[AAD+90] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. In *Proceedings of the Ninth ACM Symposium on Principles of Distributed Computing*, pages 3–19, Quebec City, Quebec, August 1990. ACM SIGPLAN/SIGOPS.

[AD76] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. *Proceedings of the Second International Conference on Software Engineering*, pages 627–644, October 1976.

[Alv96] Lorenzo Alvisi. *Understanding the Message Logging Paradigm for Masking Process Crashes.* Ph.D. dissertation, Cornell University, Department of Computer Science, January 1996.

[ANS89] ANSI. *SCSI-2 Protocol Specification*, 1989. Document Number: X3.131-198x (SCSI-2).

[Bai70] J. N. Bairstow. Many from one: The 'virtual machine' arrives. *Computer Decisions*, 2(1):28–31, January 1970.

[Bar81] J. F. Bartlett. A nonstop kernel. *Proceedings of the Eighth Symposium on Operating System Principles*, pages 22–29, December 1981.

[BB92] William Bruckert and Thomas Bissett. The VAXft 3000 architecture. In Daniel P. Siewiorek and Robert S. Swarz, editors, *Reliable Computer Systems: Design and Evaluation*, Burlington, MA, 1992. Digital Press.

[BBG+89] Anita Borg, Wolfgang Blau, Wolfgang Graetsch, Ferdinand Herrmann, and Wolfgang Oberle. Fault tolerance under unix. *ACM Transactions on Computer Systems*, 7(1):1–24, February 1989.

[Bir93] Kenneth P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12), December 1993.

[BvR94] Kenneth P. Birman and Robbert van Renesse. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, 1994.

[CMJ90] Richard W. Cutts, Jr., Nikhil A. Mehta, and Douglas E. Jewett. Multiple processor system having shared memory with private-write capability. U. S. Patent Number 4,965,717, October 1990. Assignee: Tandem Computers, Inc.

[Cri90] F. Cristian. Reaching agreement on processor group membership in synchronous distributed systems. Technical Report RJ 5964, IBM Almaden Research Center, August 1990. Revised from March, 1988.

[Eln95] E. N. Elnozahy. An efficient technique for tracking nondeterministic execution and its applications. Technical Report CMU–CS–95–157, Carnegie Mellon University School of Computer Science, May 1995.

[GBBW86] Sam Glazer, James Baumbach, Anita Borg, and Emanuel Wittels. Backup fault tolerant computer system. U. S. Patent Number 4,590,554, May 1986. Assignee: Parallel Computer Systems, Inc.

[Gle94] Barry J. Gleeson. Fault tolerant computer system with provision for handling external events. U. S. Patent Number 5,363,503, November 1994. Assignee: Unisys Corporation.

[Gol74] Robert P. Goldberg. Survey of virtual machine research. *Computer*, pages 34–45, June 1974.

[Hew90] Hewlett-Packard. *PA-RISC 1.1 Arcitecture and Instruction Set*, November 1990. Manual Part Number: 09740–90039.

[Int87] International Business Machines Corporation. *IBM System/370 : Principles of Operation*, 11 edition, 1987.

[Int92] Intel. *32–Bit Local Area Network (LAN) Component User's Manual*, 1992. Order Number: 296853–001.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[LS80] B.W. Lampson and H. E. Sturgis. Crash recovery in a distributed data storage system. Submitted to *Communication of the ACM*, 1980.

[LSP82] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[MMP94] Drew Major, Greg Minshall, and Kyle Powell. An overview of the NetWare operating system. In *Proceedings of the 1994 Winter USENIX Conference*, pages 355–372, January 1994.

[Mot86] Motorola Corporation. *M68000 8-/16-/32-bit microprocessors : Programmer's Reference Manual*, 1986.

[MPN92] Drew Major, Kyle Powell, and Dale Neibaur. Fault tolerant computer system. U. S. Patent Number 5,157,663, October 1992. Assignee: Novell, Inc.

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

[NCR92] NCR. *NCR 53C700 SCSI I/O Processor Programmer's Guide*, 1992.

[PP83] Michael L. Powell and David L. Presotto. Publishing: A reliable broadcast communication mechanism. *Proceedings of the Ninth Symposium on Operating System Principles*, pages 100–109, December 1983.

[Sch84] Fred B. Schneider. Byzantine generals in action: Implementing fail-stop processors. *ACM Transactions on Computer Systems*, 2(2):145–154, March 1984.

[Sie91] Daniel P. Siewiorek. Architecture of fault-tolerant computers: An historical perspective. *Proceedings of the IEEE*, 79(12):1710–1734, December 1991.

[SLBT96] Fred B. Schneider, Butler Lampson, Edward Balkovich, and David Thiel. Fault tolerant computer system with shadow virtual processor. U. S. Patent Number 5,488,716, January 1996. Assignee: Digital Equipment Corportation.

[Tan89] A. S. Tanenbaum. *Computer Networks.* Prentice Hall, Englewood Cliffs, N.J., 2 edition, 1989.

[Web92] Steven Webber. The Stratus case: The Stratus architecture. In Daniel P. Siewiorek and Robert S. Swarz, editors, *Reliable Computer Systems: Design and Evaluation*, pages 648–670, Burlington, MA, 1992. Digital Press.

[WLG+78] John H. Wensley, Leslie Lamport, Jack Goldberg, Milton W. Green, Karl N. Levitt, P. M. Melliar-Smith, Robert E. Shostak, and Charles B. Weinstock. Sift: Design and analysis of a fault-tolerant computer for aircraft control. *Proceedings of the IEEE*, 66(11):1240–55, October 1978.