1995

A Programming Model For Optimism

Crispin Cowan

A Programming Model for Optimism

by

Crispin Cowan

Department of Computer Science

Submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

Faculty of Graduate Studies
The University of Western Ontario
London, Ontario
January 1995

ISBN 0-315-99251-4

ABSTRACT

Technological advances are increasing the throughput of most aspects of computing systems. However, latency is being held back by the speed of light, particularly in distributed systems. Optimistic algorithms that "guess" the results of operations and proceed in parallel with confirmation of the guess are an effective way to hide the latency of slow operations with predictable outcomes. Optimism is used in concurrency control and distributed simulation systems, but is not generally used. Optimistic algorithms are difficult to write because of the necessity for checkpointing, rollback, and dependency tracking. This thesis presents a set of primitives called HOPE that simplifies the expression of optimistic algorithms. HOPE provides primitives to specify, confirm, and deny any optimistic assumption about a future state of the program. Dependence on optimistic assumptions is automatically tracked so that data can be exchanged without regard to the speculative status of the tasks involved.

We begin by defining the principles of optimism, and present some example applications of optimism. The HOPE programming model is presented, using both an informal description of its primitives, and a definition of its formal semantics. We then describe the abstract algorithms for implementing HOPE in a distributed environment, as well as the prototype HOPE system that embeds the HOPE primitives in C using the PVM message passing library. Finally, we introduce the performance characteristics of the prototype HOPE system, and describe future research directions.

ACKNOWLEDGEMENTS

HOPE was inspired by optimism studies at the IBM T.J. Watson Research Center. The initial inspiration for HOPE came from discussions with Rob Strom. Thanks go to Andy Lowry for our many constructive debates about optimism, as well as Jim Russell and Ajei Gopal for their helpful comments and suggestions. Finally, much thanks is owed to my advisors Hanan Lutfiyya, Mike Bauer, and Mike Bennett, for their time and patience in bringing it all together.

To all my friends and family, for never once teasing me about how long it took to get here.

TABLE OF CONTENTS

[1] Faint HOPE.

[2] A reason for HOPE.

[3] A shred of HOPE.

[4]The meaning of HOPE

[5]A basis for HOPE.

[6]Be careful what you HOPE for, you might get it.

[7].. measure of HOPE.

[8] HOPE for the future.

[9] Abandon all HOPE ye who enter here...

[10] Array of HOPE

LIST OF FIGURES

LIST OF TABLES

A PROGRAMMING MODEL FOR OPTIMISM

Chapter 1

Introduction[1]

"Why is this taking so long?" is a common lament among computer users. While computers are getting faster on a daily basis, there are some limitations to performance that are difficult to overcome, such as the speed of light over a geographic distance.

Distributed systems provide for the execution of programs on multiple computers that are physically distant from one another, inducing an inevitable delay in communicating data between these computers. As long as the data being communicated is strictly of an advisory nature, this delay does not present a significant problem. However, if the communications requires a transaction of some sort, e.g. the sender needs to know the results of processing the message, then the delay becomes problematic.

In the past, with slow communications networks such as 3 Mbit Ethernet, Token Ring, etc., a major obstacle to communicating with a remote computer was the time it took to transmit the data. The bandwidth was inadequate to support the timely transmission of even moderately sized messages.

However, with the advent of 100 Mb/s optical networks, the time to transmit moderate volumes of data has become a nominal component of the total latency of communicating with a remote computer. Software overhead and the speed of light are now the major contributors to the latency of sending a message to a remote computer and receiving a reply. Software overhead may eventually decrease, but the speed of light is not amenable

[1]Faint HOPE.

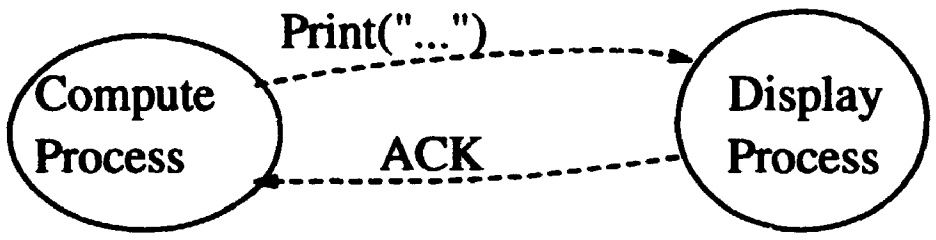

Figure 1.1: Example Distributed Program

to change. The time required to send a photon from New York to Los Angeles and back again is 30 milliseconds. While a transcontinental 100 Mb/s fiber optic channel is capable of sending 100 byte packets 100,000 times per second, it is only capable of sending that 100 byte packet 30 times per second if each transmission waits for a response.

Consider the distributed program in Figure 1.1, in which a compute process is presenting its output via a remotely located display process. To verify that the Print operation was successful, the Application Process must wait for the ACK message to return from the Display Process. If the two processes are located on nearby machines, then the delay may be negligible. However, if they are physically distant, then the limits of the speed of light impose a significant delay.

This delay is only one of many kinds of obstacles to rapid computation that we will characterize as **latency**:

Definition 1.1 Latency *is the time from when one operation is started to when the next operation may be started.*

Often, latency can be reduced by executing operations in parallel. Executing operations in parallel may reduce the overall **response time** of an application:

Definition 1.2 Response time *is the time from the submission of a user request until the first response is produced [48, pages 115-6].*

Naturally, the response time is of interest only if there is a user waiting for the response.

There are limits to the effectiveness of reducing response time through parallelism. Amdahl's Law [2] states that the speedup due to parallelism is as follows:

$$\text{Speedup} = \frac{1}{(1-\text{fraction parallelized}) + \frac{\text{fraction parallelized}}{\text{speedup of parallelism}}}$$

This equation shows that the speedup due to parallelism is limited by the fraction

of a program that can actually be parallelized. In particular, if an operation uses data provided by a previous operation, then it cannot start until the operation providing the data completes. Thus the inter-dependence of operations imposes latency on the operations, and thus imposes an upper bound on the amount of speedup that can be achieved.

Many techniques have been developed to hide or work around latency. The notion of multiprogramming was first invented to hide the latency of slow I/O devices to get more throughput from expensive CPU's [48, page 19]. Multiprogramming is a traditional way to hide latency from a computer by using parallelism among applications. Parallel programming is a more aggressive way to overcome latency by executing components of an application in parallel. Optimism is a relatively new way to further avoid latency by creating new opportunities for parallelism in an application.

Optimism increases concurrency in an application, allowing for more parallelism, which may further reduce the application's latency. A program can avoid a delay by making an optimistic assumption about its future state, and verifying the assumption in parallel with computations based on the optimistic assumption. By executing the verification of the assumption in parallel with computations that depend on the assumption, the latency of the verification procedure is masked, effectively decreasing the sequential component of Amdahl's equation.

Optimism introduces concurrency by making assumptions about what some *other* component of the system will do. Thus optimism is particularly applicable to masking the latency of components of a system that have predictable behavior. An effective optimistic assumption is one that masks a large latency relative to the likelihood of being incorrect.

Optimistic techniques can be particularly beneficial to parallel and distributed systems. Inter-node communications latency is critical to performance because it limits the degree to which an application can be parallelized. Optimism can help mask inter-node communications latency by making optimistic assumptions about the behavior of remote nodes.

As we will describe in Chapter 3, optimism can be used to transform a sequence of synchronous send-reply pairs into a stream of asynchronous messages. As networks get faster, the number of messages that can be sent while waiting for a reply increases. Similarly, as processors get faster, the amount of computing that can be done while waiting for a reply increases. Thus, advancing technology is going to increase the imbalance between latency and throughput, making the use of optimism even more beneficial.

1.1 Principles of Optimism

For our purposes, a program is composed of concurrent processes that execute operations that cause events that change the state of a process. A **computation** is a sequence of states that have occurred in the execution of a process. **Rollback** is returning a specific process to a previous state in its computation, and discarding the computation subsequent to that state. An **optimistic assumption** is an assertion about a future state that has yet to be verified. An **optimistic computation** (or **speculative computation**) is a computation that proceeds based on an optimistic assumption, and is said to be **dependent** on that assumption. If the optimistic assumption is found to be true, then the optimistic computation is retained, otherwise it is rolled back.

Observation 1 Optimistic computations *are equivalent to computations that use* **rollback.**

Justification 1 *If a computation is subject to rollback, then starting the computation is optimistic because it is based on the assumption that the computation will be retained and not rolled back. Conversely, if a computation proceeds based on any optimistic assumption at all, that assumption may prove to be incorrect, in which case all computations executed based on that assumption must be rolled back to correct for the incorrect assumption. Thus all optimistic computations are subject to rollback.* □

Observation 2 Optimistic assumptions *always increase available concurrency.*

Justification 2 *An optimistic assumption always relates to a state property that is not yet known; otherwise it would not be optimistic. Without the optimistic assumption, the assumption verification and dependent computation(s) would have to be executed sequentially. Optimistically making the assumption allows assumption verification and dependent computation to be executed concurrently. Thus optimism always increases available concurrency.* □

Figure 1.2 illustrates how optimism can increase concurrency and reduce latency. Sometimes the newly introduced concurrency is quite obvious, such as an optimistic assumption that a concurrency lock will be granted. Sometimes it is quite subtle, such as in fault tolerance applications, where the concurrency introduced is between the volatile and stable-storage components of the system.

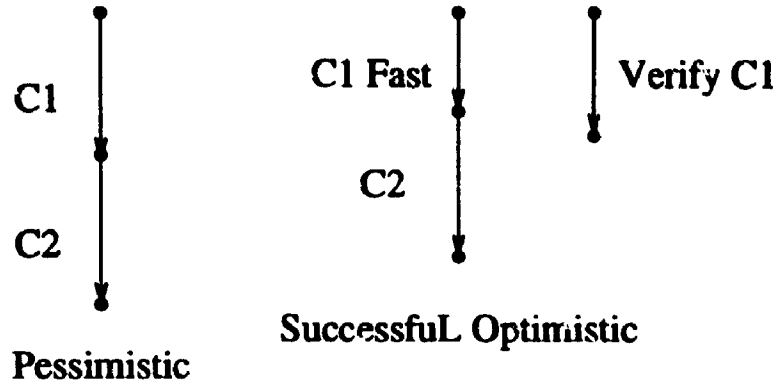

Figure 1.2: Increasing Concurrency Through Optimism

It should be noted that while optimism always increases concurrency, it does not always improve performance. Improved performance depends on the probability of the assumption being correct, the costs of rolling back computations, and the fixed costs of making rollback of computations possible [59]. However, it should also be noted that whenever rollback occurs, other rollback-free algorithms would require blocking for an amount of time equal to that spent on wasted computation [31].

1.2 HOPE: Hopefully Optimistic Programming Environment

Optimistic algorithms have been used for fault tolerance [57, 33], replication [15, 24, 35, 49, 64], concurrency control [29, 32, 36, 68], and discrete event simulation [6, 22, 31, 65]. However, the notion of optimism has not been generally exploited because optimistic algorithms are difficult to write. Obstacles to optimism include the following:

Checkpoint & Rollback	This is difficult and non-portable.
Dependency Tracking	**ALL** actions that depend on an assumption must be rolled back if the assumption is wrong, including remote processes that have been sent messages.
Transitivity	What if an optimistic computation is used to decide whether another assumption was correct?

This thesis presents a set of primitives that simplify the expression of optimistic algorithms. There are primitives to specify, confirm, and deny any optimistic assumption. The primitives automatically track dependencies so that tasks may be oblivious to the potentially speculative nature of computations in which they are involved. When combined

with a message passing facility, these primitives form a programming mode we call **HOPE** (Hopefully Optimistic Programming Environment).

Because optimistic techniques are difficult without assistance, they tend to appear only in specialized products such as database managers. By simplifying the expression of optimistic algorithms, HOPE enables the use of optimism by less sophisticated programmers for applications. Similarly, by lifting the burden for checkpointing and dependency tracking, HOPE enables easier investigation into new optimistic algorithms. Chapter 7 demonstrates the generality of the HOPE model of optimism by showing how various different optimistic systems and algorithms can be expressed with HOPE.

This thesis begins by reviewing related work in optimism in Chapter 2. Chapter 3 introduces the HOPE primitives by providing examples that use HOPE. Chapter 4 presents the formal semantics of the HOPE primitives. Chapter 5 describes the algorithms for implementing a working HOPE system, and Chapter 6 describes the prototype HOPE system that has actually been constructed. Chapter 7 describes the applicability of HOPE to some problem domains by describing how some of the optimistic algorithms and systems in Chapter 2 can be expressed with HOPE, including optimistic replication as discussed in [15]. Chapter 7 also assesses the effectiveness of HOPE by comparing the source code size and runtime characteristics of some example programs written for the HOPE prototype to their pessimistic counterparts. Finally, Chapter 8 presents conclusions and future research opportunities.

Chapter 2

Related Work in Optimism[1]

This chapter surveys related work in optimism. Such work can be subdivided into optimistic algorithms and optimistic systems. Optimistic algorithms are algorithms that use optimism to meet some specific objective, such as concurrency control or replication control [17]. Optimistic systems are programming environments that provide some facility for programmers to make optimistic assumptions. Section 2.1 surveys applications of optimism, Section 2.2 describes previous optimistic systems, and Section 2.3 summarizes the related work.

2.1 Applications of Optimism

This section surveys some applications of optimism, such as concurrency control, replication, replicated concurrency control, and fault tolerance. Since many optimistic concurrency control algorithms have been published [29, 32, 68], we do not attempt to discuss them all, and instead present only a sample: Kung & Robinson's optimistic concurrency control, and Triantafillou and Taylor's optimistic replicated concurrency control.

2.1.1 Kung and Robinson's Optimistic Concurrency Control

Concurrency control is the problem of scheduling concurrent operations in such a way that they *appear* to have been executed in a single sequence with respect to the data objects that they share. Such a schedule is called **serializable** [47]. However, it is possible to schedule such operations concurrently, so that non-interfering operations are executed in parallel. A conservative approach to scheduling would be to look at each operation, and try to find a schedule that will not force any of the operations to roll back. However, there are

[1] A reason for HOPE.

no deadlock-free conservative protocols that always provide high concurrency [36]. Thus the problem of finding a schedule forms a serial bottleneck to the concurrent execution of operations on shared data.

Kung and Robinson's Optimistic Concurrency Control [36] is an optimistic algorithm for scheduling concurrent transactions. They make the optimistic assumption that, for sufficiently large sets of data objects, operations usually do not conflict, and so it is possible to optimistically schedule all transactions to be executed concurrently as they arrive for processing, and detect and correct concurrency errors post hoc. If the optimistic assumption is correct often enough, this optimistic approach will achieve higher concurrency than a conservative schedule.

Kung and Robinson's system provides for conventional transaction processing. The mechanism used is to optimistically precompute entire transactions using copies of the data affected instead of the actual data, and then check for conflicts, writing back results only if no conflicts occur.

Kung's implementation technique is to replace read and write calls with calls to functions that create and manage local copies of any data that the transaction may want to write. Transaction commitment first performs concurrency control (ensuring that this transaction can be serialized) and then writes back the modified copies in place of the actual data.

The optimistic assumption is that a concurrency control conflict will not occur, and thus it is worthwhile to execute all of the computations of the transaction before checking that a conflict exists. The assumption verification procedure is similar to that used in a pessimistic transaction system, but it is executed after the transaction, not before. The contingency plan is that if the assumption proves false then rollback the transaction and try it again.

2.1.2 Triantafillou's Optimistic Data Replication

One approach to reducing latency in a distributed system is to replicate components of the system. Replicating data objects increases the likelihood that the object will be local and not on a remote node, reduces the mean distance to the object if it is on a remote node, and reduces the load on the nodes providing access to the object by distributing the load across multiple nodes. However, such replication of data objects only reduces latency for *read* operations, those that do not alter the object. Write operations must engage in a protocol with all of the replicas of an object so that the object stays **consistent**, i.e., appears the

same regardless of which replica is accessed. Thus write operations in a replicated system actually suffer greater latency than in non-replicated systems.

Triantafillou and Taylor [64] present a consistency mechanism for replicated data in transaction-based distributed systems. Like Kung & Robinson, Triantafillou makes the optimistic assumption that write operations do not conflict, and executes the replica consistency check in parallel with the body of the transaction.

Triantafillou's "Location-based Paradigm" features an enhanced location server, which instead of merely providing a reference to some location that holds a replica of a given object, provides a list of locations that the server believes to be up-to-date.

Two optimistic assumptions are made by this algorithm. The first assumption is that the site the client selects from the list provided by the location service actually has up-to-date data. This site is called the leader for the accessed object, not to be confused with the coordinator of the two-phase commit protocol, which is the client itself. Transactions optimistically proceed using data found at the leader site until the assumption can be confirmed or denied. The second assumption is that a lock acquired only at the leader site will be granted by a quorum of other replica sites. Thus, Triantafillou is using the conventional techniques of a location service and quorum consensus [23], but is enhancing them by using optimism to execute these functions asynchronously.

The location-based algorithm proceeds as follows: clients processing transactions begin by asking the location service for a list of up-to-date replicas of each object desired (the granularity of objects is not discussed). The client then selects one of the replicas from the list to be the "leader," using whatever desirability criteria is appropriate. From this point on, the client treats the leader as if it has the sole copy of this object. The client optimistically assumes that the leader contains an up-to-date copy of the object, and that no conflict with other transactions exists so that the quorum consensus locks will be granted by the other replicas. If either assumption proves to be false, the transaction is aborted and restarted.

Triantafillou places the transactions in an arbitrary total ordering. When conflict is discovered, the transaction to be aborted is selected by using this ordering. Starvation is avoided by retaining the original transaction number when a transaction is aborted and restarted.

When a transaction completes processing, it enters a conventional two-phase commit protocol with the set of leaders it has selected. When a leader receives a prepare message,

it decides whether to agree by determining if a quorum of the replicas have granted the lock required. If any of the leaders refuse to prepare, the transaction is aborted. The leader also checks that its own version number is the same or greater than the version number on all of the replicas that it has contacted. If any of the replicas has a higher version number, then the optimistic assumption that this site has an up-to-date copy of the object is false, and the transaction is aborted. Thus, the leader checks the optimistic assumptions that an individual data item is up-to-date and that the locks can be acquired, and the client checks the optimistic assumption that the transaction will complete. If a transaction fails, the contingency plan is to abort and restart the transaction.

When a transaction successfully completes, it momentarily has a precise record of which replicas of the objects that it wrote to are up-to-date. Unless a failure or recovery has occurred, this list should not be different from that provided by the location service. In the rare event that a failure or recovery has occurred, the client issues a write to the logically centralized location service updating the list of up-to-date replicas.

2.1.3 Goldberg's Optimistic Process Replication

In [24] Goldberg describes several mechanisms to transparently support process replication. Replicated processes appear to function identically to a single, un-replicated process, but provide lower latency for read and non-conflicting write operations. Goldberg presents one mechanism layered on top of Jefferson's Virtual Time [31], a second mechanism integrated with Virtual Time, and a third mechanism based loosely on the dependency tracking mechanisms of Strom and Yemini's Optimistic Recovery [57].

Goldberg assumes that processes communicate only with messages. He further partitions messages into **write** messages (those that change the recipient's state) and **read** messages (those that do not)[2].

Since read messages do not alter a process's state, they can be delivered to the most convenient replica process available for processing. Write messages are delivered to the primary replica where they are re-broadcast to all of the other replicas.[3] A consistency protocol is used to ensure that messages are processed in the same order at all replicas in the system.

---

[2]Goldberg also provides a virtual memory scheme to detect read and write messages when such distinctions are not available from the programmer, but we do not concern ourselves with this aspect.

[3]Goldberg also provides a distributed mechanism, but we omit it for simplicity of explanation.

The optimistic assumption contained in Goldberg's algorithm is that a write message delivered to the primary replica can be distributed to the secondary replicas without causing consistency errors. Since write messages are serialized at the primary replica, WW conflicts are not possible. Since Goldberg assumes that these messages are asynchronous (because synchronous receipt of a read message would alter the state of the recipient) and that messages can pass one another, RW conflicts are also not possible. Thus the only remaining conflict is that a cyclic dependency may form. A cyclic dependency is formed when two different clients perceive that two or more write operations took place in *different* orders. This occurs when a read message is processed "too early" by a replica. The algorithm responds by rolling back the write message that is the antecedent of one of the offending reads, thus breaking the dependency loop. Dependency tracking and message logging mechanisms are provided to facilitate detecting and correcting cyclic dependencies in this way.

2.1.4 Strom and Yemini's Optimistic Recovery

An important part of fault tolerant computing systems is that they not become inconsistent in the presence of hardware failures. A conservative approach to maintaining consistency is to always wait until the results of an operation have been written out to stable storage before proceeding to the next operation. Thus, when the machine is restored, computation can proceed from where it left off. Fault-tolerant computing becomes much more important in distributed systems because there are many more components that are susceptible to failure. However, waiting for writes to stable storage introduces large amounts of latency because I/O devices are very slow parts of a computing system. Better performance is obtained by executing I/O operations concurrent with subsequent computations.

Strom and Yemini [57] present a system for high performance fault-tolerant distributed computing based on the optimistic notion that computers mostly do not fail. They use a detailed dependency tracking mechanism to allow a distributed system to restore itself to consistency in the event of failure, yet not wait for operations results to be written to stable storage before proceeding. Any node's state that depends on a previous state that was lost due to failure is called an **orphan**, and is discarded and re-computed. The semantics presented is that of reliable, persistent processes that run transparently on unreliable computing nodes.

In [57] Strom and Yemini present a system for high-performance fault-tolerant distributed computing based on the optimistic notion that computers mostly do not fail. The

Optimistic Construct	Meaning
`if wouldbe(x <= limit) then`	`{save value of x}`
`  x := x * 2`	`tempx := x;`
	`{execute body of then-clause}`
	`x := x * 2;`
	`{in case subjunctive fails,`
	`restore original value of x}`
	`if not (x <= limit) then`
	`  x := tempx;`

Figure 2.1: Strothotte's Subjunctive Construct

semantics presented is that of reliable, persistent processes that run transparently on unreliable computing nodes.

2.2 Previous Optimistic Systems

As Section 2.1 demonstrates, optimism is a powerful technique for masking latency by introducing additional concurrency. However, optimism is not widely used, except in specialized products such as database systems, because of the difficulty of rolling back operations, and in tracking all of the dependencies of an optimistic assumption (subsequent actions taken that presume the optimistic assumption was correct). Thus, unless one has a pressing need to escape latency restrictions, optimism is not used.

To overcome these obstacles and make optimism more attractive to developers and researchers, the rollback and bookkeeping tasks of optimism must be automated. In this section, we review several previous efforts to automate parts of optimism.

2.2.1 Strothotte's Temporal Language Constructs

In [60, 61] Strothotte describes three classes of programming language constructs for expressing temporal notions in a program: those for the past tense, the future tense, and the subjunctive tense. The subjunctive tense construct, being the only one providing a facility for speculative computation and rollback, is the only component of Strothotte's language that is relevant to optimism. Figure 2.1 illustrates Strothotte's subjunctive construct. The

span of a speculative computation in Strothotte's language is limited to the scope of an `if` or `while` statement, and thus the range of optimistic computation is statically bounded. The speculative predicate (the `wouldbe` argument) is evaluated at the end of the speculative `if` or `while` block, and the computation is either retained or rolled back at that point.

2.2.2 Bubenik's Optimistic Computation

In [9, 10], Bubenik et al. describe a formalism for automatically transforming pessimistic algorithms into optimistic algorithms based on concepts in [57, 59]. Bubenik translates the algorithm into a *program dependence graph*, and then performs optimistic transformations on the graph. Bubenik also constructed an optimistic run-time environment in which to exercise these transformations.

Bubenik's optimistic facility is somewhat more dynamic than Strothotte's, but less dynamic than HOPE. Bubenik provides an operating systems facility to execute **encapsulations** whose outputs are concealed until the computation is **mandated**. As in Strothotte, the scope of an optimistically executed encapsulation is statically bounded. Unlike Strothotte, an encapsulation can be mandated or aborted asynchronously with its execution, instead of only at the precise end of the speculative computation.

Bubenik's system does not do dependency tracking. If some other encapsulation `y` wishes to use the interim results of a given encapsulation `x` before `x` has been mandated, then encapsulation `x` must be specified at the time encapsulation `y` is started. Since all uses of speculative results are explicit, dependency tracking is not needed.

Confining speculative results to within an encapsulation in this way greatly simplifies the problems of implementing support for optimism. The range of speculative computation is again statically bounded, and transitive dependency tracking is not necessary. However, this simplification also limits generality. If more time is available than expected for an optimistic computation, it cannot be utilized because the encapsulation is of fixed size. Non-optimistic segments of a program cannot benefit from the optimism of an encapsulation if it does not propagate its results until it is mandated. Bubenik and Zwaenepoel argue their granularity is deliberately large for efficiency reasons [11].

2.2.3 Time Warp

Discrete event simulation is a computing application that simulates the behavior of complex systems for the purposes of design and modeling. The number of "events" being simulated

in such a system is often very large, and so such simulations may run for a long time. Since most events are independent of one another, discrete event simulation is amenable to speedup through parallelism. However, when simulating a given event, it is often not decidable at the time whether the event in question is independent of other events in the system: is there a factor that this event has not yet learned of? Conservatively waiting for such confirmation imposes a limitation on the available concurrency in a simulation. Optimistically assuming that such a contributing factor does not exist increases available parallelism.

Jefferson's Virtual Time [31] is an algorithm for distributed process synchronization that has been widely studied [1, 26, 32, 42, 55]. Time Warp is a discrete event simulation system based on the Virtual Time concept that provides the illusion of a globally synchronized clock that can be used to preserve a total ordering across the system as defined by Lamport [38], even though processes actually are being executed out of order. Thus the semantics that it guarantees are those of a sequentially executed series of computations.

Time Warp optimistically assumes that messages are processed in receive timestamp order, i.e. if the local clock is advanced, no message will ever arrive with a time stamp preceding the new clock setting. Thus every advance of the local copy of the virtual time is a speculative action that may have to be rolled back. If a message arrives with a time stamp that precedes the current local time, then the process is rolled back to the earliest time stamp in its message queue and re-started.

Unlike the previous optimistic programming systems mentioned, the amount of speculative computation that Time Warp can execute is not statically bounded. However, the only optimistic assumptions that can be made in Time Warp are with respect to message arrival order. Any other kind of optimistic assumption would have to be re-cast in terms of assumptions about the order in which messages arrive.

2.3 Summary

Section 2.1 demonstrated the utility of optimism in various application areas. Section 2.2 surveyed systems that allow for the expression of optimistic algorithms. The section is quite brief, yet the survey is fairly complete: systems that support the expression of optimistic algorithms are rare. These systems also are all limited in some way:

Strothotte The span of a speculative computation in Strothotte's language is limited to the scope of an `if` or `while` statement, and thus the range of optimistic computation is statically bounded.

Bubenik
- The results of a speculative computation are not available outside the computation unless explicitly requested.
- The range of speculative computation is statically bounded.

Jefferson While optimism is not limited in scope, the only optimistic assumption that can be made is with respect to message arrival order.

The following chapters will present our programming model for optimism. In this model, any optimistic assumption can be made, any computation can make use of the results of a speculative computation, the span of optimistic computation is not bounded, and any combination of speculative and non-speculative computations can be used to determine the validity of an optimistic assumption.

Chapter 3

HOPE: Informal Description[1]

Previous optimistic programming systems are all restricted in some way. Here we introduce our general programming model for expressing optimism: HOPE (Hopefully Optimistic Programming Environment). HOPE is general in that:

- HOPE allows a dynamic amount of speculative computation.
- Optimistic assumptions can be verified asynchronously and dynamically by any process in the system.
- Optimistic assumptions can be transitively verified, in that speculative computations can make assertions about other optimistic assumptions, i.e., HOPE processes can use the speculative output of other speculative processes before they have been verified.

HOPE consists of one data type and four primitives, as follows:

AID x	x is an **assumption identifier** or **AID**, used to identify particular optimistic assumptions.
guess(x)	The executing process makes an assumption identified by x. **guess** speculatively returns **True** immediately, and returns **False** if rolled back.
affirm(x)	The executing process asserts that the optimistic assumption identified by x is confirmed as true.
deny(x)	The executing process asserts that the optimistic assumption identified by x is found to be false.
free_of(x)	The executing process asserts that the current computation is not, and never will be, dependent on the assumption identified by x.

[1] A shred of HOPE.

The AID is a significant novel feature of HOPE. An AID is a reference to an optimistic assumption. Using the primitives described here, dependence, precedence, and confirmation of an assumption can all be handled separately.

guess(x) appears to the programmer to be a boolean function that returns **True** if the assumption identified by x is correct, and returns **False** if x's assumption is found to be incorrect. The AID is an abstraction that represents the optimistic assumption, and can be used to **affirm** or **deny** the optimistic computation.

guess(x) will return **True** immediately, regardless of the status of the assumption. Speculative computation begins at this point, with the process "dependent" on x. Based on the returned value, the process then proceeds based on the optimistic assumption being true. If x's assumption is later discovered to be false, the process is rolled back to where it called **guess**, and **False** is returned instead of **True**. Having been informed through the return code that the assumption was incorrect, the process can then take the necessary steps to deal with the flawed assumption.

Idiomatically, **guess(x)** is embedded in an **if** statement. The "true" branch of the **if** statement contains the optimistic algorithm, and the "false" branch of the **if** statement contains the pessimistic algorithm. **aid_init(x)** is used to initialize x ahead of time, so that a checking mechanism can be set up to verify x's assumption.[2]

affirm(x) asserts that the assumption associated with the AID x is correct. Similarly, **deny(x)** asserts that the assumption associated with x is incorrect. If **affirm(x)** is executed anywhere in the system, all the speculative computations executed from **guess(x)** onward are retained. If **deny(x)** is executed anywhere in the system, the computations from **guess(x)** onward, including any causal descendants in other processes, are discarded and rolled back, and execution re-starts from **guess(x)** with a return code of **False** instead of **True**.

There is no restriction on how much computation can be executed before an optimistic assumption is confirmed. There is no restriction on which process in the program may confirm an optimistic assumption. Only one **affirm** or **deny** primitive may be applied to a given assumption identifier, because multiple **affirm** or **deny** primitives are redundant, and conflicting **affirm** and **deny** primitives have no meaning. Speculative processes can execute **affirm** and **deny** primitives, and the system will transitively apply the assertions,

[2]Although **guess** is applicable in modeling non-deterministic algorithms, it is a subjunctive statement, not a non-deterministic statement.

i.e., if a speculative process is made definite, then all **affirm** primitives it has executed will have the same effect as definite **affirm** primitives.

In summary, a **guess(x)** eventually either results in the execution of an **affirm(x)** and **guess** returns **True**, or **deny(x)** and **guess** returns **False**.

In addition to explicit **guess** primitives, processes can also become dependent on AIDs by exchanging messages. When a speculative process sends a message, the message is "tagged" with the set of AIDs that the sender currently depends on. When the message is received, the receiver implicitly applies a **guess** primitive to each of the AIDs in the message's tag.

Execution of the **free_of(x)** statement means that the executing task has no causal dependencies on any event in the speculative executions dependent on x. If any such dependency is ever detected, then x is denied. Asserting **free_of(x)** ensures an execution in which the asserting process is causally free of all events dependent on the AID x.

Section 3.1 illustrates the meaning of the HOPE primitives by presenting some example HOPE programs. Section 3.2 describes necessary and desirable language features for embedding HOPE in other languages.

3.1 Examples

This section illustrates the meaning of the HOPE primitives by presenting two optimistic program transformations [59]. An optimistic transformation is a transformation of a pessimistic (conventional) program into an equivalent program that uses optimism.

Two programs are presented in each example: an optimistic program, and a pessimistic (conventional) program. Since both examples have identical results, the pessimistic example illustrates the meaning of the HOPE primitives in the optimistic example. The examples are written in pseudo-code extended with RPC calls and the HOPE primitives.

3.1.1 Optimistic Numerical Integration

Figure 3.1 shows a simple program for computing a numerical integral using an iterative method. Figure 3.2 shows a HOPE program which optimistically evaluates the same numerical integral. The optimistic assumption is that the function is nearly linear, and so a trapezoidal approximation can be used. If true, the approximation is much faster to evaluate than the exhaustive numerical integral ($O(1)$ versus $O(N)$). However, the assumption

```
Exhaustive(float start, float end, float step)
{
  float  x, integral;

  for (x = start ; x <= end ; x += step) {
    integral = integral + function(x);
  }
  integral = integral/2;
  return(integral)
}
```

Figure 3.1: Pessimistic Numerical Integration

```
/* Optimistic Integration */
receive(float start, float end, float step):
aid_t    Linear;
float    intg;
float    i, j, step;

approx = (f(i) + f(j))*(j-i)/2;
send(Checker, Linear, approx, i, j, step);

if (guess(Linear)) {
  intg = approx;
} else {
  receive(intg);
}
/* now go use intg to compute something ... */
```

Figure 3.2: Optimistic Numerical Integration

```
/* Checker */
receive(Assumption, Approximation, start, end, step);

integral = Exhaustive(start, end, step);
/* check the assumption */
if (abs(integral - Approximation) > tolerance*integral) {
  affirm(Assumption);
} else {
  deny(Assumption);
  send (Worker, integral);
}
```

Figure 3.3: Verify Numerical Integration Assumption

```
/* ATM Process */
. . .
pin = get_PIN();
pin_ok = call authorize_PIN(pin, user_id);        /* RPC */
if (pin_ok = OK) {
  process_transaction();
} else {
  report_error();
}
/* . . . end process */

/* authorize_PIN process */
. . .
for (;;) {
  receive (pin, user_id);
  /* locally check the PIN code */
  if (pin_ok(pin, user_id) = OK) {
    return OK;
  } else {
    return(NOT_OK);
  }
}
/* . . . end process */
```

Figure 3.4: Before Expected Result Transformation

may not be correct, and so a checking procedure (Figure 3.3) that executes the exhaustive integration is spawned and executed in parallel. If the approximation is found to be within desired tolerance of the exhaustive integral, then the assumption is affirmed. If not, then the assumption is denied, rolling back the worker process to the point of the optimistic assumption, and the exhaustive function sends the correct result to the worker.

This example illustrates a separate process concurrently checking the optimistic assumption. Concurrent verification of such assumptions hides latency by not waiting for likely results. We apply this idiom to scientific applications to avoid the latency of long computations by optimistically assuming that cheap approximations are close enough.

3.1.2 Expected Result

HOPE is also applicable to the distributed processing domain, where remote communications latency can be avoided by making optimistic assumptions about the behaviour of remote tasks, such as assuming that a remote database server will grant an access lock [15], or that a remote function will return an expected result [5].

```
/* ATM Process */
aid_t    pin_ok; /* the assumption ID for the expected
                    result assumption that pin_ok = OK */

pin = get_PIN();
pin_ok = aid_init();

send(WorryWart, pin_ok, pin, user_id);
if (guess(pin_ok)) {
  /* Implicitly assume that pin_ok = OK */
  process_transaction();
} else {
  report_error();
}
/* . . . end process */

/* WorryWart Process */
aid_t    Assumption; /* use this confirm or deny guess */
int      Pin;        /* PIN to be authenticated */

receive(Assumption, Pin, User);
pin_ok = call authorize_PIN(Pin);                    /* RPC */
if (pin_ok = OK) {
  affirm(Assumption); /* assumption was correct */
} else {
  deny(Assumption);   /* assumption was wrong, deny the guess */
}
/* end process */
```

Figure 3.5: After Expected Result Transformation

Figure 3.4 presents a program for processing ATM banking requests. In this example, an ATM gets a PIN from the user, checks the validity with the authorization process, and then proceeds to carry out the transaction if the code is correct. Users almost always enter a legitimate PIN; therefore, we can optimistically assume that the PIN is correct and continue with the transaction. The **expected result** optimistic assumption is to assume that a function will return an expected value, such as success in the case of verifying an ATM PIN.

Figure 3.5 presents the results of applying the expected result optimistic assumption. The ATM process has been changed to optimistically assume that the PIN is correct, a WorryWart process (assumed to be running already) has been added to asynchronously check the results coming back from authorize_PIN, and authorize_PIN remains the same.

Since process_transaction() is a separate process, the optimistic processing in the ATM

```
/* Worker Process */
line = call print("Total is ", total); /* S1 */    /* RPC */
if (line > PageSize) {
  call newpage();                    /* S2 */    /* RPC */
}
call print("Summary ...");           /* S3 */    /* RPC */
/* ... end process */
```

Figure 3.6: Before Call Streaming Transformation

```
/* Worker Process */
aid_t    PartPage, Order;

PartPage = aid_init();
Order = aid_init();
send(WorryWart, PartPage, Order, total);
if (guess(PartPage)) {
  /* do nothing */                   /* S2 */
} else {
  call newpage();                    /* S2 */      /* RPC */
}
guess(Order);
call print("Summary ...");           /* S3 */      /* RPC */
/* ... end process. */


/* WorryWart Process(PartPage, total) */
aid_t    PartPage, Order;

receive(PartPage, Order, total);
line = call print("Total is ", total); /* S1 */     /* RPC */
free_of(Order);
if (line < PageSize) {
  affirm(PartPage);
} else {
  deny(PartPage);
}
/* ... end process */
```

Figure 3.7: After Call Streaming Transformation

process has causal descendants in other processes. If pin_ok is denied, then the processing executed by process_transaction must be rolled back along with the ATM process. HOPE automates dependency tracking and rollback, simplifying the optimistic programmer's task.

3.1.3 Bacon and Strom's Call Streaming

Let $S_1;S_2$ be two sequential operations in process P, where both are remote procedure calls (RPCs). Because RPCs are synchronous, the calling process waits idle until a response is received from the remote machine. We can avoid this latency by executing S_1 in parallel with S_2, i.e., transforming the synchronous RPCs into asynchronous messages. If S_1 and S_2 are completely independent, i.e., S_2 does not depend on any results from S_1, then it is easy to execute S_1 and S_2 concurrently.

However, what if S_1 and S_2 are not independent? The execution of S_2 may be a function of the response of the RPC done by S_1. For example, Figure 3.6 shows a program fragment in which S_1 is an RPC that prints a summary total and returns the current line number of the page. S_2 takes the line number and checks to see if the line number now exceeds page size. If it does, then S_2 creates a new page; otherwise execution can immediately proceed to S_3.

Bacon and Strom [5] present an algorithm for optimistically parallelizing two such statements. We can still parallelize S_1 and S_2 (and hence the statements after S_2) by making the (likely) optimistic assumption that the report does not end exactly at the bottom of the page, i.e., line < PageSize. Figure 3.7 shows how we can parallelize S_1 and S_2 (and hence the statements after S_2) as follows. S_1 is executed in the WorryWart process while S_2 (and the statements after S_2) is executed in the Worker process.

The Worker process concurrently executes S_1 by spawning the WorryWart process. The Worker process executes **guess(PartPage)**, and based on the optimistic **True** return code, proceeds to execute S_2 and S_3 as if the line count were in fact less than PageSize, and prints "**Summary**..." without forcing a new page. However, the assumption that line < PageSize has yet to be verified and the computation in the Worker process is now speculative.

If line < PageSize is not valid, then **deny(PartPage)** is executed. This causes the Worker process to rollback to the point that the **guess** primitive was executed. Any processes that Worker sent a message to while speculative are also rolled back. The Worker process now resumes execution with a value of **False** returned from **guess(PartPage)**. In this example, this means that the Worker knows that the line value exceeded PageSize, and so calls newpage().

The execution of S_2 and the statements after S_2 must not interfere with S_1's execution. S_3's message may arrive at the remote machine ahead of the message from S_1 in the Worry-

Wart process. The remote process becomes speculative and is dependent on the assumption identifier **Order** and by transitivity the WorryWart process becomes dependent on **Order**. Because S_3 changes the line number, S_1's test is invalidated. The **free_of(Order)** primitive is used to detect this causality violation and force rollbacks to solve the problem.

3.2 Linguistic Assumptions

Like Linda [13] and Concert [4], HOPE is not a complete programming language. Rather it is a programming model for optimism, embodied as a set of primitives designed to be embedded in some other programming language. There are very few restrictions on the kinds of systems in which HOPE can be embedded.

HOPE is designed to hide the latency of components of a system by proceeding concurrently with such component operations; thus there must be concurrent processes. HOPE does dependency tracking by marking communications between processes with references to AIDs, and so inter-process communications must be explicit, as is the case in many concurrent and distributed systems [4, 12, 30, 62]. Shared memory programming models would be impractical with HOPE, because HOPE dependency tracking operations would have to accompany every shared memory access.

Chapter 4

Operational Semantics[1]

This chapter defines the formal operational semantics of the HOPE primitives introduced in Chapter 3. Defining the precise meaning of each of the primitives allows us to reason about the implementation of the HOPE prototype to help assure its correctness, and also to reason about HOPE programs. Rare as optimistic systems are, formally defined optimistic systems are even more scarce: HOPE is one of the first formally specified systems for expressing optimism [39].

Formal meanings are defined for the following primitives:

guess(X)	The executing process makes an assumption and associates that assumption with the assumption identifier X. **guess** speculatively returns **True** immediately, and returns **False** if rolled back.
affirm(X)	The executing process asserts that the optimistic assumption associated with assumption identifier X is confirmed as true.
deny(X)	The executing process asserts that the optimistic assumption associated with assumption identifier X is found to be false.
free_of(X)	The executing process asserts that the computation does not depend on the assumption identifier X.

Chapter 3 discusses "tagging" messages between user processes with the set of assumption identifiers that the sender depends on at the time of the send, thus automating interprocess dependency tracking. We omit tagged messages, because they can be simulated by simply bracketing the receipt of tagged messages with the appropriate **guess** primitives, which is precisely what the HOPE prototype does.

Section 4.1 describes the notation that we use in Sections 4.2 and 4.3 to formally define the semantics of our primitives, and presents some theorems resulting from our semantics,

[1]The meaning of HOPE

proving that what one would intuitively expect from the primitives is indeed provided. Section 4.4 summarizes this chapter's results.

4.1 Preliminaries

We use an operational approach for defining the semantics of the HOPE constructs. The central aim in this approach is to define an **abstract machine** for interpreting programs of the subject language. The machine interprets a program by passing through a sequence of discrete states. This approach requires that the structure of states and the allowable transitions from one state to another be specifically defined.

We did not use any of the various metalanguages [28, 45] for describing the semantics of programming languages, regarding them as inadequate for our use for the following reasons. First, some of the metalanguages are for sequential languages. Second, most are state-based in the sense that the only thing defined is allowable transitions from one state to another. Rollback requires deleting states from process execution histories, which cannot be modeled as state transitions. Third, although there are operational semantic definitions of nonsequential programs, they are usually described as sequences of global states and event occurrences. In nonsequential programs, local states (i.e., those of sequential processes) are considered as parts of global states; hence, local events affect given global program states. We find this approach awkward because rollback is applied to a subset of the processes that constitute the distributed program. Although recent work has attempted to better distinguish between local and global states, this work is usually oriented towards a proof system. Currently, we are interested in defining a semantics that is relatively easy to understand in order to be used by language implementors.

The approach taken here is that a distributed program, *Prog*, consisting of a collection of communicating sequential processes, $P, Q, \ldots$, is a generator of execution sequences or histories. Each process P generates an execution sequence of process states.

We will now define the components of the abstract machine. We will start by describing the variables and states associated with a process P. Each process P consists of the following components:

V_P A finite set of **state variables**. Some of these variables represent **data variables**, which are explicitly manipulated by the program text. Other variables are **control variables**, which represent, for example, the location of control

in P and will be be denoted by PC. Three other variables of significance in this work are G, I, and IS, which will be discussed later.

Σ_i A set of states. Each state $S \in \Sigma_i$ is an interpretation of V_i, assigning to each variable in V_i a value over its domain.

The following gives a formal definition of execution history (sequence) of a process P.

Definition 4.1 *An execution* **history** *or* **sequence** *for process P is a finite or infinite sequence of* **states** *separated by* **events** *that alter the states: $H_P : S_0E_0S_1E_1S_2E_2....$ Each event is the execution of a statement by the process P.*

We will now identify relevant data and control variables. Each assumption identifier is a data variable representing an assumption. Any process in the system can apply HOPE primitives to any assumption identifier. Each assumption identifier is associated with a tuple of control variables. We represent an assumption identifier as follows:

Definition 4.2 *An* **assumption identifier** *X is associated with a control variable $X.DOM$ (Depends On Me).*

The control variable DOM is used for dependency tracking. It is invisible to the programmer in the same sense that program counters are invisible. The usage of these control variables will become clearer in the next section.

An **interval** is a subsequence of an execution history that corresponds to the smallest granularity of rollback that may occur, i.e., a process is rolled back one or more intervals. An interval is said to be **speculative** if that interval is rolled back; otherwise, the interval is said to be **definite**.

We use the control variables I and IS to represent the current interval that the state of a process is in, and the set of intervals in a process's history that are speculative, respectively. Formally, we define intervals as follows:

Definition 4.3 *A* **guess point** *in an execution history, H_P is the event E_i representing the execution of a* **guess** *primitive.*

Definition 4.4 *An* **interval**, *in process P, is defined as a subsequence of states in H_P between two guess points, between the start of the process and the first guess point, and*

between the last guess point and the current state of the process. An interval A is associated with the tuple of control variables A.PS (Previous State), A.IDO (I Depend On), A.IHD (I Have Denied), A.PID (Process ID). The control variables are considered part of the state of the process, but are transparent to the programmer.

We use control variables to denote names of intervals. Like the assumption identifier variables, the usage of these control variables should become clearer in the next section. Basically, these control variables are used for dependency tracking associated with computations. Again, all these control variables are transparent to the programmer.

We will use A, B, C to denote intervals and X, Y, Z to denote assumption identifiers.

The following definition of *dependence* is useful for discussing the relationships between intervals and assumption identifiers:

Definition 4.5 *An interval A* **depends** *on an assumption identifier X if the interval A is made definite* only if *the assumption identifier X is definitely affirmed.*

We will now define the set and sequence operations that will be used. Some of the control variables (as will be seen in the next section) are sets. All set operations are allowed. With respect to the execution sequences, we require the following: a concatenation operation for adding a state to an execution sequence, and an operation $last(H_P)$ used to denote the current state of H_P. Although we may have infinitely long execution sequences, at any one time there is exactly one current state. $Del(H_P, A)$ deletes the interval A from the sequence H_P. Theorem 4.1 will show that this deletion will only apply to the suffix of an execution sequence ending with the current state.

4.2 Semantics of the HOPE Constructs

This section defines the HOPE constructs in terms of the abstract machine described in the previous section by defining the effects of each HOPE primitive on the execution sequences. All sequences are initially null and all sets are initially empty.

4.2.1 Guess

Process P executing a **guess(X)** primitive in state S_i asserts an optimistic assumption and associates the assumption identifier X. The creation of the new interval, A, requires that a

checkpoint be taken of the current state, including the program counter. This is represented as follows:

$$A.PS \leftarrow S_i \tag{4.1}$$

$$A.PID \leftarrow P \tag{4.2}$$

We should note that the process identifier (in PID) is also recorded. The recording of the process identifier is a naming convenience.

The interval A is speculative, and is dependent on the optimistic assumption associated with the assumption identifier X. In other words, the interval A is rolled back if the optimistic assumption associated with X is discovered to be false. If the interval preceding A is speculative then the interval A is also dependent on the optimistic assumptions that the preceding interval is dependent on. For interval A, the set $A.IDO$ is used to track A's dependencies. At the beginning of interval A (i. when the **guess** primitive is executed), the dependencies are as follows:

$$A.IDO \leftarrow (S_i.I).IDO \cup \{X\} \tag{4.3}$$

Associated with each assumption identifier X is the set $X.DOM$ that tracks the intervals that are dependent on X. Thus X records its new dependent interval A as follows:

$$X.DOM \leftarrow X.DOM \cup \{A\} \tag{4.4}$$

The state S_{i+1} is constructed from S_i as follows:

$$\begin{aligned} S_{i+1} &\leftarrow S_i \\ S_{i+1}.I &\leftarrow A \\ S_{i+1}.IS &\leftarrow S_{i+1}.IS \cup \{A\} \\ S_{i+1}.G &\leftarrow \textbf{True} \end{aligned} \tag{4.5}$$

The state variable, IS, is the set of all speculative intervals leading to state S.

The **guess** primitive initially returns a value of **True**, as recorded in $S_{i+1}.G$. If rollback occurs, execution resumes by resetting the PC to the point where **guess** was called, and

returning a G value of **False**. Thus the effect of the **guess** primitive on the program counter is merely to increment it to the next operation.

The execution sequence associated with P is updated as follows:

$$H_P \leftarrow H_P S_{i+1} \tag{4.6}$$

4.2.2 Affirm

The **affirm(X)** primitive asserts that the optimistic assumption associated with X has been determined to be true. There are two cases. In the first case, the **affirm(X)** was executed in process P in state S_i where $S_i.I = \emptyset$. Therefore, the execution of the **affirm(X)** cannot be undone (**definite affirm**). Thus, all the intervals that depend on X can try to decide on whether to become definite or not as follows:

$$\begin{aligned}
&\text{If } S_i.I = \emptyset \text{ then } \forall B \in X.DOM: \\
&\quad S \leftarrow last(H_{B.PID}), \\
&\quad ((S.IS).B).IDO \leftarrow B.IDO \backslash \{X\}, \\
&\quad H_{B.PID} \leftarrow H_{B.PID} S && (4.7) \\
&\text{If } B.IDO = \emptyset \text{ then } \textbf{finalize(B)} && (4.8) \\
&X.DOM \leftarrow X.DOM \backslash \{B\} && (4.9)
\end{aligned}$$

finalize(B) transforms B from a speculative to a definite interval. **finalize** is not a part of the user's programming model, and is just used here as a shorthand notation for the effects described in Section 4.2.5.

In the second case, an **affirm(X)** was executed in a process P in state S_i where $S_i.I \neq \emptyset$. Therefore interval A may be rolled back, thus the execution of **affirm(X)** may be undone. This is called a **speculative affirm**. Each of the assumption identifiers that interval A is dependent on must keep track of the intervals that depend on X. This is reflected as follows:

$$\forall Y \in A.IDO: \quad Y.DOM \leftarrow Y.DOM \cup X.DOM \tag{4.10}$$

All intervals dependent on X are now dependent on the assumption identifiers that the interval A is dependent on.

$$\text{If } A.IDO \neq \emptyset \text{ then } \forall B \in X.DOM : \tag{4.11}$$

$$S \leftarrow last(H_{B.PID}),$$

$$((S.IS).B).IDO \leftarrow (B.IDO \cup A.IDO) \backslash \{X\},$$

$$H_{B.PID} \leftarrow H_{B.PID} S \tag{4.12}$$

$$\text{If } B.IDO = \emptyset \text{ then } \textbf{finalize(B)} \tag{4.13}$$

$$X.DOM \leftarrow X.DOM \backslash \{B\} \tag{4.14}$$

If A is dependent on only X at the time A asserts **affirm(X)** (i.e., $X.DOM = \{A\}$, called **self affirm**) then Equation 4.12 will cause $A.IDO$ (and possibly other IDO sets) to become null, in which case Equation 4.14 produces results similar to a definite affirm.

Only one **affirm** or **deny** primitive may be applied to a given assumption identifier. Multiple **affirm** primitives are redundant; once affirmed, it's affirmed. Similarly, multiple **deny** primitives are redundant. Conflicting **affirm** and **deny** primitives have no meaning: An assumption cannot be both true and false. To eliminate this kind of confusing and conflicting notation, more than one **affirm** or **deny** primitive applied to a single assumption identifier, in any combination, is a user error, and the meaning is undefined.

4.2.3 Deny

The **deny(X)** primitive asserts that the optimistic assumption associated with X has been determined to be false. There are two cases. In the first case, the **deny(X)** was executed in process P in state S_i of interval A where $S_i.I = \emptyset$ or $X \in A.IDO$. This implies that P is not dependent on any other assumption identifiers except perhaps X. Therefore, the execution of the **deny(X)** cannot be undone by another process (**definite deny**). Thus, all the intervals that depend on X must roll back. This is expressed as follows:

$$\text{If } S_i.I = \emptyset \vee X \in A.IDO \quad \text{then} \quad \forall B \in X.DOM : \textbf{rollback(B)} \tag{4.15}$$

In the second case, a **deny(X)** in an interval A dependent on other assumption identifiers other than X implies that A may be rolled back; thus **deny(X)** may be undone. This is called a **speculative deny**. The set IHD records assumption identifiers that have been denied, to be applied when the interval is made definite.

$$\text{If } S_i.I \neq \emptyset \wedge X \notin A.IDO \text{ then}$$
$$S_{i+1} \leftarrow last(H_P),$$
$$((S_{i+1}.IS).A).IHD \leftarrow A.IHD \cup \{X\},$$
$$H_P \leftarrow H_P S_{i+1} \tag{4.16}$$

deny(X) becomes definite when the interval A is made definite, i.e., when $A.IDO = \emptyset$.

4.2.4 Free_of

The execution of a **free_of(X)** primitive in process P in state S_i of interval A means that interval A must not depend on the optimistic assumption associated with the assumption identifier X. The execution of **free_of(X)** in an interval A inspects the $A.DOM$ set for the assumption identifier X. If X is not found, then the requirement is satisfied and the optimistic assumption associated with X has been confirmed: the specified ordering constraint was not violated, and the equivalent of an **affirm(X)** is executed. However, if the assumption identifier X was found in $A.DOM$, then the ordering constraint associated with X was violated, and the equivalent of a **deny(X)** is executed. The formal specification is as follows:

$$\text{If } S_i.I = \emptyset \text{ then definite } \textbf{affirm(X)} \tag{4.17}$$
$$\text{Else if } X \notin A.DOM \text{ then speculative } \textbf{affirm(X)} \tag{4.18}$$
$$\text{Else speculative } \textbf{deny(X)} \tag{4.19}$$

Like **affirm** and **deny**, **free_of** "consumes" it's argument: it is an error for more than one **affirm**, **deny**, or **free_of** primitive to be applied to the same assumption identifier.

4.2.5 Finalize

Finalizing interval A makes A a permanent part of the history $H_{A.PID}$ by transforming A from a speculative to a definite interval. Finalizing A has the precondition:

$$A.IDO = \emptyset \tag{4.20}$$

The effect of **finalize(A)** on process $A.PID$ is as follows:

$$S \leftarrow last(H_{A.PID}),$$
$$S.IS \leftarrow S.IS \backslash \{A\},$$
$$H_{A.PID} \leftarrow H_{A.PID} S \tag{4.21}$$

$$\forall X \in A.IHD \quad (\forall B \in X.DOM \quad \textbf{rollback(B)}) \tag{4.22}$$

$$\text{If } last(H_{A.PID}).IS = \emptyset \text{ then}$$
$$S \leftarrow last(H_{A.PID}),$$
$$S.I \leftarrow \emptyset,$$
$$H_{A.PID} \leftarrow H_{A.PID} S \tag{4.23}$$

Speculative execution of HOPE primitives now become definite. In particular, the speculative execution of any **deny(X)** primitives now become definite, and so all of the intervals dependent on the assumption identifiers in $A.IHD$ are rolled back by Equation 4.22.

If A was the last and only interval in the history of process $A.PID$ to date, then $A.PID$ no longer has a current interval, and becomes definite as specified by Equation 4.23.

4.2.6 Rollback

Rolling back interval A truncates the history of $A.PID$ immediately before the start of A, and all subsequent states in the history are discarded. Process $A.PID$ then resumes from the **guess** primitive, but returning **False** instead of **True**.

$$H_{A.PID} \leftarrow Del(H_{A.PID}, A),$$
$$S \leftarrow A.PS,$$
$$S.G \leftarrow \textbf{False},$$
$$H_{A.PID} \leftarrow H_{A.PID} S \tag{4.24}$$

In addition, speculative execution of **affirm** and **deny** primitives must be undone. Speculative execution of **deny** primitives are trivially undone because they are never applied: They die with the interval inside the IHD set. Rollback of a speculative **affirm(X)** is

considered equivalent to a **deny(X)**[2]. Since a speculative **affirm(X)** by A added all of $A.IDO$ to all of the intervals that depend on X, all of those intervals also will be rolled back as a result of the definite **deny** that caused the rollback of A, and so no further action is required to undo the effects of a speculative **affirm** on other interval's IDO sets.

The only remaining set manipulations to consider are the changes to sets imposed by **free_of** primitives. However, the **free_of** primitive's effects are specified entirely in terms of speculative and definite **affirm** and **deny** primitives, and so no special treatment is required.

The following Lemma shows that for all intervals and assumption identifiers, if an interval A records that it is dependent on an assumption identifier X, then the assumption identifier X also records that A is dependent on X. This Lemma is used in proving subsequent theorems.

Lemma 4.1 *For all intervals A and assumption identifiers X, $X \in A.IDO$ if and only if $A \in X.DOM$.*

Proof: This proof is based on demonstrating that if an assumption identifier X is inserted into $A.IDO$, then A is also inserted into $X.DOM$, and that if X is taken out of $A.IDO$, then A is also taken out of $X.DOM$.

Equations 4.3 and 4.12 are the only operations where assumption identifiers are inserted into IDO sets. In the first case, Equation 4.3 inserts assumption identifier X into $A.IDO$, and symmetrically Equation 4.4 inserts A into $X.DOM$.

In the second case, Equation 4.10 inserts all of the intervals that depend on X into the DOM sets of all of the assumption identifiers that A depends on, and symmetrically Equation 4.12 inserts all of the assumption identifiers that A depends on into the IDO set of all of the intervals that depend on X. Thus, in all cases, if an assumption identifier is inserted into an interval's IDO set, then the interval is inserted into the assumption identifiers DOM set.

Similarly, Equations 4.7 and 4.12 are the only two operations that remove assumption identifiers from IDO sets. In both cases, assumption identifier X is removed from $B.IDO$,

[2]Applying deny to rollback a speculative affirm is a conservative approximation, because it is difficult to roll back the effects of speculative affirm while preserving the full expressiveness of HOPE. Appendix B provides an alternative semantics for HOPE that completely rolls back the effects of speculative affirm primitives without using deny, at some cost in expressiveness.

and interval B is symmetrically removed from $X.DOM$. Therefore, since in all cases, insertion into an IDO set symmetrically necessitates insertion into a DOM set, and removal from an IDO set symmetrically necessitates removal from a DOM set, we have for all intervals A and assumption identifiers X, $X \in A.IDO$ if and only if $A \in X.DOM$. □

The rollback of an interval implies that all intervals occurring after A also are rolled back. Therefore **rollback** truncates history. This is shown by the following theorem:

Theorem 4.1 *If an interval A is rolled back in a process P then for all intervals B, where B occurs after A in H_P, B is also rolled back.*

Proof: We will first show that for all intervals that occur after interval A in H_P, $A.IDO \subset B.IDO$. There are two cases to consider that correspond to IDO updates. The first corresponds to the initial creation of an interval, and the second corresponds to the effects of speculative execution of **affirm** primitives.

This is done by induction on the number of intervals that occur after interval A. By definition, when the first interval B following immediately after A is created, $B.IDO = A.IDO \cup \{X\}$ where X is the assumption identifier in the **guess** that created B. We can immediately conclude that $A.IDO \subset B.IDO$. Assume that the first k intervals to occur after A in H_P are such that $A.IDO$ is a subset of the IDO sets associated with each of these intervals. Let B' be the k^{th} interval and B be the $k+1^{th}$ interval. By definition, when interval B is created, its associated IDO set is $B'.IDO \cup \{X\}$. Thus $B'.IDO \subset B.IDO$. By the inductive hypothesis, we have that $A.IDO \subset B'.IDO$, thus we can immediately conclude that $A.IDO \subset B.IDO$.

However, speculative **affirm** primitives also cause changes in interval's IDO sets, thus we need to show that it is always the case that $A.IDO \subset B.IDO$. By Lemma 4.1, if an assumption identifier X is in both $A.IDO$ and $B.IDO$, then both A and B are in $X.DOM$. Thus, all of the other operations that affect IDO sets (Equations 4.7 and 4.12), which always use DOM sets to select the interval IDO sets to manipulate, will affect A and all of the intervals that follow A in H_P in the same way. Thus it is always the case that $A.IDO \subset B.IDO$ for all intervals B that follow A in H_P.

By definition, the only primitive that can cause A to be rolled back is a definite **deny** of some assumption identifier $X \in A.IDO$. Since $A.IDO \subset B.IDO$ for all B that follow A in H_P, $X \in B.IDO$ for all B that succeed A in $H_{A.PID}$. Therefore, the same definite

deny(X) also will cause all following intervals to be rolled back. Therefore, rollback of A will result in a truncation of H_P. □

The following Theorem shows that once an interval's IDO set becomes empty, it can never be rolled back.

Theorem 4.2 *For any interval A, if* $A.IDO = \emptyset$ *then* **rollback(A)** *will never occur.*

Proof: From Equations 4.15 and 4.22, an interval A can only be rolled back if $A \in X.DOM$ for some assumption identifier X and a definite **deny(X)** has occurred. But from Theorem 4.1, if $A.IDO = \emptyset$, then there does not exist an assumption identifier X such that $A \in X.DOM$. Therefore, no **deny(X)** can affect A, and A cannot be rolled back. □

4.3 Semantic Implications

The preceding specifications describe the meaning of the HOPE primitives in detail, but we would like to have some assurance that they will behave in a way that the programmer would intuitively expect. If **affirm** is asserted for all of the assumption identifiers that an interval depends upon, then the programmer expects that the interval will be made definite. Conversely, the programmer expects that the interval will be made definite only if **affirm(X)** is asserted for all of the assumption identifiers X that the interval depends on. The theorems presented in this section prove that these expectations are preserved by the definitions of the HOPE primitives.

Lemmas 4.2 and 4.3 are combined together in Theorem 4.3 to show that if all of the optimistic assumptions associated with the assumption identifiers that an interval B depends on are confirmed to be true by intervals that are themselves eventually made definite, then B will be made definite. Lemma 4.2 below shows that the effects of a speculative **affirm** primitive are identical to the effects of a definite **affirm** if the asserting interval is eventually made definite.

Lemma 4.2 Affirm Transitivity: *Let B be an interval that depends on an assumption identifier X. The effect of executing* **affirm(X)** *within a speculative interval A upon B.IDO and X.DOM, followed by A eventually being made definite, is the same as the effect of a definite* **affirm(X)**.

Proof: Equations 4.7 and 4.9 define the effect of a definite **affirm(X)** as removing X from $B.IDO$, and removing interval B from $X.DOM$.

Let A be a speculative interval that executes **affirm(X)** for some $X \in B.IDO$. Equation 4.14 immediately removes B from $X.DOM$. Equation 4.12 will replace $X \in B.IDO$ with $A.IDO$. Let $\beta = B.IDO$ at the time that A executes **affirm(X)**, and $\alpha = A.IDO$ be the set of assumption identifiers that replace X.

Since the contents of α were added to β, we have $\alpha \subset \beta$. From Lemma 4.1 we have that for all assumption identifiers $Y \in \alpha : A \in Y.DOM \wedge B \in Y.DOM$. By Equations 4.9, 4.14 and 4.20, if **finalize(A)** has occurred, then $A.IDO = \emptyset$. Since all changes applied to $A.IDO$ are also applied to $B.IDO$, if $A.IDO = \emptyset$ then all of the assumption identifiers from α that were added to $B.IDO$ also will have been removed from $B.IDO$. Thus the impact of A executing **affirm(X)** followed by **finalize(A)** is the same as a definite **affirm(X)**. □

Lemma 4.3 proves that if definite **affirm(X)** is executed on all of the assumption identifiers X that an interval B depends on, then B will become definite.

Lemma 4.3 *For any interval B, if definite* **affirm(X)** *is applied to all assumption identifiers $X \in B.IDO$, then* **finalize(B)** *will result.*

Proof: Let B be an interval with an initial set of assumption identifiers $\gamma = B.IDO$.

Equation 4.7 expresses that for each $X \in \gamma$, each definite **affirm(X)** will remove X from $B.IDO$. Equation 4.9 will induce **finalize(A)** when the last assumption identifier in γ is affirmed. □

Using Lemmas 4.2 and 4.3, we can conclude that if all of the assumption identifiers that an interval B depends on are affirmed by intervals that eventually become definite, then B will become definite.

Theorem 4.3 *For any interval B, if* **affirm(X)** *is applied to all assumption identifiers $X \in B.IDO$ by intervals that eventually become definite, then* **finalize(B)** *will result.*

Proof: Lemma 4.3 shows that if all of the **affirm(X)** primitives are executed by definite intervals, then **finalize(B)** will result. Lemma 4.2 shows that speculative **affirm** primitives that are eventually made definite have the same effect as definite **affirm** primitives, and so **finalize(B)** will result in either case. □

Theorem 4.4 shows that **finalize(B)** will occur if and only if **affirm(X)** is executed on all of the assumption identifiers X that interval B depends on.

Theorem 4.4 *For all intervals B,* **finalize(B)** *occurs if and only if* **affirm(X)** *is applied to all of the assumption identifiers $X \in B.IDO$ by intervals that eventually become definite.*

Proof: From Theorem 4.3, we have the case that if **affirm(X)** is applied to all assumption identifiers $X \in B.IDO$, then **finalize(B)** will result.

We prove that **finalize(B)** implies that **affirm(X)** has been applied to all assumption identifiers $X \in B.IDO$ using proof by contradiction. Assume that **finalize(B)** has occurred, and that there exists an assumption identifier $X \in B.IDO$ that has not had **affirm(X)** executed.

Since **affirm(X)** has not occurred, no operation will have removed X from $B.IDO$. Therefore $B.IDO \neq \emptyset$, and by Equation 4.20, **finalize(B)** cannot have occurred. This contradicts the assumption that **finalize(B)** has occurred.

Therefore, for all intervals B, **finalize(B)** occurs if and only if **affirm(X)** is applied to all of the assumption identifiers $X \in B.IDO$ by intervals that eventually become definite. □

Lemma 4.4 establishes that if an assumption identifier X is speculatively affirmed by an interval A that depends on some other assumption identifier Y, then X depends on Y.

Lemma 4.4 *If an interval A depends on an assumption identifier Y and executes* **affirm(X)**, *then X will be definitely affirmed only if Y is definitely affirmed.*

Proof: We use proof by contradiction. Let X be definitely affirmed, and Y not be definitely affirmed.

From Section 4.2.2, we know that only one interval can execute **affirm(X)**. Since X was definitely affirmed, and A executed **affirm(X)**, then A must be the only interval to have executed **affirm(X)**, and from Lemma 4.2 we conclude that A must have been made definite. If A was made definite, then from Theorem 4.4 all assumption identifiers that A depended on must have been definitely affirmed. From the statement of our lemma, one of those assumption identifiers was Y, which means that Y must have been definitely affirmed, conflicting with our assumption. □

Lemma 4.4 produces Corollary 4.1, which shows that the depends-on relation between assumption identifiers is transitive:

Corollary 4.1 *If an assumption identifier X depends on another assumption identifier Y, and Y depends on a third assumption identifier Z, then X depends on Z.*

Proof: We use proof by contradiction. Let X be definitely affirmed, and Z not be definitely affirmed.

Given that X is definitely affirmed, and that X depends on Y, Lemma 4.4 gives us that Y must be definitely affirmed. Similarly, since Y is definitely affirmed, and Y depends on Z, Lemma 4.4 gives us that Z must be definitely affirmed. □

Finally, Theorem 4.5 shows that if an interval A executes **free_of(X)**, then $A.IDO$ does in fact remain free of X and all other assumption identifiers Y that depend on X.

Theorem 4.5 *For any interval A that executes* **free_of(X)**, *then either interval A never becomes dependent on assumption identifier X, or interval A is rolled back.*

Proof: We use proof by contradiction. Let A be an interval such that $X \in A.IDO$, and let the statement **free_of(X)** be executed within interval A. By inspection, Equation 4.19 will execute **deny(X)**, which will in turn apply Equation 4.15, which will roll back interval A. Thus if A depends on X and executes **free_of(X)**, then A is rolled back.

If A does not depend on X at the time it executes **free_of(X)**, then A can never become dependent on X. To become dependent on X, A would have to depend on some other AID Y that was speculatively affirmed by some interval B that depended on X. However, since the speculative execution of **free_of(X)** resulted in a speculative **affirm(X)**, there cannot exist another interval B that depends on X, because B's dependence on X will have been replaced with a dependence on $A.IDO$. Thus A can never depend on X. □

4.4 Summary

This chapter has formally defined the meaning of the HOPE primitives using an operational semantics. The operations are defined on **intervals** in the execution history of user processes, and **assumption identifiers** that identify particular optimistic assumptions.

HOPE tracks which intervals depend on which optimistic assumptions by maintaining dependency sets in the intervals and assumption identifiers. The dependency sets of an interval are as follows:

IDO I Depend On

IHD I Have Denied

The dependency set of an assumption identifier is as follows:

DOM Depends On Me set of intervals contingent on this assumption identifier

The HOPE operations are specified in terms of manipulations on the dependency sets, and specific actions (**finalize** and **rollback**) that are taken when certain properties of these sets occur.

Chapter 5

HOPE Algorithms[1]

This chapter describes a computational model, and the algorithms for realizing the HOPE primitives. Section 5.1 describes the computational model: concurrent processes exchanging messages. Section 5.2 describes the problems that can arise in the straightforward concurrent execution of HOPE primitives. This section also introduces a simple approach to solving these concurrency problems directly by introducing concurrency control, as well as the performance penalty that such an approach incurs. Section 5.3 describes a more efficient algorithm for the HOPE primitives that addresses the conflicts that concurrency introduces instead of resorting to concurrency control, and shows that the algorithm satisfies the semantics of Chapter 4. Section 5.4 summarizes the results.

5.1 Basic Model

HOPE is a small set of primitives suitable for embedding into programming environments that provide the following:

1. concurrent processes that communicate by passing messages
2. a mechanism to checkpoint and roll back these processes to a previous state

This section describes the basic structure of an algorithm for realizing the HOPE primitives in such an environment. The basic structure of an algorithm in such an environment is shown in Figure 5.1.

For purposes of discussion, we will refer to processes created by a HOPE programmer as **user processes**. Attached to each user process is a collection of modules for executing

[1] A basis for HOPE.

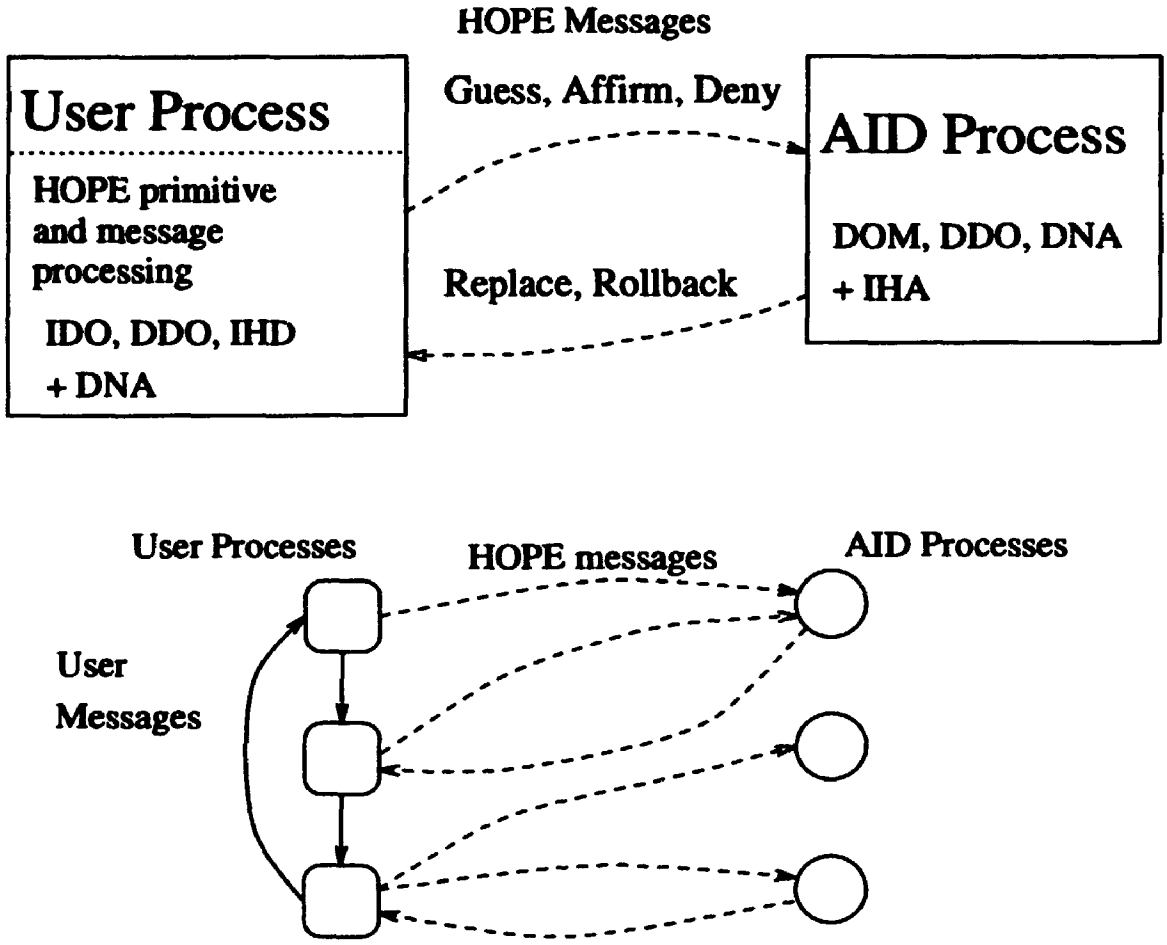

Figure 5.1: Structure of the Basic HOPE Model

HOPE primitives and for processing HOPE messages. Of primary interest is the **Control** module, which processes HOPE messages.

Each assumption identifier is encapsulated in an **AID process** (see Figure 5.1). As in Chapter 4, we will use $A, B, \ldots$ to denote intervals in the history of user processes, and $X, Y, \ldots$ to denote assumption identifiers. In addition, we will use $P_X, P_Y, \ldots$ to denote AID processes. *A.pid* denotes the process identifier of the process containing interval A, and *A.iid* denotes the interval identifier, which is unique within the process.

The intervals in a user process's history, and the dependency tracking sets associated with each interval, are stored in data structures contained in these HOPE modules in such a way that they are transparent to the user process. The dependency sets stored with each interval are as specified in Chapter 4 in Section 4.4. In addition, a **DDO** (Don't Depend

On) set of assumption identifiers that the interval may not depend on, and a **DNA** (Do Not Affirm) set of assumption identifiers that the user process must not affirm from within a given interval, are recorded with each interval.

AID processes store and process dependency tracking information relating to the assumption that they identify. Local modifications to dependency sets are then simply local operations, and modifications to remote sets become messages requesting the modification to that set sent to the appropriate user or AID processes. The dependency sets stored by an AID process include those specified in Chapter 4. In addition, each AID process also records an **IHA** (I Have Affirmed) set of assumption identifiers that have been affirmed using this assumption. Chapter 4 specifies that the rollback of a speculative **affirm** has the same effect as a **deny**, and the IHA set provides a list of AID processes that must be denied if the affirming interval is rolled back. Each AID process also records a *DDO* and *DNA* set of assumption identifiers, with meanings similar to those of the corresponding sets in user intervals.

The HOPE primitives (guess, affirm, deny, and free_of) are provided as simple functions attached to each user process. These functions make local modifications to the process history and local dependency sets, and then send messages to appropriate AID processes for further processing.

The AID processes receive and process messages sent from user processes, modifying their dependency tracking sets in response to their current state and the type and parameters of the current message being processed. The AID processes compute the remaining dependency set and history changes necessary, and send messages to other AID and user processes.

The messages sent to user processes are intercepted by the message passing system and given to the HOPE modules attached to each user process for processing. This ensures that the HOPE messages are transparent to the user processes.

5.2 The Problem: Interference

Chapter 4 specifies the execution of a HOPE primitive as a sequence of operations $a_1 a_2 a_3 \ldots$. Some of the operations require the update of variables of other processes, thus requiring interprocess communication. Because HOPE primitives may be executed by concurrent processes, they may be executed concurrently. Thus the remote data access of HOPE prim-

itive execution may result in concurrent access to shared data items. Using the definitions of Bernstein et al. [8, page 11], we can describe the interaction of HOPE primitive execution as follows.

Let a_i be an operation in the execution of a HOPE primitive by process P in interval A. Let b_i and b_j be two operations in the execution of another HOPE primitive by process Q in interval B. We then have the following:

Definition 5.1 *The execution of the HOPE primitives in distinct processes P and Q are* **interleaved** *if an operation a_i from P executes between two operations b_i and b_j from process Q.*

Definition 5.2 *If two operations a_i and b_j access the same data item, and at least one of them writes to the data item, then the two operations are said to* **conflict**.

Definition 5.3 *If the execution of two HOPE primitives produces interleaved, conflicting operations, then that execution of the two HOPE primitives is said to* **interfere**.

Bernstein et al. [8, page 1] describe an **atomic** operation such that:

1. each operation accesses shared data without interfering with other operations; and
2. if an operation terminates normally, then all of its effects are made permanent; otherwise it has no effect at all.

However, for our purposes, we are only concerned with the indivisibility of sequences of operations with respect to interleaving. Thus we use the following definition:

Definition 5.4 *An* **atomic** *operation accesses shared data without interfering with other operations.*

The manipulations specified in Chapter 4 for each HOPE primitive can be arbitrarily decomposed into operations $a_1a_2a_3\ldots$, as long as the individual operations a_i are atomic (i.e., not subject to interleaving), and thus any operation a_i is free of interference from the execution of other HOPE primitives.

As an example of conflicting HOPE primitives, consider the case in which an interval A in process P depends on the assumption identifier Y, and another interval B in process Q depends on the assumption identifier X, producing the following initial values:

$$
\begin{aligned}
A.IDO &= \{Y\} \\
B.IDO &= \{X\} \\
P_X.DOM &= \{B\} \\
P_Y.DOM &= \{A\}
\end{aligned}
$$

If P then executes an **affirm(X)** primitive in interval A, the expected result is that both A and B become dependent only on the assumption identifier Y. If Q subsequently executes an **affirm(Y)** primitive in interval B, then both assumption identifiers X and Y will have been definitely affirmed, and both intervals A and B should be finalized. The dualistic result that occurs if the **affirm** primitives are executed in the opposite order also results in finalizing both intervals.

However, if the execution of the two **affirm** primitives in this example is interleaved, then errors may result if conflicts occur. In particular, Equation 4.10 for speculative **affirm** in A updates the contents of $P_Y.DOM$, and in B updates $P_X.DOM$, and Equation 4.12 reads the contents of $P_X.DOM$ and $P_Y.DOM$.

Recall Equation 4.10:

$$\forall Y \in A.IDO : Y.DOM \leftarrow Y.DOM \cup X.DOM$$

and Equation 4.12:

$$
\begin{aligned}
\forall B \in X.DOM : \quad & S \leftarrow last(H_{B.PID}) \\
& S.(IS.(B.IDO)) \leftarrow (B.IDO \cup A.IDO) \backslash \{X\} \\
& H_{B.PID} \leftarrow H_{B.PID} S
\end{aligned}
$$

To more precisely illustrate the specific operations in this example, we decompose the execution of the two **affirm** primitives into the following arbitrarily atomic operations:

a_1 process P in interval A reads values for $A.IDO$, $B.IDO$, $P_X.DOM$, and $P_Y.DOM$

a_2 P's application of Equation 4.10

a_3 P's application of Equation 4.12

a_4 process P writing back its results

b_1 process Q in interval B reading values $A.IDO$, $B.IDO$, $P_X.DOM$, and $P_Y.DOM$

b_2 Q's application of Equation 4.10

b_3 Q's application of Equation 4.12

b_4 process Q writing back its results

If the execution of the **affirm** primitive in interval A completes before the execution of the **affirm** primitive in interval B begins (i.e., no interleaving), then the operations proceed as follows:

$$a_1 a_2 a_3 a_4 b_1 b_2 b_3 b_4$$

The operations then produce the following results:

a_1	Read:	$A.IDO = \{Y\}$, $B.IDO = \{X\}$, $P_X.DOM = \{B\}$, $P_Y.DOM = \{A\}$
a_2	Applying:	$\forall Y \in A.IDO : P_Y.DOM \leftarrow P_Y.DOM \cup P_X.DOM$
	Yields:	$\{A, B\} \leftarrow \{A\} \cup \{B\}$
a_3	Applying:	$\forall B \in P_X.DOM : B.IDO \leftarrow (B.IDO \cup A.IDO) \backslash \{X\}$
	Yields:	$\{Y\} \leftarrow (\{X\} \cup \{Y\}) \backslash \{X\}$
a_4	Write back:	$P_Y.DOM = \{A, B\}$, $B.IDO = \{Y\}$
b_1	Read:	$A.IDO = \{Y\}$, $B.IDO = \{Y\}$, $P_X.DOM = \{B\}$, $P_Y.DOM = \{A, B\}$
b_2	Applying:	$\forall Y \in B.IDO : P_Y.DOM \leftarrow P_Y.DOM \cup P_X.DOM$
	Yields:	$\{A, B\} \leftarrow \{A, B\} \cup \{B\}$
b_3	Applying:	$\forall A \in P_Y.DOM :$
		$A.IDO \leftarrow (A.IDO \cup B.IDO) \backslash \{Y\}$
	Yields:	$\emptyset \leftarrow (\{Y\} \cup \{Y\}) \backslash \{Y\}$

	Applying:	$B.IDO \leftarrow (B.IDO \cup B.IDO) \backslash \{Y\}$
	Yields:	$\emptyset \leftarrow (\{Y\} \cup \{Y\}) \backslash \{Y\}$
b_4	Write back:	$P_Y.DOM = \{A, B\},\ A.IDO = \emptyset,\ B.IDO = \emptyset$

However, any interleaving in which a_4 occurs after b_3 and b_4 occurs after a_3, will cause the execution of the two **affirm** primitives to interfere. Consider:

$$a_1 a_2 a_3 b_1 b_2 b_3 a_4 b_4$$

In the above interleaving of the example, the application of Equation 4.12 by process P in interval A is oblivious to the change in $P_X.DOM$ made by Equation 4.10 by process Q in interval B, and Equation 4.12 in interval B is similarly oblivious to the change in $P_Y.DOM$ made by Equation 4.10 in interval A. The execution proceeds as follows:

a_1	Read:	$A.IDO = \{Y\},\ B.IDO = \{X\},\ P_X.DOM = \{B\},\ P_Y.DOM = \{A\}$
a_2	Applying:	$\forall Y \in A.IDO : P_Y.DOM \leftarrow P_Y.DOM \cup P_X.DOM$
	Yields:	$\{A, B\} \leftarrow \{A\} \cup \{B\}$
a_3	Applying:	$\forall B \in X.DOM : B.IDO \leftarrow (B.IDO \cup A.IDO) \backslash \{X\}$
	Yields:	$\{Y\} \leftarrow (\{X\} \cup \{Y\}) \backslash \{X\}$
b_1	Read:	$A.IDO = \{Y\},\ B.IDO = \{X\},\ P_X.DOM = \{B\},\ P_Y.DOM = \{A\}$
b_2	Applying:	$\forall X \in B.IDO : P_X.DOM \leftarrow P_X.DOM \cup P_Y.DOM$
	Yields:	$\{B, A\} \leftarrow \{B\} \cup \{A\}$
b_3	Applying:	$\forall A \in P_Y.DOM : A.IDO \leftarrow (A.IDO \cup B.IDO) \backslash \{Y\}$
	Yields:	$\{X\} \leftarrow (\{Y\} \cup \{X\}) \backslash \{Y\}$
a_4	Write back:	$P_Y.DOM = \{A, B\},\ B.IDO = \{Y\}$
b_4	Write back:	$P_X.DOM = \{B, A\},\ A.IDO = \{X\}$

The updates resulting from process P's application of Equation 4.12 in interval A is only applied to interval B, and the updates resulting from process Q's application of Equation 4.12 in interval B is only applied to interval A, resulting in A and B exchanging their

dependencies. Instead of both X and Y being definitely affirmed, and both A and B being finalized, as should happen in this example, the result is for A to depend on X and B to depend on Y.

One way to avoid interference problems is to prevent the interleaved execution of HOPE primitives:

Definition 5.5 *A* **serial** *execution of two or more HOPE primitives is one in which for every pair of HOPE primitive executions, all operations of one primitive execution complete before any of the operations of the other primitive execution [8, page 13].*

Definition 5.6 *A* **concurrent** *execution of two or more HOPE primitives is one that is not serial.*

Serially-executed HOPE primitives cannot interfere with one another. Serial execution of HOPE primitives is trivially achieved, for example, by processing HOPE primitives in a single, sequential process.

However, serial execution of HOPE primitives imposes a serious limitation on the concurrent execution of the processes in a HOPE program. Instead, we can relax the restriction on HOPE primitive execution as follows:

Definition 5.7 *A* **serializable** *execution of two or more HOPE primitives is one that produces the same set of effects as some serial execution of the same set of HOPE primitives.*

Serializable executions can be achieved by applying one of many concurrency control algorithms [8, 29, 32, 36, 68] to the execution of HOPE primitives. However, the time and space requirements of incorporating a general concurrency control system within the HOPE run-time are prohibitive. The cost of executing a concurrency control protocol for every HOPE primitive would be excessive: The remote communications latency inherent in a concurrency control protocol is precisely the form of delay HOPE was designed to avoid.

Instead of using a heavy solution such as general concurrency control system to avoid conflicts, it is also possible to allow conflicts to occur and correct for the errors that result after the fact. Such an approach has the same effect as concurrency control (produce only serializable executions) but does so at a lower cost by exploiting specific knowledge of the HOPE primitives. We can show that such an algorithm is consistent with the semantics of

Chapter 4 by showing that the algorithm satisfies Theorems 4.4 and 4.5:

Theorem 4.4 *For all intervals B,* **finalize(B)** *occurs if and only if* **affirm(X)** *is applied to all of the assumption identifiers $X \in B.IDO$ by intervals that eventually become definite.*

Theorem 4.5 *For any interval A that executes* **free_of(X)**, *then either interval A never becomes dependent on assumption identifier X or any of X's causal descendants, or interval A is rolled back.*

Section 5.3 describes an algorithm that satisfies Theorems 4.4 and 4.5, but does so by applying the HOPE primitives without regard to the potential for conflict, thus avoiding concurrency control delays. Instead, the algorithms anticipate and accommodate the errors that result from interleaving conflicts.

5.3 The HOPE Algorithm

The primary purpose of optimistic primitives is to avoid latency, and so it is an important design criterion that all of the remote operations resulting from user processes executing HOPE primitives be asynchronous: user processes executing HOPE primitives should never have to wait for a message from another process. This section describes an algorithm to provide the HOPE primitives that is consistent with this design criterion, and uses only concurrent processes, messages, and a rollback facility for the processes.

The following subsections describe a progression of algorithms to implement HOPE. Subsection 5.3.1 describes a simplistic solution in which the dependency sets are updated without regard to interleaving conflicts. We show that this algorithm satisfies Theorem 4.4 under special circumstances: If the intervals executing concurrent **affirm** primitives do not mutually depend on the assumption identifiers being affirmed by the other intervals, then the algorithm detects and corrects for conflicts. See Subsection 5.3.1.4 for further details.

Subsection 5.3.2 extends this algorithm to detect and correct the problems that can result from interleaved execution of the HOPE primitives when the special circumstances do not apply, and shows that this algorithm satisfies Theorem 4.4 under all circumstances. Finally, Subsection 5.3.3 extends the algorithm to ensure that no ordering requirements resulting from **free_of** assertions are violated, and shows that this algorithm satisfies Theorem 4.5 as well as Theorem 4.4.

Type	From	To	Arguments	Meaning
Guess	User	AID	*iid*	Sender guesses recipient is true
Affirm	User	AID	*iid*, *IDO*	Sender affirms recipient, subject to *IDO*
Deny	User or AID	AID		Sender unconditionally denies recipient
Add_DDO	User	AID	$\{X\}$	Add X to recipient's *DDO*
Add_DNA	User	AID	*IDO*	Add *IDO* to recipient's *DNA*
Replace	AID	User	*iid*, *IDO*, *DNA*	Replace sender with *IDO* in *iid.IDO*, and add *DNA* to *iid.DNA*
Rollback	AID	User	*iid*	Rollback interval *iid*
Free	AID	AID	*DDO*	Check for **free_of** violations in recipient's *DNA*, and add *DDO* to the recipient's *DDO*
Free_ACK	AID	AID		Acknowledge a Free message

Table 5.1: Basic HOPE Messages

5.3.1 A Basic Algorithm

This section describes a basic algorithm for HOPE that implements the set updates, but does not prevent interleaving conflicts. We refer to this as **Algorithm 5.3.1**. The algorithm keeps user programs free of synchronization delay because at no point in the execution of a HOPE primitive does any user process wait for acknowledgment from any other process. AID processes do sometimes block and wait for results, but this waiting is always hidden from user processes, so that user processes will never block as a result of executing HOPE primitives.

When one process of the system requires set updates in another process of the system, the update is propagated in the form of a message. These messages fit into well-defined types. Table 5.1 lists these message types, describing the source and destination of the message, the arguments contained in the message, and the approximate meaning of the message.

The specifics of the messages are as follows. In the source and destination specification,

"User" indicates the HOPE modules attached to user processes as in Figure 5.1, and "AID" indicates an assumption identifier process. The text of a message from a user process to an AID process is denoted as:

$$< Type, iid, \{dependencies\} >$$

The *iid* field is the identity of the interval sending the message, and the $\{dependencies\}$ is the set of AIDs pertinent to the message, as specified in Table 5.1. Not all messages use all of the arguments to a message: Omitted arguments are considered to be $\emptyset$.

The text of a message from an AID process to a user process is denoted as follows:

$$< Type, iid, IDO, DNA >$$

For Replace messages, the IDO set is the set of AIDs that the AID sending the message should be replaced with (details to follow), and the DNA set is a set of additional AIDs that the interval *iid* should not be involved in affirming. Rollback messages do not use the IDO and DNA arguments, which are considered to be $\emptyset$.

The text of a message from one AID process to another is denoted as follows:

$$< Type, DDO >$$

The DDO argument is present only for Free messages, and is considered $\emptyset$ for Free_ACK messages.

In the Arguments field in Table 5.1, the arguments are the specified sets associated with the sender of the message. In the case of AID processes, this is simply the specified dependency sets. However, if the sender is a user process, then the dependency sets are those associated with the interval identified by the *iid* field.

As stated in Section 5.1, each of the HOPE primitives is provided in the form of a user-callable function. Each takes an assumption identifier[2] as its argument.

Primitive execution takes the form of local set modifications, followed by sending a message to the specified assumption identifier process, and possibly messages to other assumption identifier processes in the user process's dependency sets. All of the primitives except **guess** expect the argument to be the process identifier of an AID process. **guess** will also take such an argument and use it, but in addition, if the argument is $\emptyset$, then the **guess** primitive decides that this is a *new* optimistic assumption, and so spawns a new AID process to track the new optimistic assumption.

[2]Process identifier of the AID process.

5.3.1.1 AID Processes

An assumption identifier is used to represent an optimistic assumption. An assumption identifier (AID) process models an optimistic assumption by storing the state of the assumption identifier. The **affirm** and **deny** primitives applied to assumption identifiers are translated into Affirm and Deny messages sent to the AID process for processing. The remaining message types exist to carry out the bookkeeping required by speculative **affirm** executions. The execution of the AID processes are described using state machines that process these messages by constantly executing a loop receiving and processing messages.

The action taken by an AID process in response to a message is a function of the contents of the message and the state of the AID state machine. The AID state is represented by the dependency sets associated with the assumption identifier, and the variable **state** which abstractly represents what is known about the truth of the associated optimistic assumption. The **state** variable may take on the following values:

Cold	no primitives have yet been applied to the AID
Hot	AID has received a Guess message, but has not yet been affirmed
Maybe	AID has received an Affirm message, but was affirmed subject to the set IDO of other AID's also being affirmed
True	AID has been unconditionally affirmed
False	AID has been unconditionally denied

The AID process records the dependency sets associated with the assumption identifier as follows:

1. the AID dependency sets specified in Chapter 4 (DOM, DDO, and DNA)
2. the A_IDO (Affirm-I Depend On) dependency set

Each AID process P_X stores the set of dependency sets as specified in Chapter 4 (DOM, DDO, and DNA). However, interleaving of the execution of HOPE primitives may result in an AID process P_X being subject to both a **guess** primitive (say in interval A), and an **affirm** primitive (say in interval B). If such is the case, then the AID process P_X must know the set of AIDs that the affirming interval B depended on, so that A's **guess** can be processed appropriately. P_X records the set of assumption identifiers that the affirming interval B depended on in the **A_IDO**: Affirm-I Depend On, set of assumption identifiers.

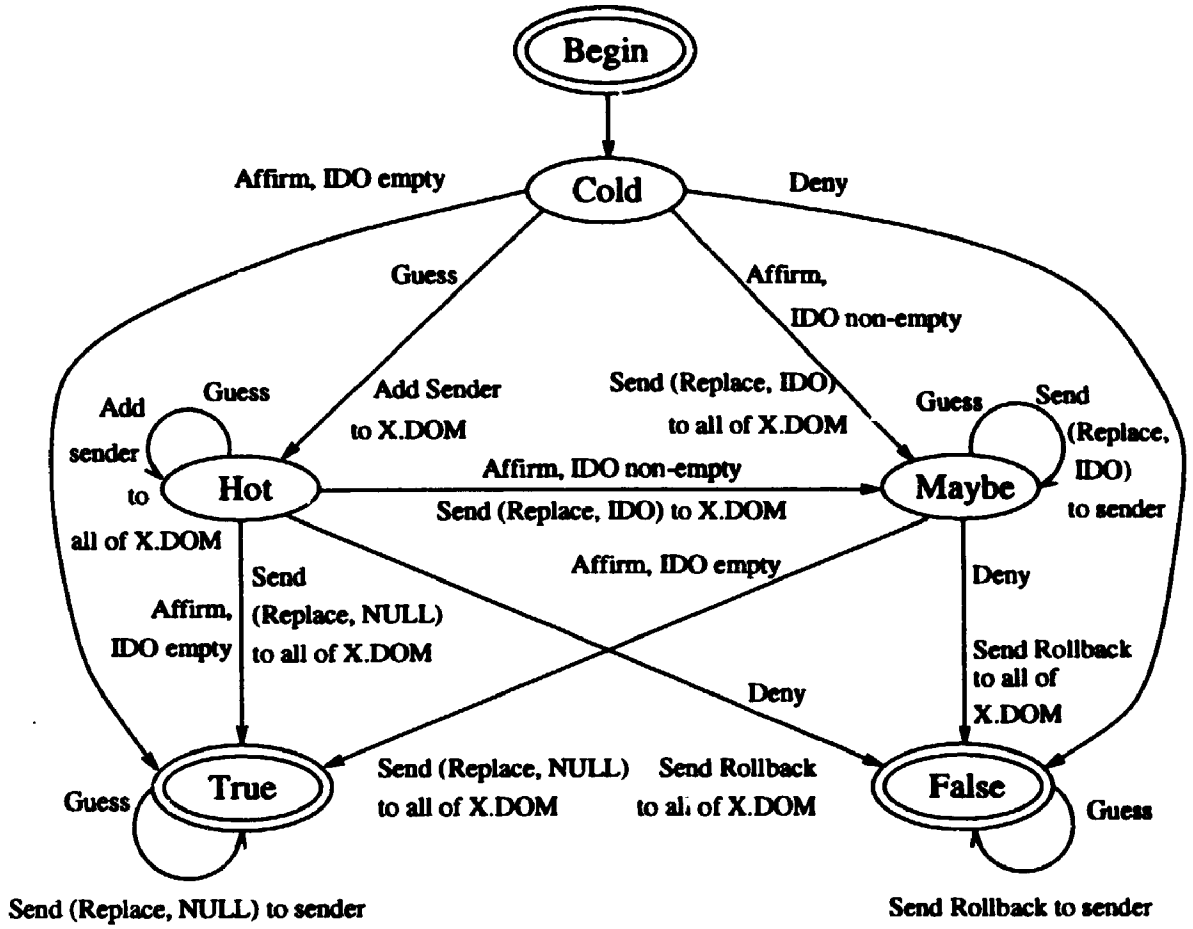

Figure 5.2: AID State Machine Diagram

The AID state machine begins in state **Cold**, and "terminates" in either state **True** or **False**. These states are terminal, in that once the machine enters either state, it never leaves. The AID process does not terminate, however, because there may still exist processes with pending **guess** primitives to be applied to a given assumption identifier, and the corresponding AID process must continue to function to respond appropriately to **guess** primitives. To garbage collect a given AID process, it must first be determined that no process in the system will apply a HOPE primitive to that assumption identifier. Such garbage collection can be done by counting references from assumption identifiers to AID processes [63].

Figure 5.2 shows the major state transitions of the AID state machine. The state **Begin** is the initial state, and the states **True** and **False** are terminal. Figure 5.3 shows the top level of the formal specification of the AID state machine. The machine receives messages,

```
state := Cold
for ever do
    if count = 0 then
        M := receive any message
    else
        M := receive only messages of type Free, Free_ACK, or Deny
        // other messages remain on the receive queue
    end if

    switch M.type:
    case Guess:
        process_guess()
    case Affirm:
        process_affirm()
    case Free:   // sender is an AID process
        process_free()
    case Free_ACK:
        process_free_ack()
    case Deny:  // unconditional
        process_deny()
    case Add_DDO:
        process_add_ddo()
    case Add_DNA:
        process_add_dna()
    end switch
end for
```

Figure 5.3: AID State Machine for P_X

and uses the type of the message to select further processing, as shown in Figures 5.4, 5.5, 5.6, 5.7, 5.8, and 5.9. The following text surrounding the figures is an informal description of what the algorithm is doing. The comments throughout the algorithm assume that X is the identity of the assumption associated with this AID process, and that the variable **my_pid** will reflect this as P_X. The variable **sender** always indicates the process identifier of the process that sent the message to P_X.

AID: Guess Message Processing: The **guess** primitive eventually returns either "true" or "false," indicating whether the final state of the AID was **True** or **False**. Thus Guess messages are requests from User processes to AID processes for the terminal state of the AID process: either **True** or **False**. If P_X is in state **Cold** or **Hot**, then the terminal state of the AID is not yet known, and so the AID process adds the sender to the $P_X.DOM$ set until the state is resolved into **True** or **False**. If P_X is in state **Maybe**, then it has been **speculatively affirmed**, which is to say that the validity of the affirmation is dependent on

```
process_guess()
    switch state:
    case Cold:
        DOM := {sender} // record the Guess
        state := Hot

    case Hot:
        DOM := DOM ∪ {sender}        // record the Guess
        // state is unchanged

    case Maybe:
        send <Replace, M.iid, A_IDO, DNA> to sender
        // tells the sender to depend on the list of AID processes
        // specified in the A_IDO set instead of X
        // state is unchanged

    case True:
        send <Replace, M.iid, ∅, ∅> to sender
        // replace X with ∅ in sender's IDO
        // state is unchanged

    case False:
        send <Rollback, M.iid> to sender
        // state is unchanged
    end switch
end process_guess
```

Figure 5.4: Guess Message Processing

all of the other AIDs in the $P_X.A\_IDO$ set (which is the set used to speculatively affirm X) also being affirmed. So P_X sends a Replace message back to the sender, telling the sender to depend on the list of AID processes specified in $P_X.A\_IDO$ set instead of X (P_X in effect "passes the buck"). Finally, if the AID process is in state **True** or **False**, then the request can be answered immediately, and so the AID process sends back a $< Replace, A, \emptyset, \emptyset >$ (replace X with $\emptyset$ in $A.IDO$) or $< Rollback, A >$ message, respectively.

AID: Affirm Message Processing: An Affirm message tentatively asserts that the assumption the AID process represents is true[3]. The assertion is tentative in that it depends on all of the other AID processes in the accompanying $M.IDO$ set also being affirmed. If the $M.IDO$ set is empty, then the AID process P_X has been **definitely affirmed**, and so proceeds directly to state **True**, and sends $< Replace, A, \emptyset, \emptyset >$ messages to all intervals A

[3]It is a user error for an Affirm message to be sent to an AID process in a state other than **Cold**, **Hot**, or **Maybe**

```
process_affirm()
    switch state:

    case Cold, Hot, Maybe:
        if M.IDO = ∅ then                    // P_X has been definitely affirmed
            for all members B ∈ DOM set do
                send <Replace, B.iid, ∅, ∅> to B.pid
            end for
            state := True
        else if M.IDO ≠ ∅ then   // P_X has been speculatively affirmed
            // and must check for free_of violations
            A_IDO := M.IDO
            if ∃ Z : Z ∈ M.IDO and Z ∈ DDO then
                // free_of violation
                for all members B ∈ DOM set do
                    send <Rollback, B.iid> to B.pid
                end for
                state := False
            else          // speculative affirm
                for all members Y ∈ M.IDO do
                    send <Free, DDO> to P_Y
                end for
                // a non-zero count value stops processing of messages
                // other than Free and Free_ACK messages
                count := size of M.IDO
                state := Maybe
                // the Affirm procedure is completed when processing
                // Free_ACK messages decrements count to 0
            end if
        end if

    case True, False:    // user error
        abort

    end switch
end process_affirm
```

Figure 5.5: Affirm Message Processing

found in the $P_X.DOM$ set. However, if the $M.IDO$ set is not empty, then the AID process P_X must do further checking. What follows describes the checking procedure to process a **speculative affirm**.

To satisfy the requirements of any **free_of** primitives that may have been executed, P_X first checks to see if any of the AIDs in $M.IDO$ are also in $P_X.DDO$; if any AID is found in both sets, then a violation of a **free_of** has occurred. P_X proceeds immediately to state **False** and sends Rollback messages to all intervals in $P_X.DOM$.

```
process_free()
    switch state:

    case True:  // True, so it can't cause a free_of violation
        send <Free_ACK> to sender
        // state is unchanged

    case False: // ignore the message, sender will be denied anyway

    case Cold, Hot, Maybe:
        if sender ∈ DNA then                    // free_of violation
            send <Deny> to sender
        else
            DDO := DDO ∪ M.DDO
            IHA := IHA ∪ {sender}
            send <Free_ACK> to sender
        end if
        // state is unchanged

    end switch
end process_free
```

Figure 5.6: Free Message Processing

AID process P_X then checks for **free_of** violations in all of the AID processes that the sender depends on by sending Free messages to all of the AID processes specified in $M.IDO$. P_X does not wait for individual responses to the Free messages, so as to prevent deadlock among AID processes exchanging Free messages. Instead, a count is kept of how many Free messages are sent, and the count is decremented back to zero as Free_ACK messages are received. In addition, the AID process stops receiving messages other than Free and Free_ACK messages, leaving others in the receive queue[4].

If some other AID process Y receives a Free message from P_X, and is in state **True**, it immediately sends back a Free_ACK message. If Y is in state **False**, it ignores the message because the sender X will eventually receive a Deny message, terminating the Free_ACK count in X. Otherwise, it checks if the sender is in $P_Y.DNA$. If it is, then a **free_of** violation has occurred, and P_Y responds by sending an unconditional Deny message to P_X. Otherwise, the accompanying DDO set from P_X is added to $P_Y.DDO$, the sender X is added to $P_Y.IHA$, and a Free_ACK message is returned to P_X.

[4]Achieved by masking the "receive message" primitive to only select messages of a specified type. If such a facility is not present in the message passing system, then the AID process receives and records unwanted messages until the count is complete.

```
process_free_ack()
    switch state:

    case Maybe:
        counter := counter − 1
        if counter = 0 then        // speculative Affirm successful
            for all members B ∈ DOM set do
                send <Replace, B.iid, A_IDO, DNA> to B.pid
            end for
            // tells them to depend on the set of assumption
            // identifiers in A_IDO instead X
            // state remains in state Maybe
        end if

    case False: // ignore the message
        // state is unchanged

    case Cold, Hot, True:        // not possible
        abort

    end switch
end process_free_ack
```

Figure 5.7: Free_ACK Message Processing

When the count of Free_ACK messages received equals the number of Free messages sent, the AID process P_X has succeeded in completing at least the tentative affirmation of its assumption, as in the speculative affirm of Section 4.2.2. As such, P_X records the IDO that the execution of the speculative **affirm** depended on in $P_X.A\_IDO$. P_X sends $< Replace, A, A\_IDO \ldots >$ messages to all intervals A in $P_X.DOM$ telling them to depend on the set of assumption identifiers specified in the accompanying IDO set instead of X. P_X remains in state **Maybe**.

If one or more of the Free messages was ignored, the recipient AID process Y must have been in state **False**. Since the interval A that affirmed X must depend on Y, then A is doomed to be rolled back, and thus X will be sent a Deny message, terminating its counting and placing X in state **False**. Refer to Section 5.3.1.3 for further details.

AID: Deny Message Processing: Unlike Affirm messages, Deny messages are always unconditional[5]. Chapter 4 specifies that more than one **affirm** or **deny** primitive applied

[5]affirm assertions are propagated speculatively, because they apply transitively, and thus a circular ring of speculative affirm executions can be discovered to be definite. deny, however, has no transitive meaning, and so propagating speculative Deny messages would serve no purpose.

```
process_deny()
    switch state:

    case Cold, Hot, Maybe:
        for all members B ∈ DOM set do
            send <Rollback, B.iid> to B.pid
        end for
        state := False

    case False: // redundant, ignore

    case True: // user error
        abort

    end switch
end process_deny
```

Figure 5.8: Deny Message Processing

to a single assumption identifier, in any combination, is a user error, and the meaning is undefined. Thus AID processes in states **True** and **False** ignore Deny messages. In all other states (**Cold**, **Hot**, and **Maybe**[6]), the AID process unconditionally proceeds to state **False**, and sends Rollback messages to all processes who's intervals appear in $P_X.DOM$.

AID: Add_DDO and Add_DNA Message Processing: Add_DDO and Add_DNA are the direct results of **free_of** primitives. In addition to the **affirm** and **deny** operations specified in Section 4.2.4, local processing of **free_of** primitives sends Add_DDO and Add_DNA messages to propagate **free_of** requirements to all of the intervals that the executing interval depends on.

These messages are processed in a very similar manner, distinguished only in the sets that they access. If the AID process P_X is in state **True**, then it cannot cause a **free_of** violation because the assumption associated with X is already known to be true. If P_X is in state **False**, then it does not matter whether X can cause a **free_of** violation, because any interval that depends on X will be rolled back. However, for the remaining states (**Cold**, **Hot**, and **Maybe**) then the A_IDO and IHA sets must be checked for **free_of** violations. If violations are found, a Deny message is sent to the AID process P_Y found in the conflicting sets. Otherwise, the specified AID process identifiers are added to the DDO

[6]AID processes in state **Maybe** do not ignore Deny messages because they may be the result of a failed speculative **affirm**.

```
process_add_ddo()
    switch state:

    case True, False: // ignore the message

    case Cold, Hot, Maybe:    // check for free_of violations
        if ∃ Y : Y ∈ M.DDO and Y ∈ A_IDO then // free_of violation
            send <Deny> to P_Y
        else
            DDO := DDO ∪ M.DDO
        end if
        // state is unchanged

    end switch
end process_add_ddo

process_add_dna()
    switch state:

    case True, False: // ignore the message

    case Cold, Hot, Maybe:    // check for free_of violations
        if ∃ Y : Y ∈ M.DNA and Y ∈ IHA then // free_of violation
            send <Deny> to P_Y
        else
            DNA := DNA ∪ M.DNA
        // state is unchanged

    end switch
end process_add_dna
```

Figure 5.9: Add_DDO and Add_DNA Message Processing

and DNA sets, respectively. Add_DDO and Add_DNA messages do not directly affect the state of the **state** variable, although the sending of Deny messages may eventually result in a Deny message being sent to P_X.

5.3.1.2 Control Message Processing in User Processes

The operations of AID processes mandate making changes to the execution history of user processes that are presently executing, often on remote machines. Changes include updates to the dependency sets, finalization of intervals, and the rollback of the user processes. The updates are specified by messages that AID processes send back to user processes, as shown in Figure 5.1. The function Control in the collection of HOPE modules attached to each user process processes these messages and does the required updates.

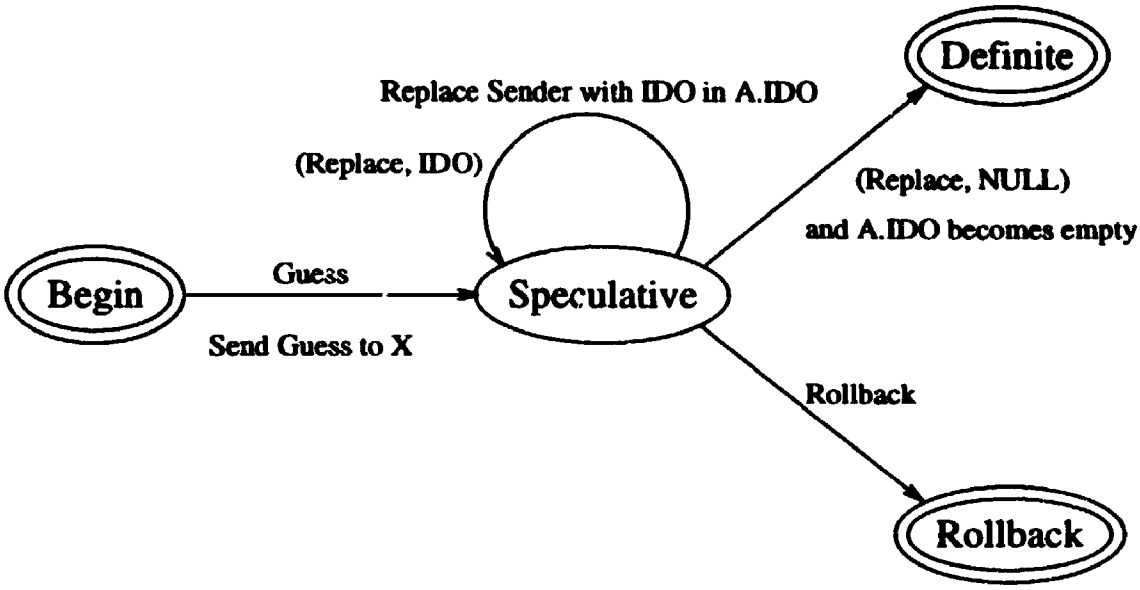

Figure 5.10: Control: Interval State Machine Diagram

Control treats the sequence of intervals in the process's history as a set of state machines. Control uses the type and parameters of the message together with the state of the specified interval to compute the required updates. The state of an interval is comprised of the following:

IDO	I Depend On
DDO	Don't Depend On
DNA	Do Not Affirm
IHD	I Have Denied
IHA	I Have Affirmed

Figure 5.10 shows the state machine for each interval. Figure 5.11 shows the formal definition of the control state machine, and the following text presents an informal description. As in the AID state machine, **sender** indicates the process identifier of the process that sent the message. **target** is the specific interval that the message should be applied to, as specified in *M.IDO*.

Control: Rollback Message Processing: A Rollback message causes the specified interval A, and all subsequent intervals, to be rolled back. The rollback happens regardless

```
control (message M)
    target := M.iid

    switch M.type
    case Rollback:
        if target ∈ history then
            rollback(target)
        end if

    case Replace:
        if M.IDO = ∅ then
            target.IDO := target.IDO \ {sender}
            if target.IDO = ∅ then
                finalize(target)
            end if
        else if M.IDO ≠ ∅ then
            for each Y ∈ M.IDO do
                if Y ∈ target.DDO then
                    send <Guess, target, ∅> to P_Y
                    send <Deny> to P_Y
                    // wait here for Rollback from P_Y
                else
                    target.IDO := (target.IDO ∪ {Y}) \ {sender}
                    send <Guess, target, ∅> to P_Y
                end if
            end for
        end if
    end switch
end control
```

Figure 5.11: Control State Machine

of the state of A, so long as A has not already been rolled back. The process is rolled back to the state immediately preceding the beginning of interval A.

Control: Replace Message Processing: A Replace message has more complex implications. Replace indicates that the sending AID process P_X should be removed from the IDO set of the specified interval A, and replaced with the accompanying $M.IDO$ set. If the resultant $A.IDO$ set is empty, then interval A is finalized. All AID processes listed in $A.IHD$ are sent Deny messages. Interval A is then marked as "definite". However, if any member Y of $M.IDO$ is also in $A.DDO$, then a **free_of** violation has occurred: Y is denied,[7] and the process waits for the inevitable Rollback message from P_Y.[8]

[7] A Guess message is first sent to Y so that A will be included in the set of intervals that Y rolls back.

[8] The user process does not actually have to wait here, but all computations from here on will necessarily

```
finalize (interval A)
    remove the checkpoint of the process state created when A was started
    mark A as "definite" in the process history
    for all members Y ∈ A.IHA do
        send <Affirm, A, ∅> to P_Y                    // unconditional Affirm
    end for
    for all members Y ∈ A.IHD do
        send <Deny> to P_Y
    end for
end finalize

rollback (interval A)
    for all members Y ∈ A.IHA do
        send <Deny> to P_Y
    end for
    roll back the process to the state checkpointed at the
        beginning of interval A
    truncate the process history just prior to A
    return False to the guess primitive that initiated interval A
end rollback
```

Figure 5.12: Interval Management Functions

More complex results occur if the interval is not finalized. Section 4.2.2 specifies that speculative execution of **affirm(X)** in interval A should add all intervals listed in $X.DOM$ to the DOM set of all assumption identifiers listed in $A.IDO$. The AID process initiated this addition to the DOM sets by sending the Replace message to all dependent intervals. Now the Control function completes the DOM addition by sending Guess messages to all of the new assumption identifiers in the replacement IDO set. Each guess message adds interval A to the DOM set of the recipient AID process.

5.3.1.3 Interval Management

The procedure in Figure 5.11 applies **finalize** and **rollback** functions to specified intervals in a user process history. Finalize applied to an interval A causes A to become definite, and makes appropriate updates to AID processes that were the subject of speculative HOPE primitives within interval A. Rollback applied to an interval A similarly rolls back the interval A, and applies updates to AID processes that were the subject of speculative HOPE primitives within interval A. Figure 5.12 presents the algorithms for finalize and rollback.

be rolled back. Waiting instead of proceeding with further useless computation produces the same results and consumes less CPU time.

5.3.1.4 Satisfying Theorem 4.4

We now show that Algorithm 5.3.1 satisfies Theorem 4.4 under certain circumstances. To specify the circumstances under which Algorithm 5.3.1 functions, it is necessary to first define an abstract representation of the dependencies that form between intervals and assumption identifier , and between assumption identifiers. We call this representation a **dependency graph.**

We begin by defining dependence:

Definition 5.8 *An interval A is said to* **depend on** *an assumption identifier Y in any instance in which $Y \in A.IDO$.*

Definition 5.9 *An assumption identifier X is said to* **depend on** *an assumption identifier Y in any instance in which $Y \in P_X.A\_IDO$.*

Note that assumption identifiers are added to the A_IDO set as a result of speculative execution of **affirm** primitives. Thus if $Y \in A.IDO$ when a process P in interval A executes **affirm(X)**, then Y will be added to $P_X.A\_IDO$ along with the rest of $A.IDO$.

Definition 5.10 *A dependency graph is a graph-theoretic representation of the current state of the dependencies between intervals and assumption identifiers, and the dependencies between assumption identifiers. The nodes of the graph are intervals and assumption identifiers and the directed edges represent the "depends on" relation. An interval A which depends on an assumption identifier Y is denoted by the edge $A \rightarrow Y$, and an assumption identifier X which depends on another assumption identifier Y is denoted $X \rightarrow Y$.*

We now apply dependency graphs to an example. Recall the example from Section 5.2 in which interval A depends on assumption identifier Y, and interval B depends on assumption identifier X; **affirm(X)** is executed in A and **affirm(Y)** is executed in interval B. Figure 5.13 illustrates the sequence of the dependency graphs in the non-interleaved case where **affirm(X)** in interval A is executed first. The speculative **affirm** of X in A while A depends on Y adds Y to $P_X.A\_IDO$, and so the new dependency graph representing this state includes the new dependency edge indicating that X depends on Y.

Figure 5.14 shows the sequence of the dependency graphs in the case where the execution of **affirm** primitives in intervals A and B are interleaved and interfere. The speculative **affirm** of X in A while A depends on Y introduces a dependency from X to Y, as before.

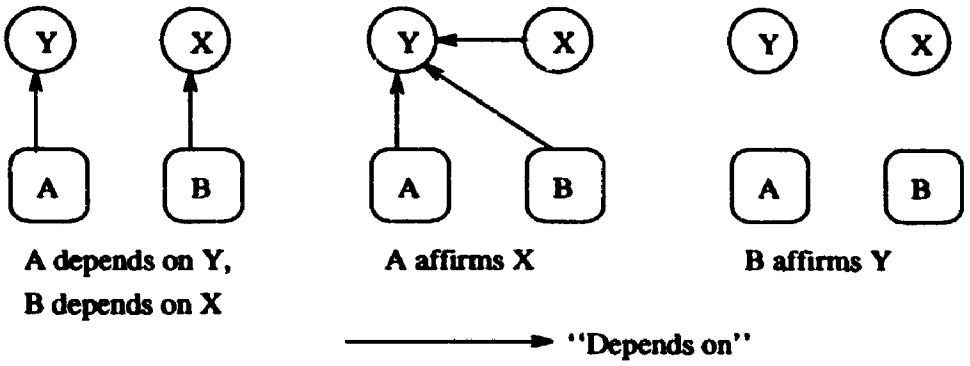

Figure 5.13: Non-interleaved Affirm Dependency Graph

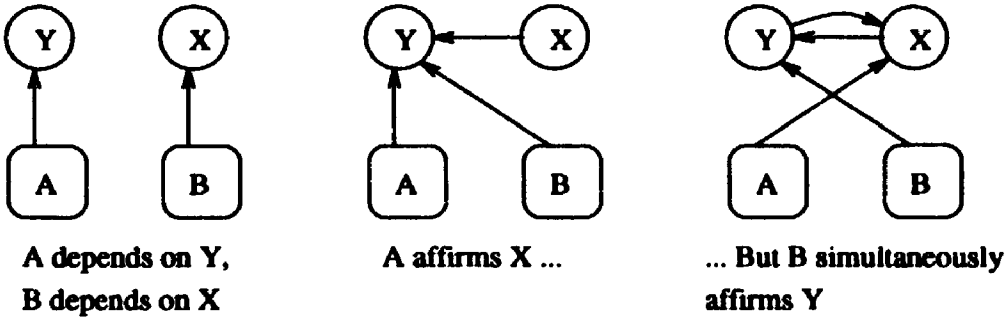

Figure 5.14: Interleaved Affirm Dependency Graphs: Interference

However, the simultaneous speculative affirm of Y in B while B depends on X also introduces a dependency from Y to X. Thus a **cyclic dependency** is formed between Y and X.

Definition 5.11 *A* **cyclic dependency** *is a cycle within a dependency graph.*

Cyclic dependencies can occur in rings of any size, from one (the self affirm described in Section 4.2.2) on upwards, depending on the number of assumption identifiers and intervals involved.

We now show that Algorithm 5.3.1 satisfies Theorem 4.4 as long as the dependency graphs formed by the execution of the program are always acyclic. Lemma 5.1 first shows that Algorithm 5.3.1 detects and corrects for conflicts between concurrent executions of **affirm** primitives as long as the resultant dependency graphs are acyclic. Lemma 5.2 then similarly shows that Algorithm 5.3.1 detects and corrects for conflicts between concurrent executions of **affirm** and **guess** primitives.

Lemmas 5.3 and 5.4 that follow are very similar to Lemmas 4.2 and 4.3 in Chapter 4. Lemma 5.3 shows that speculative execution of an **affirm** in an interval that is subsequently finalized has the same effect as definite execution of the **affirm** primitive. Lemma 5.4 shows that if all assumption identifiers that an interval A depends on are definitely affirmed, then A will be finalized.

Finally, Theorem 5.1 shows that Algorithm 5.3.1 satisfies 4.4 as long as the dependency graphs formed by the execution of the program are always acyclic.

Lemma 5.1 *For any two conflicting executions of* **affirm** *primitives, either:*

1. *The conflicting operations commute to produce the same result,*
2. *Algorithm 5.3.1 detects and corrects for the conflict, or*
3. *A cyclic dependency is formed.*

Proof: The proof proceeds by construction. First the set of atomic read and write operations resulting from **affirm** executions is identified. It is then shown that all of the potential conflicts between these read and write operations satisfy one of the three conditions in the lemma.

A process P in interval C executing **affirm(X)** results in the following atomic operations:

$C.\alpha$	Read $C.IDO$.
$C.\beta$	Read $P_X.DOM$.
$C.\gamma_i$	$\forall Y \in C.IDO$: $P_Y.DOM \leftarrow P_Y.DOM \cup \{B\}$, for some interval $B \in P_X.DOM$. This atomic read-modify-write is labelled $C.\gamma_i$.
$C.\delta_i$	$\forall B \in P_X.DOM$: $B.IDO \leftarrow (B.IDO \cup C.IDO) \backslash \{X\}$. This atomic read-modify-write is labelled $C.\delta_i$.

i and j are only used as indexes into the for-all expressions that produce the γ and δ operations. There is no relationship between the size of any of $C.IDO$, $D.IDO$, $C.DOM$, and $D.DOM$.

Inspection of Algorithm 5.3.1 shows that these are all of the read and write accesses to the dependency sets resulting from execution of **affirm** primitives. The operations occur in the following places:

$C.\alpha$	Executing **affirm(X)** atomically reads $C.IDO$ and sends it along with the affirm message to X.

$C.\beta$ P_X atomically reads $P_X.DOM$ in Figure 5.7.

$C.\gamma_i$ Control responds to $< Replace, X, A.IDO(= C.IDO) \ldots >$ messages by sending $< Guess \ldots >$ messages to each AID process in $C.IDO$. Each $P_Y \in C.IDO$ responds to the $< Guess \ldots >$ message by atomically updating $P_Y.DOM$ as shown in Figure 5.4.

$C.\delta_i$ Control responds to $< Replace, X, A.IDO(= C.IDO) \ldots >$ messages by atomically updating $C.IDO$ as shown in Figure 5.11.

These operations are all atomic because they all access data items that are local to the process applying the operation, and each process has exclusive access to its own data. Since the process applying each of these operations does not admit any other update requests in the midst of the operation, the operation cannot be subject to interleaving by any operation that will interfere. Thus each of these operations is atomic.

The order in which these atomic operations are applied within a single **affirm** execution is such that first the α and β read operations are applied, and then all of the γ_i and δ_i are applied concurrently.

A conflict has been defined as two operations operating upon the same data item, where at least one of them is a write. Only γ_i and δ_i write to data items. Since γ_i and δ_i are the only write operations, all potential conflicts must involve either γ_i or δ_i. Thus the complete list of conflicts consists of the following:

- γ_i can conflict with all operations accessing a DOM set: β and γ_j from another concurrent execution of **affirm**
- δ_i can conflict with all operations accessing an IDO set: α and δ_j from another concurrent execution of **affirm**

Thus the potential conflicts between a process P in interval C executing **affirm(X)** and a process Q in interval D concurrently executing **affirm(Z)** are as follows:

1. $C.\alpha$ vs. $D.\delta_j$
2. $C.\beta$ vs. $D.\gamma_j$
3. $C.\gamma_i$ vs. $D.\gamma_j$
4. $C.\delta_i$ vs. $D.\delta_j$

C and D can be exchanged in the above conflicts to produce a symmetric set of conflicts. Since these operations are duals, without loss of generality we consider only the above conflicts.

If process P is executing a definite (rather than speculative) **affirm**, then $C.IDO$ is null. It thus follows that $C.\alpha$ cannot conflict with any operation, and that because $C.IDO$ is empty, there will not be any $C.\gamma_i$ operations. Similarly, if process Q is executing a definite **affirm**, then $D.IDO$ is null and thus the $D.\alpha$ and $D.\gamma_i$ operations cannot produce conflicts. Thus the set of potential conflicts produced by one or more definite executions of **affirm** primitives is a strict subset of the set of potential conflicts produced when all **affirm** primitives are speculatively executed.

The results of each type of conflict are as follows:

1. $C.\alpha$ vs. $D.\delta_j$. The operations are:

$$C.\alpha \ : \ \text{Read } C.IDO$$

$$D.\delta_j \ : \ B.IDO \leftarrow (B.IDO \cup D.IDO)\backslash\{Z\} \text{ where } B \in P_Z.DOM$$

If $B = C$, then these operations conflict, producing the following operations:

$$C.\alpha \ : \ \text{Read } C.IDO$$

$$D.\delta_j \ : \ C.IDO \leftarrow (C.IDO \cup D.IDO)\backslash\{Z\} \text{ where } C \in P_Z.DOM$$

Applying $D.\delta_j$ followed $C.\alpha$ is consistent with a serialization in which **affirm(Z)** is followed by **affirm(X)**. $D.\delta_j$ is the last time that the execution of **affirm(Z)** will reference $B.IDO$. Thus if $D.\delta_j$ is applied first, then all conflicts with respect to $C.IDO$ are consistent with a serialization in which **affirm(Z)** is followed by **affirm(X)**. The effect of such a serialization is that $D.IDO$ is included in $C.IDO$ instead of Z when $C.\alpha$ is applied.

Applying $C.\alpha$ followed by $D.\delta_j$ can be consistent with a serialization in which **affirm(X)** is followed by **affirm(Z)**, but only if the rest of the execution of **affirm(X)** completes before any conflicting operations from **affirm(Z)** are applied. Otherwise applying $C.\alpha$ first will induce the subsequent sending of $< Replace, X, AIDO(= C.IDO) \ldots >$ messages[9] to all of the processes containing

[9] Or possibly $< Rollback \ldots >$ messages if the Free procedure failed and produced a $< Deny >$ message to X.

intervals listed in $P_X.DOM$ (see Figure 5.7). If $Z \in C.IDO$, then this will result in Control sending in $< Guess, \ldots >$ messages from each process in $P_X.DOM$ to P_Z (see Figure 5.11).

The application of $D.\delta_j$ implies that P_Z is necessarily in state **Maybe**, **True**, or **False**, so P_Z will respond to the $< Guess, \ldots >$ messages with $< Replace, C, A\_IDO(= D.IDO) \ldots >$ messages[10] (see Figure 5.4) that post-hoc replace Z with $D.IDO$ in $C.IDO$.

This Replace message effectively applies $D.\delta_j$ before $C.\alpha$: P_Z is supplying C with the message that it would have received had $D.\delta_j$ been applied first. If $D.\delta_j$ is applied first, then $D.IDO$ is included in $C.IDO$ instead of Z. If $C.\alpha$ is applied first, then Z is included in $C.IDO$ instead of $D.IDO$; however, the post hoc Replace message that P_Z supplies to C replaces Z with $D.IDO$ in $C.IDO$, exactly as if $D.\delta_j$ had been applied first. This is consistent with a serialization of **affirm(Z)** followed by **affirm(X)**. Thus Algorithm 5.3.1 satisfies condition 2 of the lemma.

2. $C.\beta$ vs. $D.\gamma_j$. The operations are as follows:

$$C.\beta \ : \ \text{Read } P_X.DOM$$

$$D.\gamma_j \ : \ P_Y.DOM \leftarrow P_Y.DOM \cup \{B\} \text{ where } Y \in D.IDO \text{ and } B \in P_Z.DOM$$

If $Y = X$ then the operations conflict, producing the following operations:

$$C.\beta \ : \ \text{Read } P_X.DOM$$

$$D.\gamma_j \ : \ P_X.DOM \leftarrow P_X.DOM \cup \{B\} \text{ where } X \in D.IDO \text{ and } B \in P_Z.DOM$$

Applying $D.\gamma_j$ followed by $C.\beta$ is consistent with a serialization in which **affirm(Z)** is followed by **affirm(X)**. $D.\gamma_j$ is the last time that the execution of **affirm(Z)** will reference $P_Y.DOM$. Thus if $D.\gamma_j$ is applied first, then all conflicts with respect to $P_Y.DOM$ are consistent with a serialization in which **affirm(Z)** is followed by **affirm(X)**. The effect of such a serialization is that $B \in P_X.DOM$ when $C.\beta$ is applied.

Applying $C.\beta$ followed by $D.\gamma_j$ can be consistent with a serialization in which **affirm(X)** is followed by **affirm(Z)**, but only if the rest of the execution of **affirm(X)**

[10] Or $< Rollback \ldots >$ messages if P_Z entered state **False** after sending the $< Replace \ldots >$ message.

completes before any conflicting operations from **affirm(Z)** are applied. Otherwise applying $C.\beta$ first will result in the subsequent sending of $<$ *Replace* ... $>$ messages[11] to all of the processes containing intervals listed in $P_X.DOM$ (see Figure 5.7). However, since $D \notin P_X.DOM$[12], the process Q containing interval D will not receive such a message.

The application of $C.\beta$ implies that P_X will necessarily be in state **True**, **False**, or **Maybe**, so the subsequent application of $D.\gamma_j$ will simply produce an additional $<$ *Replace* ... $>$[13] (see Figure 5.4) to the process containing interval B.

This Replace message effectively applies $D.\gamma_j$ first: P_X is supplying D with the message that it would have received had $D.\gamma_j$ been applied first. If $D.\gamma_j$ is applied first, then B is included in $P_X.DOM$, and thus B will receive a Replace message along with the rest of $P_X.DOM$. If $C.\beta$ is applied first, then B is not included in $P_X.DOM$; however, when $D.\gamma_j$ is subsequently applied (in the form of a Guess message to P_X), P_X responds by sending the same Replace message to interval B that would have been sent had B been included in $P_X.DOM$. This is consistent with a serialization of **affirm(X)** followed by **affirm(Z)**. Thus Algorithm 5.3.1 satisfies condition 2 of the lemma.

3. $C.\gamma_i$ vs. $D.\gamma_j$. The operations are as follows:

$$C.\gamma_i \ : \ P_Y.DOM \leftarrow P_Y.DOM \cup \{A\} \text{ where } Y \in C.IDO \text{ and } A \in P_X.DOM$$

$$D.\gamma_j \ : \ P_W.DOM \leftarrow P_W.DOM \cup \{B\} \text{ where } W \in D.IDO \text{ and } B \in P_Z.DOM$$

If $W = Y$ then these operations conflict, producing the following:

$$C.\gamma_i \ : \ P_Y.DOM \leftarrow P_Y.DOM \cup \{A\} \text{ where } Y \in C.IDO \text{ and } A \in P_X.DOM$$

$$D.\gamma_j \ : \ P_Y.DOM \leftarrow P_Y.DOM \cup \{B\} \text{ where } Y \in D.IDO \text{ and } B \in P_Z.DOM$$

However, any two such operations applied to $P_Y.DOM$ will commute to produce the same result in either order of application, satisfying condition 1 of the lemma.

---

[11] Or possibly $<$ *Rollback* ... $>$ messages if the Free procedure failed and produced a $<$ *Deny* $>$ message to X.

[12] If $D \in P_X.DOM$ then $D.\gamma_j$ has already been applied.

[13] Or a $<$ *Rollback* ... $>$ message.

4. $C.\delta_i$ vs. $D.\delta_j$. The operations are as follows:

$$C.\delta_i \ : \ A.IDO \leftarrow (A.IDO \cup C.IDO) \backslash \{X\} \text{ where } A \in P_X.DOM$$
$$D.\delta_j \ : \ B.IDC \leftarrow (B.IDO \cup D.IDO) \backslash \{Z\} \text{ where } B \in P_Z.DOM$$

If $A = B$ these operations conflict, producing the following:

$$C.\delta_i \ : \ A.IDO \leftarrow (A.IDO \cup C.IDO) \backslash \{X\} \text{ where } A \in P_X.DOM$$
$$D.\delta_j \ : \ A.IDO \leftarrow (A.IDO \cup D.IDO) \backslash \{Z\} \text{ where } A \in P_Z.DOM$$

A correct serialization of the execution of **affirm(X)** followed by **affirm(Z)** will produce the following effect:

$$A.IDO \leftarrow (((A.IDO \cup C.IDO) \backslash \{X\}) \cup D.IDO) \backslash \{Z\}$$

However, because the execution is serialized, the removal of X from $A.IDO$ will also be applied to $D.IDO$, and thus we are assured that the term $\cup D.IDO$ will not re-insert X. Thus the effect is:

$$A.IDO \leftarrow (A.IDO \cup C.IDO \cup D.IDO) \backslash \{X, Z\}$$

A serialization of **affirm(Z)** followed by **affirm(X)** symmetrically produces the same effect.

Conflict breaks down into four cases:

$X \notin D.IDO \wedge$ $Z \notin C.IDO$	These operations commute to produce the same result, satisfying condition 1 of the lemma.
$X \in D.IDO \wedge$ $Z \in C.IDO$	This is a cyclic dependency. δ corresponds to a $< Replace \ldots, A\_IDO, \ldots >$ message, which is produced only after A_IDO has been set by the operations in Figure 5.5. $X \in D.IDO$ thus implies that $X \in P_Z.A\_IDO$ and Z depends on X. Similarly, $Z \in C.IDO$ implies that $Z \in P_X.A\_IDO$ and X depends on Z. Thus X and Z form a cyclic dependency, satisfying condition 3 of the lemma.

$X \notin D.IDO \wedge$ $Z \in C.IDO$	Applying $C.\delta_i$ followed by $D.\delta_j$ is consistent with a serialization of **affirm(X)** followed by **affirm(Z)**. The purpose of $D.\delta_j$ is to remove Z from $A.IDO$ and replace it with $D.IDO$. Since $Z \in C.IDO$, applying $C.\delta_i$ first will add Z to $A.IDO$, but the subsequent application of $D.\delta_j$ ensures that Z will be removed again. Applying $D.\delta_j$ first, however, generates conflict: $D.\delta_j$ removes Z from $A.IDO$, but because $Z \in C.IDO$, the subsequent application of $C.\delta_i$ re-inserts Z back into $A.IDO$. When Control in Figure 5.11 applies $C.\delta_i$, it also sends a $< Guess \ldots >$ message to Z. Prior application of $D.\delta_j$ ensures that P_Z is necessarily in state **True**, **False**, or **Maybe**, and thus when the $< Guess \ldots >$ message arrives, P_Z detects the conflict and responds by sending a $< Replace, A, \ldots >$ message[14] to the process containing interval A. The Replace message again replaces Z with $D.IDO$ in $A.IDO$. Thus the effect is the same as: $$A.IDO \leftarrow (A.IDO \cup C.IDO \cup D.IDO) \backslash \{X, Z\}$$ This is consistent with a serialization of **affirm(X)** is followed by **affirm(Z)**. Thus Algorithm 5.3.1 satisfies condition 2 of the lemma.
$X \in D.IDO \wedge$ $Z \notin C.IDO$	Symmetric dual of previous case.

Thus for all possible conflicts, either condition 1, 2, or 3 from the lemma holds. □

Lemma 5.2 *For any conflicting concurrent execution of an* **affirm** *primitive and a* **guess** *primitive, either:*

1. *The conflicting operations commute to produce the same result, or*
2. *Algorithm 5.3.1 detects and corrects for the conflict.*

Proof: As before, a process P in interval C executing **affirm(X)** results in the atomic

[14] Or $< Rollback \ldots >$ message if Z is in state **False**.

operations $C.\alpha$, $C.\beta$, $C.\gamma_i$, and $C.\delta_i$. A process Q in interval D executing **guess(Z)** creates the new interval E through the following atomic operations:

$E.\zeta$ Create interval E, $E.IDO \leftarrow D.IDO \cup \{Z\}$ (local primitive processing applying Equation 4.3).

$E.\eta$ $P_Z.DOM \leftarrow P_Z.DOM \cup \{E\}$ (shown in Figure 5.4).

Since $E.\zeta$ atomically creates the interval E and initializes $E.IDO$, no conflicts can occur with respect to $E.IDO$. Similarly, because $D.IDO$ is local to the same process as $E.IDO$, and the initialization of $E.IDO$ is atomic, no conflicts can occur with respect to $D.IDO$ in $E.\zeta$ either. Thus $E.\zeta$ is free of conflicts. Thus the operations in E have the following potential conflicts:

- η can conflict with all operations accessing a DOM set: β and γ_j from a concurrent execution of **affirm**

Thus the potential conflicts between a process P in interval C executing **affirm(X)** and a process Q in interval D concurrently executing **guess(Z)** are as follows:

1. $E.\eta$ vs. $C.\beta$
2. $E.\eta$ vs. $C.\gamma_i$

C and E can be exchanged in the above conflicts to produce a symmetric set of conflicts. Since these operations are duals, without loss of generality we consider only the above conflicts.

If process P is executing a definite (rather than speculative) **affirm**, then C will not exist, and thus $C.IDO$ is null. It thus follows that because $C.IDO$ is empty, there will not be any $C.\gamma_i$ operations. Thus the set of potential conflicts produced by the definite execution of an **affirm** primitive is a subset of those produced by speculative execution of an **affirm** primitive.

Similarly, if process Q is definite when executing **guess(Z)**, then D will not exist, and thus the $E.IDO = \{Z\}$ and $E.\zeta$ cannot produce any conflicts. However, since *zeta* is free of conflicts in any case, speculative vs. definite **guess** operations have the same set of potential conflicts.

The results of each type of conflict are as follows:

1. $E.\eta$ vs $C.\beta$. The operations are as follows:

$$E.\eta \ : \ P_Z.DOM \leftarrow P_Z.DOM \cup \{E\}$$

$$C.\beta \ : \ \text{Read } P_X.DOM$$

If $X = Z$ then these operations conflict. Applying $E.\eta$ followed $C.\beta$ is consistent with a serialization in which **guess(Z)** is followed by **affirm(Z)**.

Applying $C.\beta$ followed by $E.\eta$ means that $P_Z.DOM$ does not include E, so when Algorithm 5.3.1 sends $< \textit{Replace} \ldots >$ or $< \textit{Rollback} \ldots >$ messages to all processes possessing intervals in $P_Z.DOM$, E is not included. However, the application of $C.\beta$ means that P_Z will necessarily be in state **True**, **False**, or **Maybe**. Thus when the $< \textit{Guess}, E >$ message (the application of $E.\eta$) is processed by P_Z, P_Z responds with a $< \textit{Replace} \ldots >$ or $< \textit{Rollback} \ldots >$ message (see Figure 5.4), maintaining consistency with a serialization of **affirm(Z)** followed by **guess(Z)**. Thus Algorithm 5.3.1 satisfies condition 2 of the lemma.

2. $E.\eta$ vs. $C.\gamma_i$. The operations are as follows:

$$E.\eta \ : \ P_Z.DOM \leftarrow P_Z.DOM \cup \{E\}$$

$$C.\gamma_i \ : \ P_Y.DOM \leftarrow P_Y.DOM \cup \{A\} \text{ where } Y \in C.IDO \text{ and } A \in P_X.DOM$$

If $Y = Z$ then these operations conflict, producing the following:

$$E.\eta \ : \ P_Z.DOM \leftarrow P_Z.DOM \cup \{E\}$$

$$C.\gamma_i \ : \ P_Z.DOM \leftarrow P_Z.DOM \cup \{A\} \text{ where } Z \in C.IDO \text{ and } A \in P_X.DOM$$

However, any two such operations applied to $P_Z.DOM$ will commute to produce the same result in either order of application, satisfying condition 1 of the lemma.

Thus Algorithm 5.3.1 detects and corrects for all possible conflicts resulting from concurrent execution of **guess** and **affirm** primitives. □

The following lemma, similar to Lemma 4.2, shows that speculative execution of an **affirm** primitive followed by the speculative interval being finalized is equivalent to definite execution of the **affirm**.

Lemma 5.3 Affirm Transitivity: *Let B be an interval in process Q that depends on an assumption identifier X, i.e., $X \in B.IDO$, and let all dependency graphs produced by this execution be acyclic. The effect of executing* **affirm(X)** *within a speculative interval A upon $B.IDO$ and the state of P_X, followed by A eventually being finalized, is the same as the effect of definite execution of* **affirm(X)**.

Proof: Definite execution of **affirm(X)** will place P_X in state **True** and send a $< Replace, B, \emptyset, \ldots >$ message to process Q. Thus the impact is to remove X from $B.IDO$ and to place P_X in state **True**.

Let A be a speculative interval that executes **affirm(X)** for some $X \in B.IDO$. Speculative execution of **affirm(X)** in interval A will place P_X in state **Maybe**, and set $P_X.A\_IDO \leftarrow A.IDO$ (see Figures 5.5 and 5.7). Since $B \in P_X.DOM$, P_X will send a $< Replace, B, A\_IDO(= A.IDO), \ldots >$ to process Q. Figure 5.11 shows that Control will use this message to replace X with $A.IDO$ in $B.IDO$, and will send $< Guess \ldots >$ messages to all AID processes in $A.IDO$. The $< Guess \ldots >$ messages will in turn add B to the DOM set attached to each AID process in $A.IDO$, ensuring that B is in the DOM set of all AID processes that also contain A. Let $\beta = B.IDO$ after the Replace message is processed by Control in Q, and let $\alpha = A.IDO$ be the set of assumption identifiers that replace X.

By assumption, interval A is subsequently finalized. Since Control in Figure 5.11 will only finalize A if $A.IDO$ is null, we know that all AID processes in $A.IDO$ have entered state **True**, and thus been replaced with $\emptyset$. Since B is in all of the DOM sets that contain A, all Replace messages sent to A will also be sent to B. Lemma 5.1 and the acyclic assumption assure that any concurrency conflicts between Replace messages are detected and corrected. All replacements made in $A.IDO$ will also be made in $B.IDO$, so the replacements that reduced $A.IDO$ to $\emptyset$ will remove all of α from $B.IDO$. Thus X has effectively been removed from $B.IDO$.

Since A has been finalized, an $< Affirm, A, \emptyset >$ message is sent to X, as shown in Figure 5.12, placing X in state **True**, as shown in Figure 5.5. Thus the effect is the same as definite execution of **affirm(X)**. □

Lemma 5.4 *For any interval B, if* **affirm(X)** *is definitely executed on all assumption identifiers $X \in B.IDO$, then* **finalize(B)** *will result.*

Proof: The definite execution of **affirm(X)** for each $X \in B.IDO$ will result in $< Affirm, \ldots, \emptyset >$ messages being sent to each P_X. Figure 5.5 shows that these Affirm messages will place each P_X in state **True**, and send $< Replace, \ldots, \emptyset, \ldots >$ messages to each interval in $P_X.DOM$.

Each X was added to $B.IDO$ either by the execution of a **guess** primitive, or the processing of a Replace message. In both cases, a corresponding $< Guess \ldots >$ message is sent to the associated AID process P_X to add B to $P_X.DOM$. Lemma 5.2 assures that concurrency conflicts between Replace and Guess messages are detected and corrected. Thus for each $X \in B.IDO$, it is also the case that $B \in P_X.DOM$.

Since each AID process P_X sent a $< Replace, \ldots, \emptyset, \ldots >$ message to each interval in its DOM set, such a Replace message has necessarily been applied it to $B.IDO$. Thus all assumption identifiers listed in $B.IDO$ have been replaced with $\emptyset$, reducing $B.IDO$ to $\emptyset$, and inducing Control to finalize B as shown in Figure 5.11. □

Theorem 5.1 *For any HOPE program execution in which the dependency graph is always acyclic: for all intervals, say B, Algorithm 5.3.1 executes* **finalize(B)** *if and only if* **affirm(X)** *is applied to all of the assumption identifiers $X \in B.IDO$ by intervals that are eventually finalized.*

Proof: First we show that if **affirm(X)** is applied to all assumption identifiers $X \in B.IDO$ by intervals that eventually become definite, then **finalize(B)** will result.

Lemma 5.4 shows that if all of the **affirm(X)** primitives are executed by definite intervals, then **finalize(B)** will result. Lemma 5.3 shows that speculative execution of **affirm** primitives in intervals that are eventually finalized has the same effect as definite execution of **affirm** primitives, and so **finalize(B)** will result in either case.

We now show that if **finalize(B)** implies that **affirm(X)** has been applied to all assumption identifiers $X \in B.IDO$ using proof by contradiction. Assume that B has been finalized, and that there exists an assumption identifier $X \in B.IDO$ that has not been affirmed.

Since X has not been affirmed, no operation will have removed X from $B.IDO$. Therefore $B.IDO \neq \emptyset$, preventing Control in Figure 5.11 from finalizing B. This contradicts the assumption that B has been finalized.

Therefore, for all intervals B, B is finalized if and only if all of the assumption identifiers $X \in B.IDO$ are affirmed in intervals that eventually become definite. □

5.3.2 Cycle Detection

Algorithm 5.3.1 will will satisfy Theorem 4.4, so long as the dependency graph as defined in Section 5.2 remains acyclic. However, if the dependency graph becomes cyclic, as is the case when mutual **affirm** primitives are executed simultaneously as illustrated in Figure 5.14, then Algorithm 5.3.1 will fail to detect the cycle, and the participating intervals will "bounce" their way around the cyclic ring of dependent AID processes forever.

To specifically illustrate that Algorithm 5.3.1 can fail in the presence of cyclic dependencies, we consider the potential conflicts between a process P in interval C executing **affirm(X)** and a process Q in interval D concurrently executing **affirm(Z)**, as in Lemma 5.1. In particular, consider conflict case 4 between $C.\delta_i$ and $D.\delta_j$. The operations are as follows:

$$C.\delta_i \ : \ A.IDO \leftarrow (A.IDO \cup C.IDO) \backslash \{X\} \text{ where } A \in P_X.DOM$$
$$D.\delta_j \ : \ B.IDO \leftarrow (B.IDO \cup D.IDO) \backslash \{Z\} \text{ where } B \in P_Z.DOM$$

If $A = B$ these operations conflict, producing the following:

$$C.\delta_i \ : \ A.IDO \leftarrow (A.IDO \cup C.IDO) \backslash \{X\} \text{ where } A \in P_X.DOM$$
$$D.\delta_j \ : \ A.IDO \leftarrow (A.IDO \cup D.IDO) \backslash \{Z\} \text{ where } A \in P_Z.DOM$$

If $X \in D.IDO \wedge Z \in C.IDO$, then a cyclic dependency has formed. Both P_X and P_Z are in state **Maybe**, and $P_X.A\_IDO = \{Z\}$ and $P_Z.A\_IDO = \{X\}$. When processes P and Q send $< Guess \ldots >$ messages to P_X, P_X responds with $< Replace, \ldots, A\_IDO(= \{Z\}), \ldots >$, which will cause Control to send a $< Guess \ldots >$ message to P_Z. P_Z will similarly respond with $< Replace, \ldots, A\_IDO(= \{X\}), \ldots >$. Thus intervals C and D in processes P and Q will cycle back and forth between P_X and P_Z forever.

This subsection describes extensions to the algorithms of Subsection 5.3.1 that detect such cycles and close them by removing dependencies from intervals to AID processes that have been found to be members of a cycle. We refer to this as **Algorithm 5.3.2**. Figure

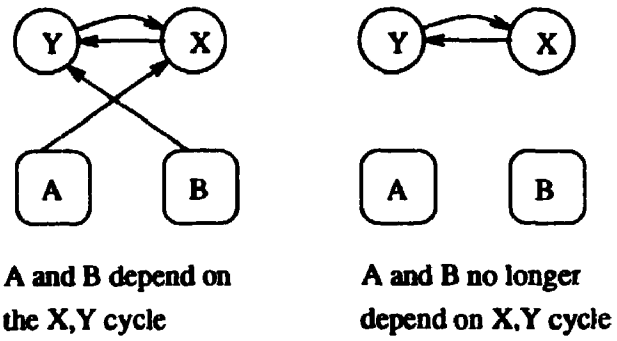

Figure 5.15: Correcting Cyclic Dependency

5.15 shows the progression of the dependency graph starting from the cyclic dependency state in Figure 5.14 to a state in which the intervals no longer depend on the cycle. If one or more of the intervals that executed the speculative affirms that constructed the cycle are eventually finalized as a result, they will send unconditional Affirm messages to the members of the cycle, causing them to be definitely affirmed.

The collection of dependency sets stored with each interval, and thus the state of each Control state machine, is extended as follows:

UDO Used to Depend On set of assumption identifiers that used to be in IDO, to prevent a process from cyclicly exchanging one dependency for another.

5.3.2.1 Control Message Processing in User Processes

Cycle detection does not require any changes in the AID state machine. The Control function does all of the cycle detection work in the course of processing Replace messages. When processing a Replace message, in addition to removing the sender of the Replace message from the specified interval's IDO set, the sender is also added to the UDO set. Figure 5.16 shows the formal definition of the extended Control state machine, and the following text informally describes the extended Control state machine.

If a Replace message arrives with a non-empty replacement $M.IDO$ set, then $M.IDO$ is compared against the specified interval's UDO set. If an AID in $M.IDO$ is found in the UDO set, it is discarded: This represents a cycle in the dependency loop, and once detected the interval no longer needs to depend on the cycle.

Here we show that Algorithm 5.3.2 completely satisfies Theorem 4.4 by showing that Algorithm 5.3.2 detects and removes all dependencies from intervals to assumption identifiers

```
control (message M)
    target := M.iid

    switch M.type
    case Rollback:
        if target ∈ history then
            rollback(target)
        end if

    case Replace:
        if M.IDO = ∅ then
            target.IDO := target.IDO \ {sender}
            if target.IDO = ∅ then
                finalize(target)
            end if
        else if M.IDO ≠ ∅ then
            for each Y ∈ M.IDO do
                if Y ∈ target.DDO then
                    send <Guess, target, ∅> to P_Y
                    send <Deny> to P_Y
                    // wait here for Rollback from P_Y
                else if Y ∈ target.UDO then
                    target.IDO := target.IDO \ {sender}
                    if target.IDO = ∅ then
                        finalize(target)
                    end if
                else
                    target.IDO := target.IDO ∪ {Y}
                    send <Guess, target, ∅> to P_Y
                end if
            end for
            target.IDO := target.IDO \ {sender}
            target.UDO := target.UDO ∪ {sender}
        end if
    end switch
end control
```

Figure 5.16: Control State Machine with Cycle Detection

that are members of a cycle.

Lemma 5.5 *All members of a cyclic dependency ring must be assumption identifier nodes that have been speculatively affirmed.*

Proof: The proof proceeds by process of elimination, eliminating all possible nodes from the graph other than those assumption identifier nodes that have been speculatively affirmed. We first observe that all nodes in a cycle must have both in-bound and out bound edges, and

then show that only speculative execution of **affirm** primitives can attach both in-bound and out-bound edges to an assumption identifier node.

An in-bound edge attached to a node X is a dependence on X, i.e., $\rightarrow X$. An out-bound edge from a node X indicates that X depends on some other node, i.e., $X \rightarrow$. Dependency graphs are composed of both assumption identifier and interval nodes. However, interval nodes only have out-bound dependency edges, because no other node can ever depend on an interval. Thus dependency cycles can only be composed of assumption identifiers.

Execution of **guess** and **affirm** primitives can add in-bound edges to assumption identifier nodes. However, the *A_IDO* set associated with each AID process defines the set of out-bound edges attached to each assumption identifier node. Only the speculative execution of **affirm(X)** will set $P_X.A\_IDO$ to a non-null value, as shown in Figure 5.5. Thus all members of a cyclic dependency must be assumption identifier nodes that have been speculatively affirmed. □

Theorem 5.2 *If a set of AID processes forms a dependency graph G that contains a cycle C, then Algorithm 5.3.2 will remove all dependencies from speculative intervals on all members of the set C.*

Proof: By inspection, we show that the processing of speculative **affirm** executions detects cycles and eliminates dependencies on such cycles.

Whenever an AID process P_X has been speculatively affirmed, it is left in state **Maybe** with an *A_IDO* set indicating the list of other AID processes that it depends on from the speculative affirm. Any interval A in a user process attempting to become dependent on such an AID process by sending it a Guess message (either in response to a user **guess** primitive or in the course of processing a Replace message) will get a $< Replace, A, A\_IDO \ldots >$ message as a result. The *A_IDO* set points to the set of AID processes that X depends on. Thus user processes that attempt to depend assumption identifiers that have been speculatively affirmed are forced to instead depend on the set of assumption identifiers that the speculatively affirmed identifiers depend on.

If an assumption identifier X has been speculatively affirmed, and is in a dependency cycle, then any interval attempting to depend on X will be forced to instead depend on the "next" assumption identifier in X's cycle, Y. Attempting to depend on Y will similarly pass

the interval on to the following assumption identifier. This can be thought of as "walking around" the ring of dependencies, as in a walk of a graph.

As a dependent interval A walks around the dependency ring, it records the list of AID processes that it has attempted to depend on in $A.UDO$. When A attempts to depend on an AID process that it has already tried to depend on, it is detected by comparison with $A.UDO$ as shown in Figure 5.16. Control responds by deleting A's dependency on the ring. □

5.3.3 Ordering Requirements Detection

The **free_of** primitive specifies a causal ordering requirement between an interval and an assumption identifier that must *not* occur. Violation of such a requirement can occur in two possible ways:

1. It is false at the time the requirement is specified.

2. It is not false at the time specified, but eventually becomes false.

The first case is detected locally by the library function for **free_of**. The second case can only occur if the executing interval participates in a cyclic ring of speculative **affirm** primitives in which the dependency among intervals and assumption identifiers is opposite to that specified in the **free_of** primitive. Therefore, we need only check for **free_of** violations when closing cycles of **affirm** primitives.

Algorithm 5.3.2 satisfies Theorem 4.4, even in the presence of cyclic dependencies. When Algorithm 5.3.2 detects a cycle in Figure 5.16, it closes the cycle by removing the dependency edges that have been discovered to be members of a cycle. Algorithm 5.3.2 will frequently detect **free_of** violations through the propagation of the DDO and DNA sets, and the checks performed on these sets. However, there are some cases in which the requirements of a **free_of** specification can leak through the checks.

Because the speculative **affirm** and **free_of** data are propagated *asynchronously* (sent without waiting for results), it is possible for the **affirm** and **free_of** propagation messages to be sent *simultaneously*, allowing them to pass one another while flowing in opposite directions around the dependency cycle. For instance, consider a circumstance in which an interval A in process P depends on X, and an interval B in process Q depends on Y. If P executes **free_of(Y); affirm(Y)** in interval A and Q concurrently executes **affirm(X)**

Type	From	To	Arguments	Meaning
P_Guess	User	AID	*iid*	Sender post hoc re-guesses that recipient is true
P_ACK	AID	User	*iid*	Acknowledge P_Guess

Table 5.2: Order Requirement Detection HOPE Messages

in B, then it is possible for the UDO cycle detection mechanism shown in Figure 5.16 in process Q to finalize interval B. Simultaneously, the DDO **free_of** detection mechanism in Figure 5.16 in process P is sending $< Deny >$ to P_X, resulting in the inconsistent view that X is both **True** and **False**. Thus, Algorithm 5.3.2 does not satisfy Theorem 4.5.

To guard against such a circumstance, we enhance the algorithms to *double check* that a cyclic dependency is uncontaminated with **free_of** assertions that would cause closure of the cycle to be invalid. Double checking means that once it is detected that an interval depends on an assumption identifier X that is a member of a cycle, instead of merely removing the dependency on X, the interval checks back with P_X to ensure that the cycle has not violated any **free_of** assertions. We refer to this as **Algorithm 5.3.3**.

Table 5.2 shows the additional messages required for double checking. The additional dependency tracking set stored with each interval, and thus in the state of each Control state machine, is as follows:

PDO Post hoc Depend On, set of assumption identifiers the interval post hoc became dependent on.

Figure 5.17 shows the top level of the formal specification of the AID enhanced state machine. As before, the machine receives messages, and uses the type of the message to select further processing. The functions used for processing the messages are unchanged from Section 5.3.1, except for the addition of Figure 5.18, which specifies the processing for P_Guess messages. The formal specification of the enhanced Control state machine is shown in Figure 5.19.

Figure 5.16 shows that Algorithm 5.3.2 discards new dependencies in the replacement IDO set that accompanies Replace messages if they are found in the UDO set of the specified interval, because it has detected a cycle and considers the cycle to be closed. To double check that the cycle does not contain any latent **free_of** requirements that would break

```
state := Cold
for ever do
    if count = 0 then
        M := receive any message
    else
        M := receive only messages of type Free, Free_ACK, or Deny
        // other messages remain on the receive queue
    end if

    switch M.type:
    case Guess:
        process_guess()
    case P_Guess:
        process_p_guess()
    case Affirm:
        process_affirm()
    case Free:  // sender is an AID process
        process_free()
    case Free_ACK:
        process_free_ack()
    case Deny:  // unconditional
        process_deny()
    case Add_DDO:
        process_add_ddo()
    case Add_DNA:
        process_add_dna()
    end switch
end for
```

Figure 5.17: AID State Machine for P_X with Free_Of Detection

the cycle, the enhanced Control function responds by sending P Guess messages to AID processes found in both the replacement IDO set and the UDO set. When a P_Guess message is sent to an AID process P_X, X is added to the PDO set associated with the interval that Control is processing, so that the X cannot be dislodged from the IDO set until the P_ACK message arrives.

An AID process that receives a P_Guess message simply verifies that the AID process has not processed a Deny message since it was first affirmed[15]. If the AID process P_X is still counting its Free messages, then the P_Guess request is recorded until the count is complete. When the count of Free messages returns to zero (or if it is already zero), P_X then sends a P_ACK message to all requesting intervals indicating that they can safely discount this

[15]It must have been affirmed before it can appear in a UDO set, which is the only case that induces a P_Guess message.

```
process_p_guess()
    switch state:
    case Maybe, True:
        send <P_ACK, M.iid> to sender
        // state is unchanged
    case False:
        send <Rollback, M.iid> to sender
        // state is unchanged
    case Cold, Hot:      // not possible
        abort
    end switch
end process_p_guess
```

Figure 5.18: P_Guess Message Processing

AID and consider it **True**. Otherwise the AID issues Rollback messages to all intervals in its DOM set.

Only P_ACK messages can remove an AID from an interval's IDO set once a P_Guess message is issued. This protocol forces the ring to be checked after it has been completely traversed. Since any **free_of** restrictions would have been propagated when the ring was initially constructed, the P_Guess procedure will necessarily discover these restrictions, and force appropriate Rollbacks.

Finally, we show that Algorithm 5.3.3 also satisfies Theorem 4.5 by showing that it detects all cycles, but only finalizes those intervals that depend on cyclic dependencies that do not conflict with any **free_of** requirements, and rolls back all others.

Theorem 5.3 *For any interval A that executes* **free_of(X)**, *then either interval A never becomes dependent on assumption identifier X or any of X's causal descendants, or Algorithm 5.3.3 rolls back interval A.*

Proof: **free_of** specifications describe a total ordering of the dependencies between intervals and assumption identifiers, and by transitivity[16], between assumption identifiers and other assumption identifiers. Violating such a specification either requires the construction of the exact opposite of such an ordering, or the construction of a cyclic dependency which reduces to a violation of the **free_of** ordering specification.

A direct violation requires that the invoking interval is already dependent on the specified AID process. Since the **free_of** primitive checks locally for such violations, we have

[16] From 4.4, if interval A depends on assumption identifier X, and A affirms Y, then by transitivity, Y depends on X.

```
control (message M)
    target := M.iid

    switch M.type
    case Rollback:
        if target ∈ history then
            rollback(target)
        end if

    case Replace:
        if M.IDO = ℓ and sender ∉ target.PDO then
            target.IDO := target.IDO \ {sender}
            if target.IDO = ∅ then
                finalize(target)
            end if
        else if M.IDO ≠ ∅ then
            for each Y ∈ M.IDO do
                if Y ∈ target.DDO then
                    send <Guess, target, ∅> to P_Y
                    send <Deny> to P_Y
                    // wait here for Rollback from P_Y
                else if Y ∈ target.UDO and sender ∉ target.PDO then
                    target.PDO := target.PDO ∪ {Y}
                    send <P_Guess, target, ∅> to P_Y
                else
                    target.IDO := target.IDO ∪ {Y}
                    send <Guess, target, ∅> to P_Y
                end if
            end for
            target.UDO := target.UDO ∪ {sender}
            if sender ∉ target.PDO then
                target.IDO := target.IDO \ {sender}
            end if

    case P_ACK:
        target.PDO := target.PDO \ {sender}
        target.IDO := target.IDO \ {sender}
        if target.IDO = ∅ then
            finalize(target)
        end if
    end switch
end control
```

Figure 5.19: Control State Machine with Free_Of Detection

satisfied the first contingency.

The total ordering requirements of a **free_of** specification, encoded in the DDO (Don't Depend On) and DNA (Do Not Affirm) sets, are propagated along the causal dependencies when speculative **affirm** primitives are executed. Cyclic speculative **affirm** rings, once detected, force the AID processes to double check for **free_of** violations in the form of P_Guess messages. Since the **free_of** specification must pass around the cyclic dependency ring, and finalization of an interval that depends on such a cycle can only happen by way of a P_Guess message, which can only happen after the cycle has been completely traversed, it is not possible for the **free_of** specification to go undetected by the ring closure mechanism. Thus an interval depending on cyclic dependency that violates a **free_of** total ordering specification cannot be successfully finalized. Thus we have satisfied the second contingency. Therefore, intervals that depend on cyclic dependencies that violate **free_of** specifications cannot be finalized, and will instead be rolled back. Therefore intervals with dependencies that violate **free_of** requirements will be rolled back. Thus Algorithm 5.3.3 detects all **free_of** ordering requirements. □

5.4 Summary

This chapter presents an algorithm for the HOPE primitives. The algorithm does not intrinsically limit optimism, in that user invocations of HOPE primitives never have to wait for results.

Not waiting for results is difficult in a distributed environment because of the potential for concurrency conflicts. We avoid both conflicts and delays by constructing an algorithm that anticipates and corrects for potential conflicts. The algorithm is constructed in a progression:

1. a simple algorithm, subject to concurrency conflicts
2. a more robust algorithm, safe from concurrency conflicts except for **free_of** requirements
3. a complete algorithm, safe from concurrency conflicts

By using such a progression, proofs are constructed that the algorithm is consistent with the important semantic implications of Theorems 4.4 and 4.5 from Chapter 4.

Chapter 6

HOPE Implementation[1]

This chapter describes the actual software that implements the HOPE prototype, using the algorithms presented in Chapter 5. Section 6.1 outlines the basic structure of the system. Section 6.2 describes the checkpoint and rollback mechanism. Finally, Section 6.3 describes future research and enhancements to be made to the prototype.

6.1 Prototype Structure

One of the primary research objectives of HOPE is to increase accessibiiity of optimistic techniques to applications programmers. Thus it was determined that HOPE should be built on top of a pre-existing message passing environment, preferably one with a large user base.

The PVM system [62] was selected as the first target environment for HOPE. PVM (Parallel Virtual Machine) is a software package that allows a network of computers to appear as a single concurrent computational resource. PVM presents a programming model of concurrent processes that communicate with asynchronous messages, reminiscent of several popular massively parallel supercomputers, such as the Intel Hypercube, Thinking Machine's CM5, and the nCUBE multiprocessor. PVM is implemented as a library of message passing and administrative functions callable from C or FORTRAN, and a pvmd router daemon. The user's processes, and the router daemons, run as ordinary processes on the host operating system (either UNIX-like computers, or several different kinds of multiprocessor computers). Tasks communicate using TCP/IP sockets between networked computers, and use the native communications mechanism on multiprocessors.

[1] Be careful what you HOPE for, you might get it.

PVM was selected as a target environment because:

- the source code was freely available
- PVM is well supported by the authors at the Oak Ridge National Laboratory and the University of Tennessee
- PVM has a broad user base, providing a potentially large audience for optimism

Figure 6.1 shows the basic structure of the implementation. The HOPE primitives are provided to user programs in the form of `HOPElib`: a library of HOPE-related functions. `HOPElib` contains functions for invoking each of the HOPE primitives, as well as the `control` function described in Chapter 5. AID processes are spawned in the course of executing the HOPE **guess** function.

Subsection 6.1.1 describes some of the key data structures used in the PVM implementation of HOPE. Subsection 6.1.2 describes the changes to PVM that allow HOPE control messages to be injected into remote executing processes, even if such processes are compute-bound. Finally, Subsection 6.1.3 describes the modifications to PVM that extend HOPE dependency tracking to user message traffic.

6.1.1 Furniture: Data Structures

The history of a user process, as described in Chapter 4, is chronicled in the user's PVM processes using a `hope_history` data structure. This data structure is a linked list of nodes, where each node stores all of the pertinent dependency set information for that interval. Each interval in the history is assigned a unique, ascending interval number

The basic structure for a HOPE dependency set is in fact a set. The basic operations that need to be performed on these sets are insertion, deletion, and search. For technical reasons, it is occasionally necessary to process all members of a set in ascending order, so the sets were implemented as doubly linked lists maintained in ascending numerical order.

The DOM set is a set of interval identifiers, represented as two integers: one for the process identifier, and one for the interval number within the process. All other sets are sets of assumption identifier references, represented as integers containing the process IDs of the assumption identifier processes.

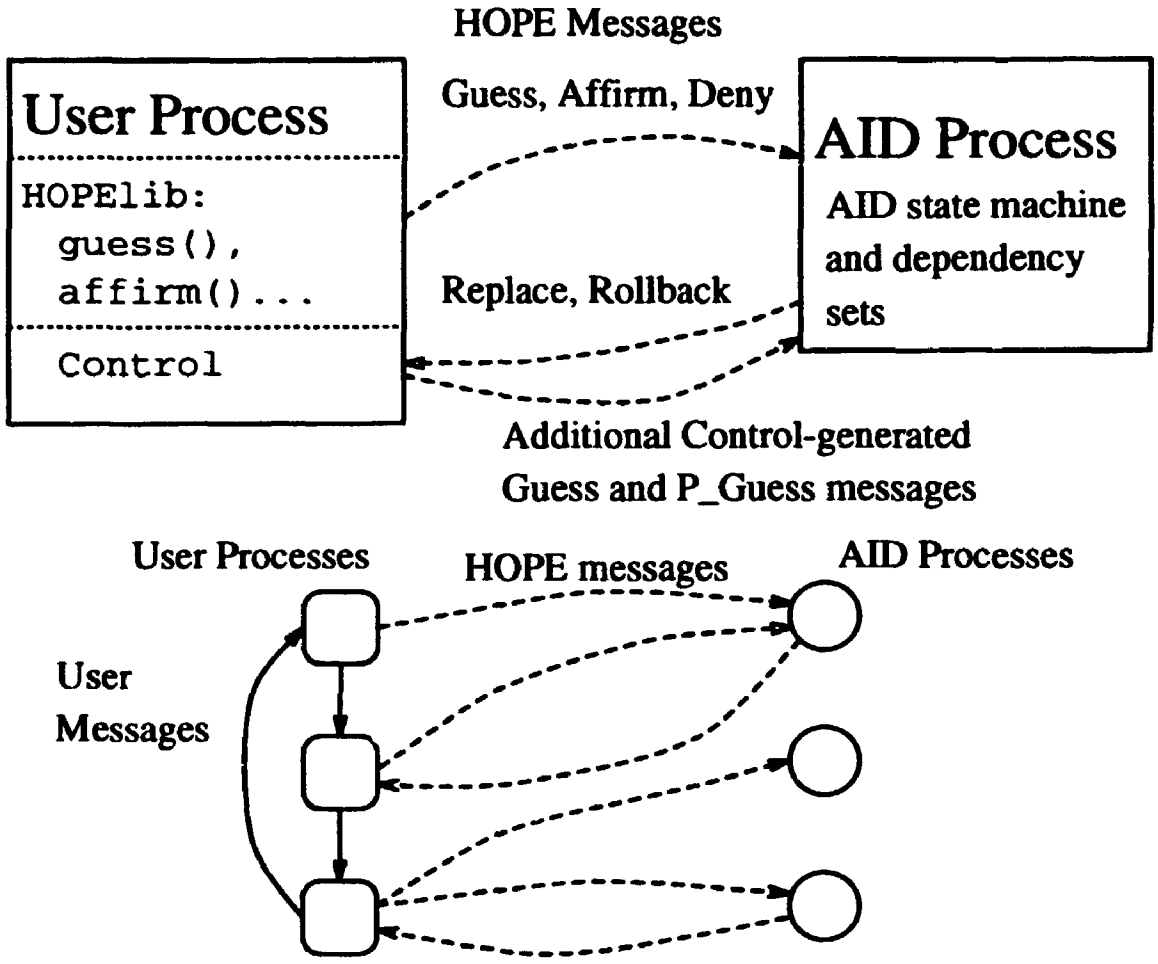

Figure 6.1: Structure of the Basic HOPE Model

6.1.2 Control Messages

When an assumption identifier process sends a message to a user process (a control message), the user process is not looking for HOPE control messages, and in fact may be compute-bound and not interacting with the PVM virtual machine at all. To force the user process to asynchronously process HOPE control messages, the assumption identifier process sends a UNIX signal to the user process (using the PVM function `pvm_sendsig`) after sending the control message. The signal is trapped by a signal handler installed in the user process by `libpvm`, which in turn receives and processes the HOPE control message. Thus, HOPE control messages can be asynchronously *injected* into user processes, even if they are compute-bound.

6.1.3 Distributed Computing: Tagged Messages

Chapter 3 describes user messages sent from speculative intervals as being "tagged" with the set of assumption identifiers that the interval depends on (the contents of the sender's IDO set). Chapter 4 omits tagged messages from the formal specification of HOPE, noting that message tagging can be provided by simply performing appropriate guesses in response to the contents of a message tag. This section describes the implementation of message tagging.

Message tagging requires modification of the message passing protocol used to exchange messages between user processes. The message sending function must be modified to include the current IDO with each message sent. Conversely, the message receiving function must be modified to look for this "tag" containing the sender's IDO set, and execute **guess** operations on each assumption identifier in the tag.

The **guess** operations applied to the tag set are not exactly the same as the user **guess** operations. The optimistic assumption being made is that the sending process will be finalized, and so the current message will continue to "exist" (i.e., be the result of a computation that has not been rolled back). The message logging and replay mechanism must be modified so that if a rollback occurs because of a tag guess, the message with the offending tag is removed from the log.

Executing a sequence of user-level **guess** primitives would perform a checkpoint of the user's process for each assumption identifier in the tag. However, there is really only one optimistic assumption being made (that the message is persistent), and so only one checkpoint is needed. If *any* one of the assumption identifiers in the tag is denied, then the same state checkpoint can be used to support the required rollback.

The **guess_tag** function provides the required functionality. The **pvm_send**, **pvm_recv** and **pvm_nrecv** functions have been modified to encode and decode the sender's IDO set as a tag attached to the message, and once decoded, invoke **guess_tag** with the entire tag set as the argument. **guess_tag** in turn performs a single checkpoint, and constructs a new interval **a** (as in the user **guess** primitive), but adds all of the tag set to **a.IDO**. **guess_tag** then sends Guess messages to each of the AID processes specified in the tag set.

6.2 Rollback

PVM tasks are UNIX processes. As such, HOPE's checkpoint and rollback requirements necessitate the ability to checkpoint and rollback a UNIX process. This section outlines the techniques used to implement checkpoint and rollback of PVM tasks, while Appendix A provides the full details.

The desired facility is a portable, user-space mechanism for checkpointing the state of a UNIX process in C, and then later rolling the process back to the state that was checkpointed. While efficiency was a concern, it was secondary to simplicity. The mechanism should roll back all program-visible aspects of the user data space. Since the mechanism is rollback instead of restart, there is no need to restore code space or kernel space state data. The state of a process comprises four principal components:

- the stack
- the heap
- static data
- the program counter

Basic checkpointing a process consists of writing the above data out to a disk file. Rolling back a process consists of reading back the data from disk into the process's memory at the same location from which it was saved, and jumping back to the stored location of the program counter.

Saving and restoring a process's data in this fashion suffices if the process is not interacting with external entities. Complications arise, however, if the process is interacting with external entities, such as other processes. If communications buffers referring to external entities are stored within the address space of a process that is subject to checkpointing and rollback, then these communications buffers can become inconsistent with the external entity if rollback occurs.

The **receive queue** is the queue of messages that have been delivered to a given PVM task, but have not yet been processed by PVM's `pvm_recv` primitive. From a logical point of view, messages still residing on the receive queue are not part of the task's state, and so should not be subject to rollback. However, PVM buffers some of the messages on the receive queue within the task's address space, and so these messages are subject to rollback.

If a message is delivered to a task but not yet received, and the task then rolls back to a point in time before the message was delivered, then that message can be lost.

To protect against such a circumstance, the receive queue is **exempted** from the rollback procedure using the following mechanism:

1. save the user space buffer to storage not subject to rollback, i.e., write to a non-buffered file
2. roll back the user state of the process
3. free the (now stale) user space buffers that were restored when the process was rolled back
4. load the user space buffer saved in step 1

This same mechanism is used to exempt certain HOPE data structures from rollback, such as the `hope_history` data structure, and the interval counter. Refer to Appendix A for full details on the implementation of process rollback in a PVM environment.

6.3 Future Research and Enhancements

The described prototype HOPE system is complete and consistent with the semantics of the HOPE language described in Chapter 4. However, there are several topics that should be addressed in future work.

The AID state machines are implemented as individual processes. This implementation is consistent with the abstraction that each AID is an independent entity that can be distributed in any fashion and accessed using only messages. However, it is possible that improved performance could be realized by using a single AID manager process per node of the PVM virtual machine. The PVM manager process would maintain individual state machines for each AID under its care, and otherwise function the same as the existing AID processes. This approach could improve performance by reducing the context switching and message passing overhead required to process the AID-related state transitions and messages on a single node, while maintaining the ability to distribute AID state machines across multiple nodes.

The checkpointing technique used (save the entire stack, heap, and static data area to disk) works, but is very expensive: Checkpointing a sample 250 KB process takes 20

milliseconds to a RAM disk [46], and 300 milliseconds to a remote physical disk across a 10 MBit Ethernet. More efficient methods, such as message logging [57] could be used to greatly reduce the cost of checkpointing. The potential for using compiler-driven checkpointing schemes also remains unexplored.

The data structures used to implement the dependency tracking sets are less than optimal. Numerous sites in the code have been commented with the string "`OPTIMIZE`" where obvious optimizations could be introduced, such as using a logarithmic data structure [52] instead of the simple linear linked lists that were used. However, since the current system performance is dominated by the time required to take checkpoints, spawn processes, and exchange messages, the time required for set processing does not as yet affect system performance.

Chapter 7

Analysis[1]

The fundamental purpose of optimism is to improve the response time of computer systems. The HOPE prototype that has been constructed is far from optimal, as many simplifications were made to produce the experimental software in a reasonable amount of time with minimal manpower. Nonetheless, the prototype needs to demonstrate performance enhancements to provide convincing evidence for the future of HOPE.

This chapter analyzes the performance properties of HOPE. The purpose is not to fully characterize the performance characteristics of the prototype implementation, or to characterize the performance characteristics of the particular optimistic algorithms in use. Instead, this chapter's purpose is to show that the HOPE prototype can provide performance improvements under some plausibly practical circumstances.

Bacon and Strom's Call Streaming [5] (see Section 3.1.3) provides one of the most powerful new applications for optimism: parallelizing remote procedure calls in a distributed system. To study the performance of Call Streaming in HOPE under controlled circumstances, a test application was written that executed a parameterized number of remote procedure calls, with a programmable amount of "work" (delay) in both the caller and destination processes. Section 7.1 describes the hypothesis of the experiment, Section 7.2 describes the experimental methodologies (software) used, and Section 7.3 presents the results of the experiment. Section 7.5 summarizes the results.

[1] A measure of HOPE.

7.1 Hypothesis

This experiment seeks to outline the circumstances under which a Call Streaming optimistic assumption is worthwhile. Given a fixed set of performance characteristics (the cost of a process checkpoint, the cost of process rollback, and the latency and bandwidth of inter-process message passing), **when** is it **beneficial** to make an optimistic assumption about the behaviour of a remote procedure call?

Specifically, **beneficial** means that the total response time of the application is reduced. **When** is a combination of the local delay in the process executing the RPC, the remote delay of waiting for the response from the server process, and the probability that the RPC will perform as expected.

For the purpose of repeatability of the experiments, we have approximated the "probability" of the server performing as expected with a fixed, pseudo-random sequence of events as provided by the UNIX `random()` C library function. In these experiments, "probability of success" is the fraction of calls to the server that will behave "as expected."

It is hypothesized that in circumstances that yield a profitable use of optimism, there is a relationship between the probability of expected behaviour, and the latency of the operation whose behaviour is being optimistically assumed. Specifically, it is hypothesized that for shorter-latency operations, a higher probability of expected behaviour is required before an optimistic assumption about the behaviour will yield shorter response time on average. The experiment tests the cross product of a range of probabilities and RPC delays to determine what combinations of probabilities and latencies derive benefits from optimism.

The basic experiment is to compare the response time of a pessimistic program to an optimistic program. However, there is more than one kind of optimism that can be used for Call Streaming:

Definition 7.1 Aggressive optimism *speculatively executes* **affirm** *primitives, optimistically assuming that the interval in which the* **affirm** *is executed will not be rolled back.*

Definition 7.2 Conservative optimism *constrains the program to only execute* **affirm** *primitives when the affirming process is definite.*

In aggressive optimism, **affirm** primitives may be executed speculatively, resulting in a cascade of Replace messages (see Chapter 5) replacing the affirmed AID with the set of AIDs that the affirming interval depended on. In contrast to this, conservative optimism executes

each **affirm** primitive only in definite processes, resulting in a set of Replace messages that simply delete the affirmed AID from the IDO sets of the intervals that depended on it.

The trade-off between aggressive and conservative optimism is that aggressive optimism imposes a heavier computational load on the dependency tracking system. In exchange, the **affirm** primitives can be executed sooner, which may improve the response time of the application. Conservative optimism, on the other hand, keeps the computational load on the dependency tracking system relatively light, but to achieve that must wait until the assumption verification process is definite before executing the **affirm** primitive.

The full generality of aggressive optimism is necessary for some kinds of applications, such as optimistic replication protocols [15] (as discussed in Sections 2.1.2 and 2.1.3). However, aggressive optimism may not always deliver the best performance. To test the performance impact of speculative **affirm**, the response times of both aggressive and conservative optimistic programs are compared to the response times of the pessimistic programs. It is hypothesized that either aggressive or conservative optimism will provide superior response time to the application.

7.2 Experimental Design

The test application simulates a client-server application. The client delays for a specified amount of time, simulating local "work." Then the client sends a message to the server requesting some simulated work from the server, and then waits to receive a response. The server receives the request, delays for the amount of time specified in the message to simulate server "work," and then returns the result to the client. Each such complete cycle is referred to as a **transaction**.

To simulate probabilistic expected behaviour in the RPC call, the server is started with a fixed probability of "success" between 0 and 1 as a parameter. The probability of success value is compared against the output from a pseudorandom number generator, and if the fixed probability value is greater, then the server returns the "expected" value. If the fixed value is less than the pseudorandom value, then the server "fails" and returns an unexpected value.

To simulate RPCs of various granularities, the amount of local (client) "work" and remote (server) "work" are parameters to the client process when it is started. For simplicity, it is assumed that the amount of client and server work are the same. Work is specified as

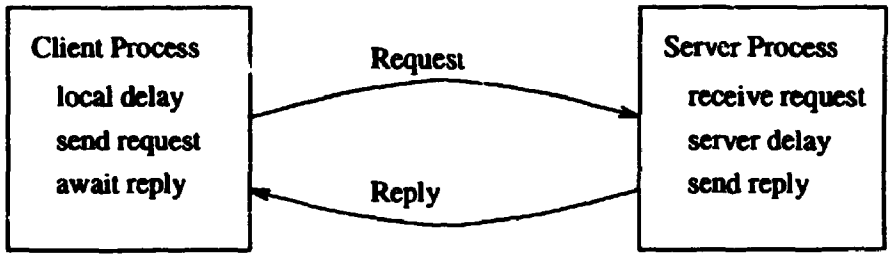

Figure 7.1: Pessimistic RPC Test Program

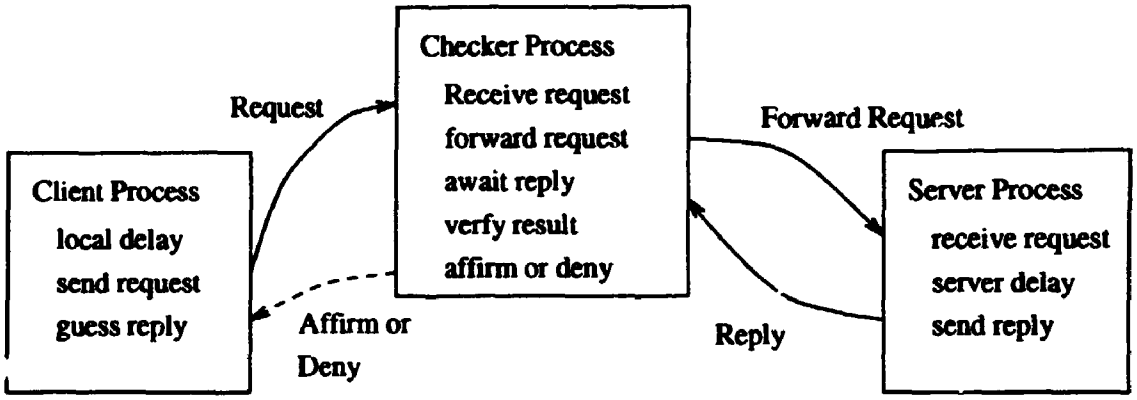

Figure 7.2: Optimistic Call Streaming RPC Test Program

delay in milliseconds.

To measure the benefit of Call Streaming, a pessimistic and an optimistic version of this application were constructed. The pessimistic version consists of a client and a server, as described above and shown in Figure 7.1. The optimistic version uses the same server process, but uses Bacon and Strom's Call Streaming algorithm in the form of a "worry wart" checker process similar to that found in Section 3.1.3, as shown in Figure 7.2. Both aggressive and conservative optimistic test programs were constructed.

The fixed parameters of the experiment are as follows:

- Guess time: 100 ms, including AID process spawn time
- Message latency: 5 ms between two workstations on a 10 Mbit Ethernet

The variable parameters of the experiment are as follows:

- number of transactions: 10, and 100

- transaction length in milliseconds: 400, 500, 600, 700, 800, 900, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000
- Probability of success at the server: 1, 0.99, 0.98, 0.97, 0.96, 0.95, 0.9, 0.8, 0.7, 0.6
- aggressive vs. conservative Optimism: Both aggressive and conservative versions of each application are presented

The results of the experiment is the **profit ratio**: the ratio of the response time for the pessimistic test program to the response time of the optin istic test program. A profit ratio greater than one indicates that optimism has improved the response time.

The theoretical limit to the speedup induced by Call Streaming is a factor of 2 improvement in response time. If the sum of the local delay and the guess time is exactly equal to the sum of the remote delay and the message latency, then the client and server processes are both 100% busy 100% of the time, achieving perfect parallelism and a speedup of 2 over the pessimistic case, in which only one of the client process or the server process can be actively executing at any given time.

7.3 Results

With two different transaction counts, 16 transaction lengths, 10 different success probabilities, and two different types of optimism, there are 640 experimental results to report, requiring a graphical presentation. The four types of parameters produce a 4-dimensional space of results, making the results difficult to plot on a graph.

To address these problems, we present the results on multiple graphs. First, in all cases the 10 transaction cases are plotted separately from the 100 transaction cases. The aggressive optimism cases are also plotted separately from the conservative optimism cases.

Figures 7.3 and 7.4 present the results for aggressive and conservative optimism for the 10 transaction experiments. The graphs plot the profit ratio with respect to the transaction length for some selected probabilities: 1, 0.95, 0.9, and 0.7.

Figures 7.5 and 7.6 present similar results for aggressive and conservative optimism for the 100 transaction experiments. Again, the graphs plot the profit ratio with respect to the transaction length for some selected probabilities: 1, 0.95, 0.9, and 0.7.

Figures 7.7 and 7.8 present more general results for the aggressive and conservative 10 transaction experiments. The graph plots a 3-dimensional contour surface representing

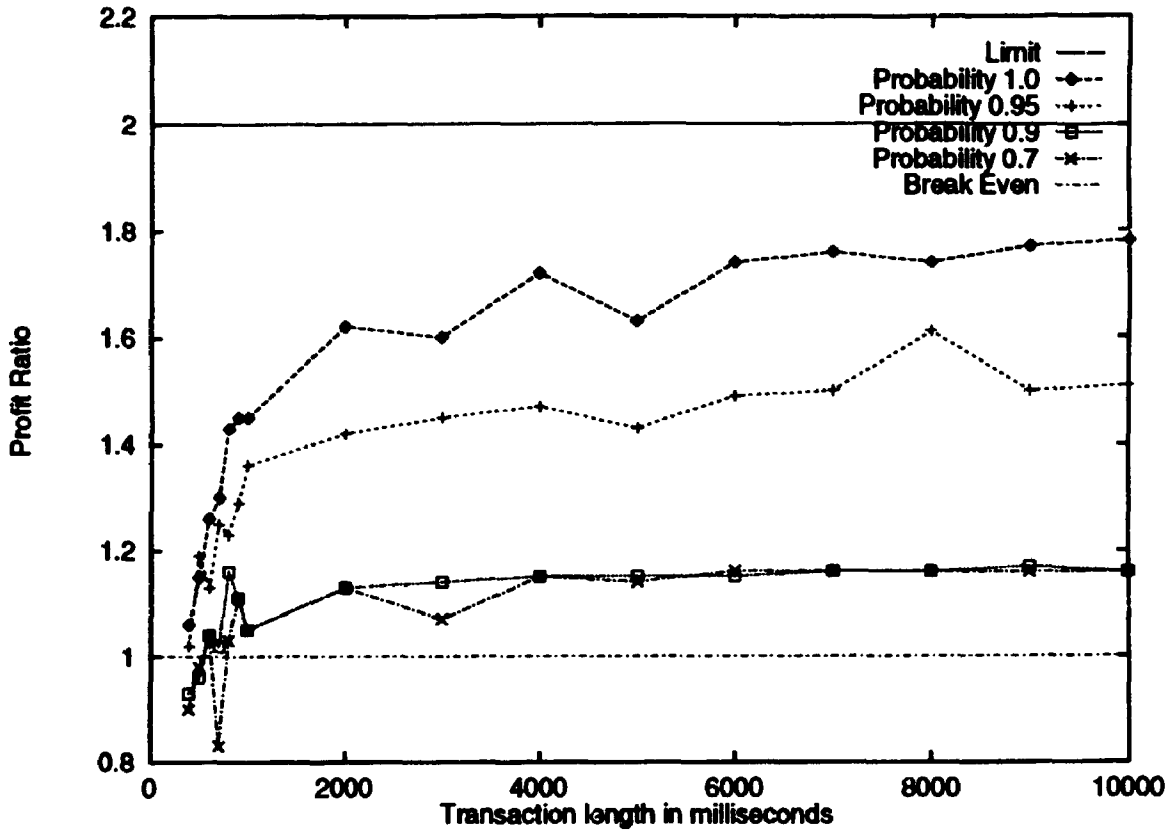

Figure 7.3: Profit Ratio: Aggressive Optimism, 10 Transactions, Selected Probabilities

the profit ratio between the optimistic and pessimistic test programs for aggressive and conservative optimism, respectively. A flat plane surface has been added to represent the "Break Even" point of a profit ratio of 1.0. Hidden surfaces have been eliminated to make the graph more readable.

Figures 7.9 and 7.10 present similar 3-dimensional contour surfaces for the aggressive and conservative 100 transaction experiments.

7.4 Interpretation

Longer transactions should produce greater benefit from optimism, because the amount of delay thus avoided increases. Similarly, a higher probability that the RPC will perform as expected should produce greater benefit from optimism. As expected, all of the graphs generally show that longer transactions and higher probabilities produce larger speedups due to optimism.

Somewhat more interesting is that this increase in performance due to increased prob-

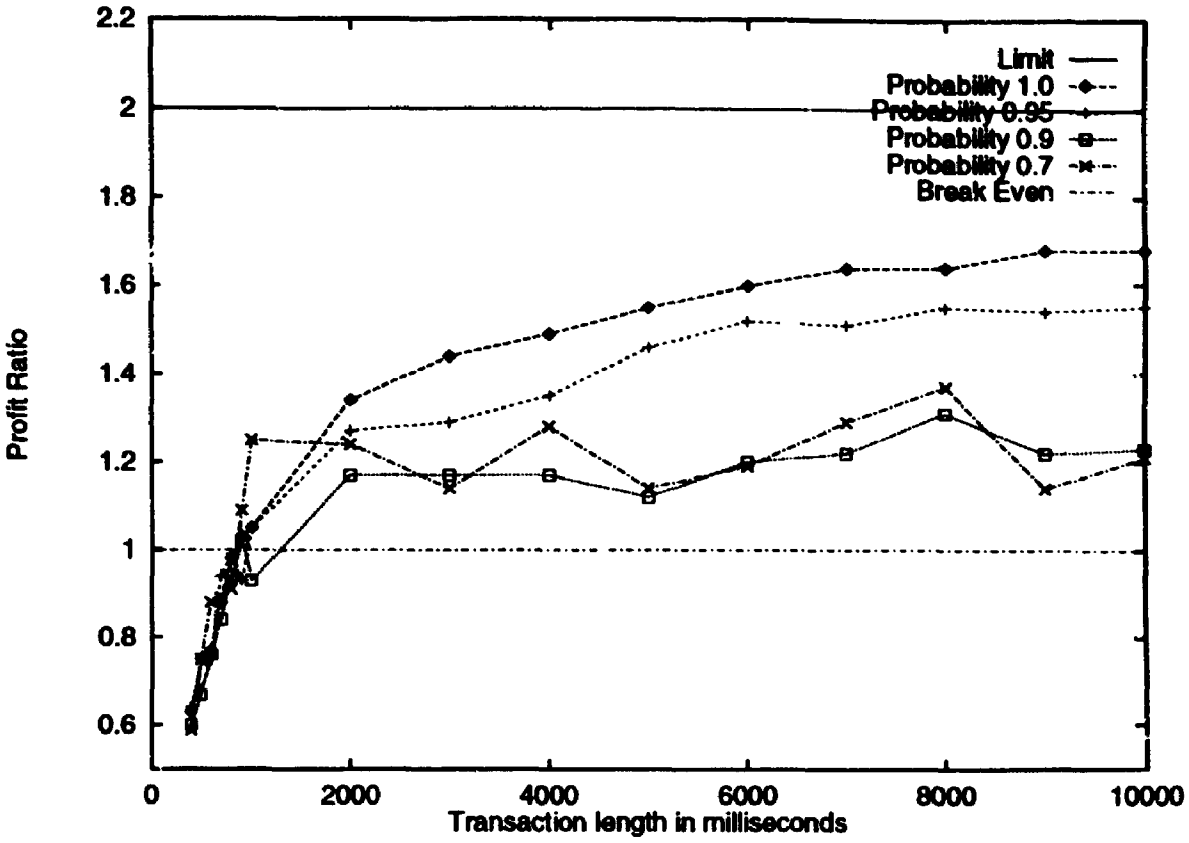

Figure 7.4: Profit Ratio: Conservative Optimism, 10 Transactions, Selected Probabilities

abilities and increased transaction lengths is non-linear. In each of Figures 7.3, 7.4, 7.5, and 7.6, as the transaction length increases, the profitability rises sharply at first, and then becomes asymptotic.

Consider the case where the probability of the RPC behaving as expected is 1.0[2]. The initial sharp rise in profitability reflects the transition from **guess** and **affirm** primitives being slower than the RPC operations to the **guess** and **affirm** operations becoming faster than the RPC operations as the length of the RPC transactions increases. Once the cost of the optimistic operations is less than the cost of the RPC operations, the profit ratio becomes asymptotic to 2.0 (the theoretical limit to speedup in this algorithm). As the transaction length increases, the costs of optimism are amortized over more time, gradually increasing the benefit asymptotic to the limit.

The initial phase is largely unaffected by the probability of expected behaviour. If the execution of a **guess** primitive is more costly than the RPC itself, it doesn't matter very much whether the guess is correct or not. However, once the **guess** and **affirm** primitives

[2] $P = 1.0$ isn't really optimism, but illustrates the operation of the HOPE system in the simplest case.

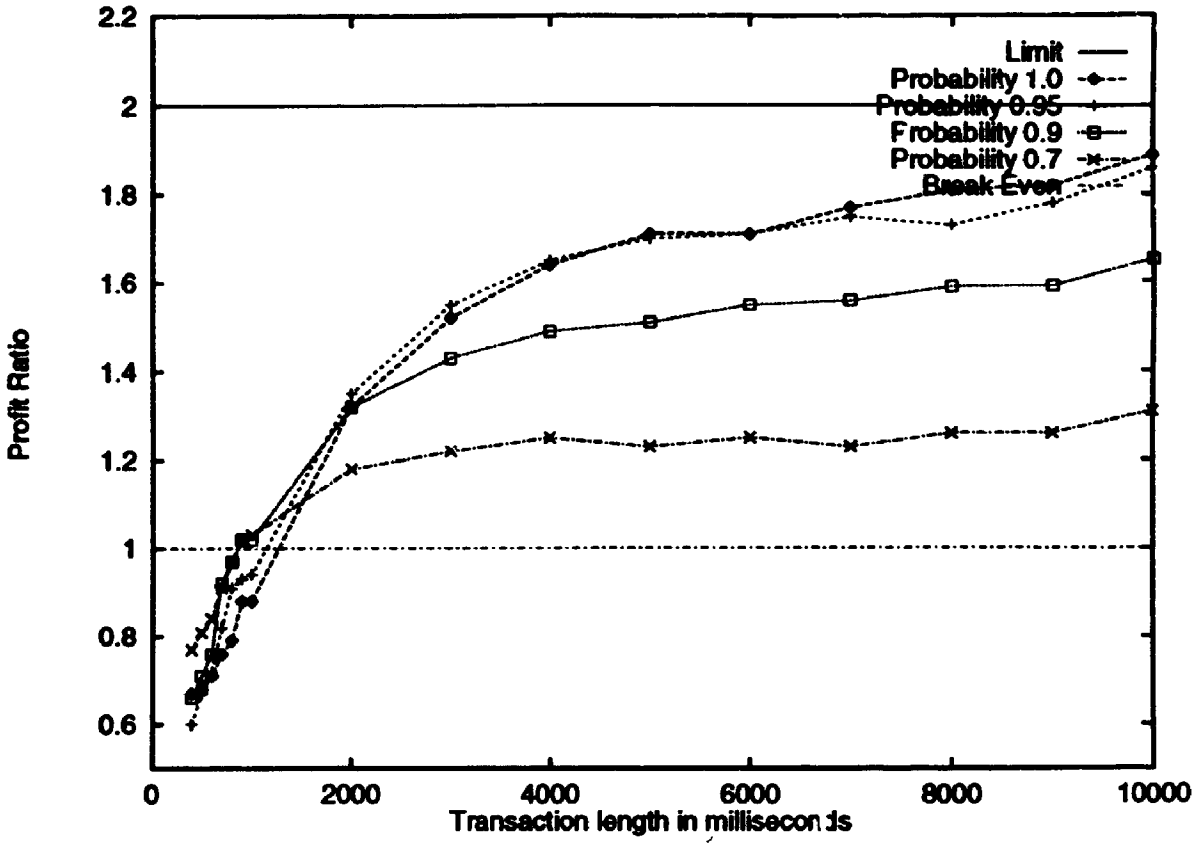

Figure 7.5: Profit Ratio: Aggressive Optimism, 100 Transactions, Selected Probabilities

are less costly than the RPC, the probability of expected behavior begins to distinguish the speedup due to optimism. For instance, at a probability of 0.7, 70% of the RPC operations will produce the correct result and yield a benefit, while 30% will produce an unexpected result and consume the same amount of time as in the pessimistic program, plus the cost of checkpoint and rollback. Thus the theoretical limit to speedup due to optimism with a probability of 0.7 is a factor of $\frac{2}{0.7+0.3\times 2} \approx 1.54$. As transaction length increases, the cost of checkpoint and rollback is amortized into longer transactions and reduces in significance, and the speedup due to optimism becomes asymptotic to the theoretical limit imposed by the probability of expected behaviour.

It is interesting to note that even for transactions with a poor probability of producing the expected result (i.e. 0.6, as shown in Figures 7.7, 7.8, 7.9, and 7.10), the optimistic test programs always produced a performance enhancement for transactions of 2.0 seconds or greater. It is a theorem due to Jefferson [31] that the amount of time "wasted" on speculative computations based on incorrect optimistic assumptions would have been spent blocked waiting for results in a pessimistic program. The only penalty for using optimism is

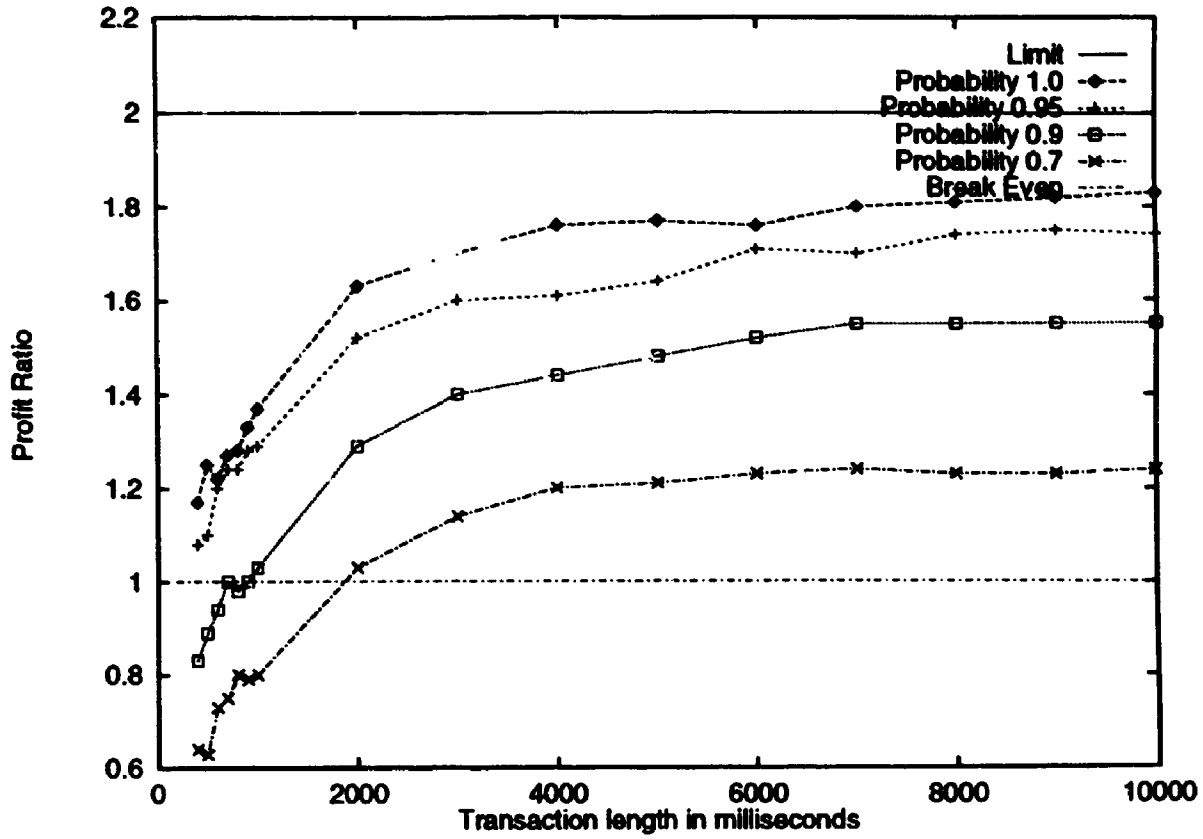

Figure 7.6: Profit Ratio: Conservative Optimism, 100 Transactions, Selected Probabilities

the extra time required to perform checkpointing, dependency tracking, and rollback. Thus we should expect that for an arbitrarily remote chance of expected behaviour, there should exist a sufficiently long transaction to make such a optimistic assumption beneficial. In point of fact, for a 10 transaction execution with 10 second transactions and a probability of expected behaviour of only 0.1, an aggressively optimistic program produces a speedup of 1.04, and a conservatively optimistic program produces a speedup of 1.11.[3]

For the 10 transaction experiments and transactions shorter than 6.0 seconds, the aggressively optimistic test program provided strictly superior performance relative to the conservatively optimistic test program. Conversely, for the 100 transaction experiments and transactions less than 6.0 seconds, the conservatively optimistic test program provided superior performance relative to the aggressively optimistic test program for all probabilities higher than 0.7. This is because the aggressively optimistic program is using speculative **affirm** primitives.

The semantics of an **affirm(X)** within a speculative interval *A* specify updates to all

[3]This program was measured, but not shown in the graphs because it is far out of range.

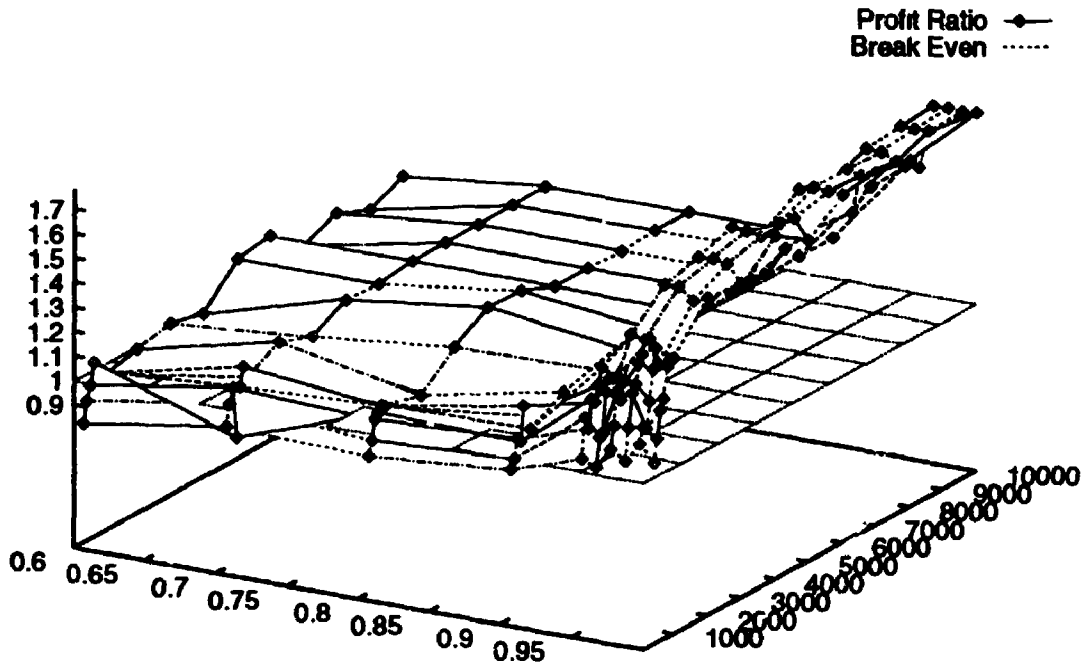

Figure 7.7: Profit Ratio: Aggressive Optimism, 10 Transactions, All Values

assumption identifiers Y that A depends on involving all intervals B that depend on X, and updates to all intervals B involving all assumption identifiers Y. Thus the complexity of a speculative **affirm** is intrinsically quadratic in the number of assumption identifiers in $A.IDO$ and the number of intervals in $X.DOM$. If the rate of confirmation of **guess** primitives is slower than the rate of executing new **guess** primitives, then the size of $A.IDO$ in the checking process will grow with the number of transactions.

For a small number of transactions, the actual computation required for **affirm** is sub-linear, and so the aggressive program provides greater speedup. For a large number of transactions, the quadratic behaviour dominates, and so the conservative program provides greater speedup. For longer transactions (in this implementation, greater than 6.0 seconds) the rate of executing new **guess** primitives is lower than the rate of **affirm** primitives applied to previous optimistic assumptions, and so the quadratic complexity of speculative execution of **affirm** primitives becomes irrelevant.

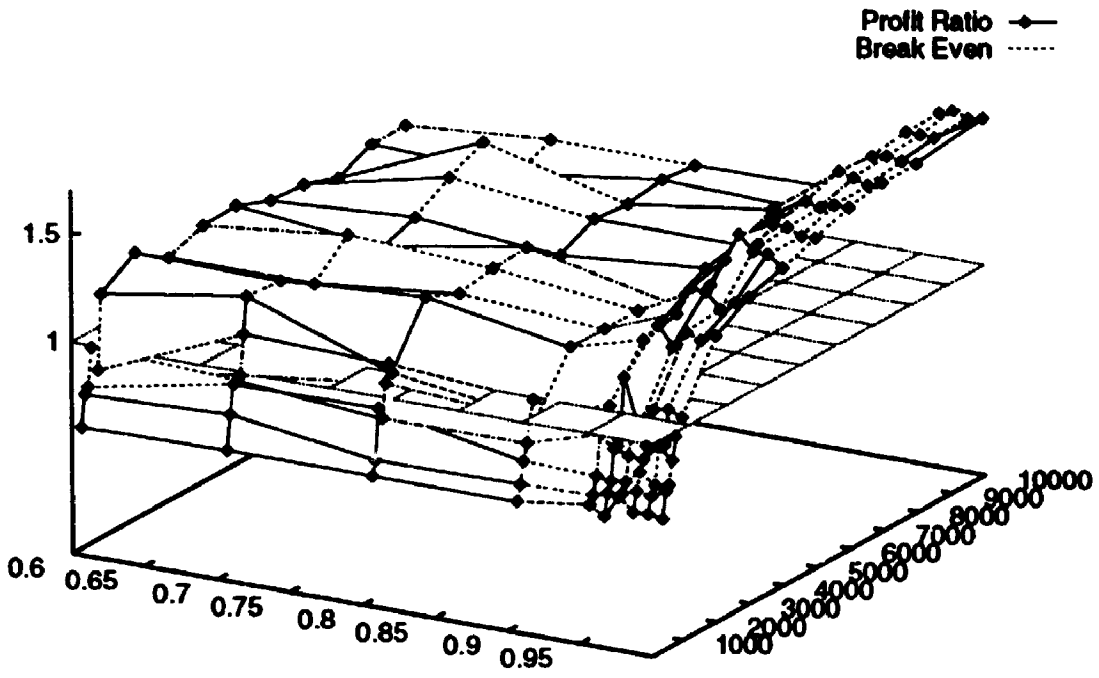

Figure 7.8: Profit Ratio: Conservative Optimism, 10 Transactions, All Value

7.5 Summary

The performance experiments presented here are far from comprehensive. The experiments presented exhibit a number of interesting artifacts which merit further investigation. As shown in Chapter 2, Call Streaming is not the only kind of optimistic algorithm. In particular, an investigation should be made into the performance of optimistic replication algorithms, which often require speculative **affirm** primitives to function correctly [15].

However, these experiments have shown that the HOPE prototype system can deliver performance gains in program response time for all optimistic assumptions relating to remote operations tested taking more than 2.0 seconds and having a greater than 60% chance of producing the expected result. This is a sufficiently broad range of opportunity for optimism that HOPE can be used as a tool for writing optimistic programs, which is its design goal.

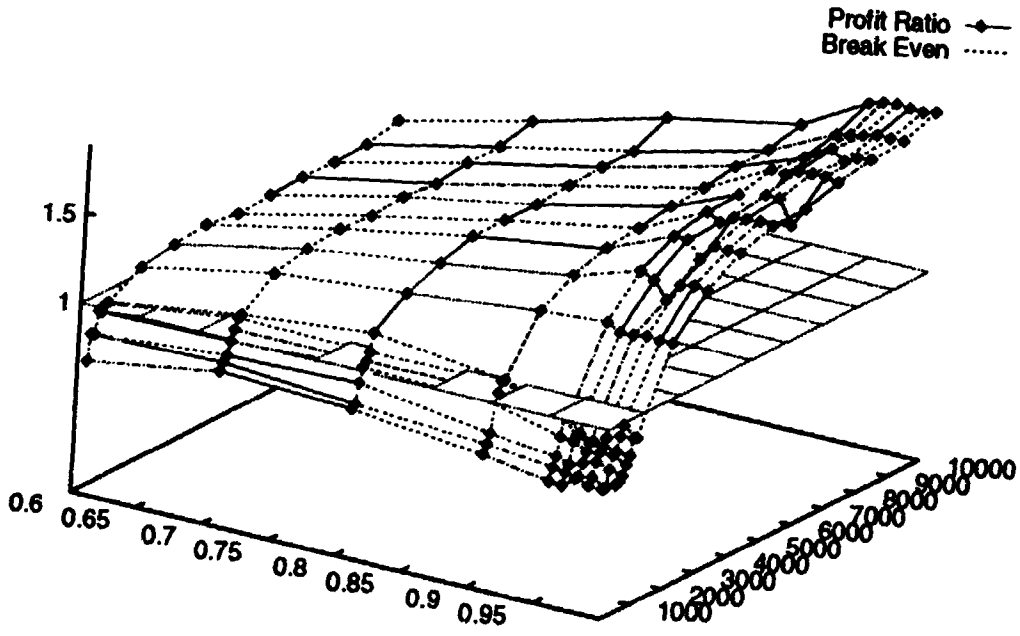

Figure 7.9: Profit Ratio: Aggressive Optimism, 100 Transactions, All Values

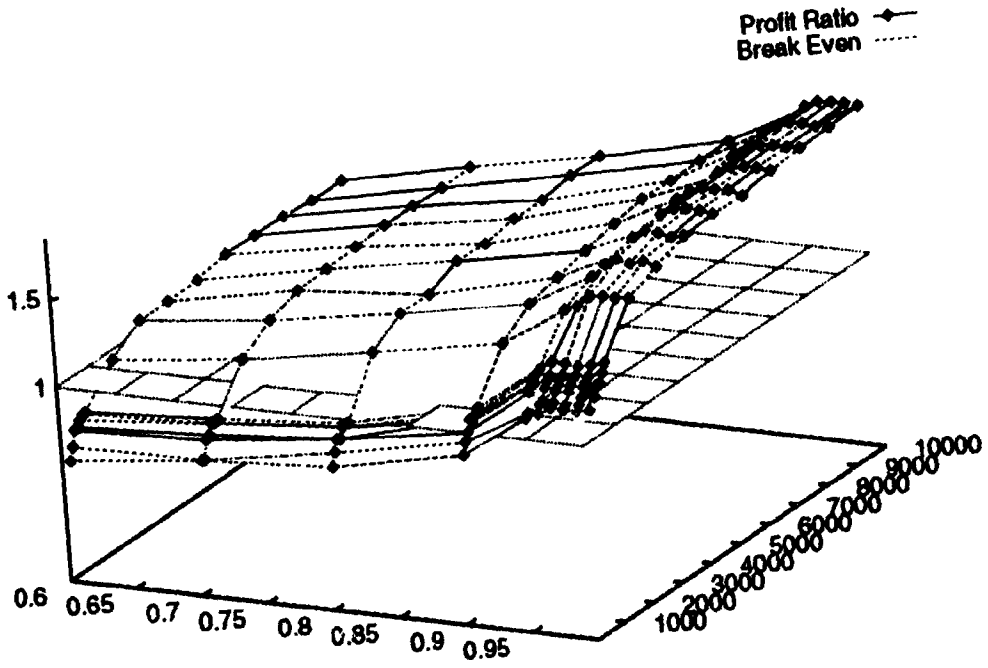

Figure 7.10: Profit Ratio: Conservative Optimism, 100 Transactions, All Values

Chapter 8

Conclusions[1]

This chapter summarizes the results presented in this thesis. Section 8.1 describes the results produced from HOPE. Section 8.2 describes future research opportunities for HOPE. Section 8.2.4 summarizes the results.

8.1 Results

HOPE is a novel programming model for writing optimistic programs. HOPE is more general than previous systems that support the expression of optimistic algorithms in that previous systems are all limited in some way:

- Limited scope of optimism
- Limited use of optimistic results until they have been confirmed
- Limited type of optimistic assumptions
- Limited application domain

HOPE is unrestricted in all of these areas. HOPE provides a general framework in which any optimistic assumption can be made about the future state of the application. Any subset of the processes in the application can become dependent on an optimistic assumption. Any process in the system can decide the correctness of the optimistic assumption, and the process making such a decision can be dynamically changed.

[1] HOPE for the future.

A formal semantics of the meaning of HOPE has been presented. Using this formal definition of HOPE, implementations of HOPE can be constructed on top of any system providing concurrent processes that communicate via messages.

An prototype implementation of HOPE has been constructed, and its algorithms have been described. The prototype allows HOPE programs to be written in ordinary C, with message passing primitives provided by the widely-used PVM system [62], and HOPE primitives provided by additions and modifications to the PVM library of functions.

While the performance delivered by the HOPE prototype is far from optimal, the design is faithful to the basic use of optimism, which is to avoid latency. None of the HOPE primitives in the prototype ever wait for confirmation from a remote resource before returning control to the user process. Initial performance evaluation of the prototype shows that it can deliver improvements over a range of circumstances broad enough for HOPE to be a viable tool for optimistic algorithm research.

8.2 Future Research

A great deal of effort has gone into HOPE research to date, but there are many additional opportunities for research. Future research avenues can be broadly divided into three categories: performance enhancing algorithms for HOPE implementations, characterizing viable conditions for optimism, and applications of HOPE.

8.2.1 HOPE Design and Implementation Issues

The algorithms used to implement HOPE were designed primarily with the goal of avoiding blocking user processes when HOPE primitives are invoked. This is consistent with the basic purpose of optimism (avoiding latency), but does not necessarily provide the best performance. The forward error recovery approach taken to address the potential conflicts that arise in a concurrent implementation of HOPE is effectively an optimistic algorithm itself: It is optimistically assumed that conflicts will not occur, and corrective measures are taken if conflicts do occur. The merits of this assumption, especially with respect to the impact on response time of HOPE applications, warrant further study.

The checkpointing mechanism used in the HOPE prototype is simple and effective, but not very fast. Checkpointing mechanisms often assume that they must transparently work with arbitrary programs. However, since HOPE programs already have been enhanced at

the source-code level to include the use of HOPE primitives, it is possible to introduce *compiler* driven checkpointing mechanisms [58]. A compiler has enough information to determine what aspects of a process's state have potentially changed between checkpoints. Instead of checkpointing the entire state of the process, a compiler driven checkpoint mechanism can save only those aspects of the state that have changed.

Both of these topics are of great interest, and will be pursued. However, they are fundamentally separate from the question of providing a basic programming model for optimism and demonstrating its viability, and so have been left to future work.

8.2.2 Conditions for Optimism

Chapter 7 clearly shows that the latency of an operation and the probability of the operation behaving as expected have a large impact on whether it is beneficial to use optimism to guess the results of the operation. It was expected that the probability of expected behaviour would be the dominant factor in determining whether optimism is called for. However, the results presented in Chapter 7 suggest that the latency of the operation in question may be even more important in determining if optimism is warranted. More research in characterizing the conditions that mandate the use of optimism is called for.

8.2.3 Applications of HOPE

The HOPE prototype can show performance improvements for a broad range of optimistic opportunities. However, because the prototype contains less than one man-year of work, it cannot hope to compete with the tuned industrial-strength checkpoint and rollback mechanisms of commercial products such as database systems or simulation engines, which may contain hundreds or thousands of man-years of work. Thus, there is little point in applying HOPE to traditional application areas for optimism.

Of greater interest is applying HOPE to some non-traditional application areas of optimism. The Call Streaming algorithms described in Chapter 3 and studied in Chapter 7 have broad applicability to the burgeoning area of distributed computing. Distributed systems suffer from greatly increased latency in communicating with remote elements of a program, and Call Streaming can often mask that latency with optimism.

Replication is a growing area of interest in distributed systems. Like optimism, replication can mask the latency of remote operations by moving replicas of an object or process closer to those modules that need to access it, reducing both communications latency and

overload on a single node attempting to serve an entire system. However, replication often trades reduced *read* latency for increased *write* latency [15]. Using optimism to maintain consistency among replicas can help to reduce this increase in the latency of writing to replicated objects. HOPE can be of particular assistance by making it feasible to incorporate optimistic object replication directly into an application. Without HOPE, the software complexity of dependency tracking, checkpointing, and rollback would be prohibitive to most applications programmers.

Cooperative work is an active new area of study in distributed systems. It is a particular specialization of replication, in which a document is replicated among multiple users who concurrently edit the document. Cooperative editors often use locking schemes to maintain consistency among the replicated documents [21, 25, 37, 41]. If the multiple users are geographically removed from one another, locking can impose undesirable amounts of latency on traditionally "simple" edit operations. Optimistic replicated consistency protocols can help reduce this latency.

Optimism is often used in hardware fault-tolerant systems to maintain consistency. Executable assertions are often used to ensure correct behaviour in software fault-tolerance research. However, the time required to verify some executable assertions may be prohibitive. Replacing an executable assertion with a HOPE **guess**, and concurrently executing the assertion to validate the optimistic assumption can avoid imposing this latency on the application [44].

Automatic transformation of pessimistic programs into optimistic programs is an attractive notion because pessimistic programs are easier for programmers to reason about and understand. Both Strom et al. [5, 59], and Bubenik [9] have studied the prospect of automatically optimistic transformation. However, Strom did not have an environment for executing optimistic programs, and so did no experimental work. Bubenik did construct an environment for executing optimistic programs, but it is much more restrictive than HOPE, and in fact is incapable of executing many of the examples presented in this thesis, including Call Streaming. The enhanced prospects for automatic transformation of pessimistic programs into optimistic programs using the HOPE model should be studied.

In the area of artificial intelligence, **non-monotonic reasoning** is the notion of *tentatively* drawing a conclusion based on questionable evidence or reasoning, and if the conclusion is later disproved, discarding the conclusion from the knowledge base. Truth Maintenance systems [18, 19, 20] do their own management of adding and removing conclusions

from the knowledge base. By using HOPE optimistic assumptions to tentatively add questionable conclusions to the knowledge base, the task of managing removal of incorrect conclusions from the knowledge base can be automated, effectively allowing simpler monotonic reasoning systems to be made non-monotonic.

8.2.4 Summary

The complexity and tedium of dependency tracking, checkpointing, and rollback have made research into optimistic programming difficult to conduct. HOPE was designed to ease these burdens, making research into optimistic programming much easier to do. The existence of the above mentioned future research topics is evidence that this goal has been achieved, and there is HOPE for the future.

Appendix A

Rollback[1]

Checkpoint and restart of a process is a standard technique used to protect work in progress from failure of the machine on which the process is running. While the checkpointing mechanisms used in restart are similar to those used in rollback, rollback is distinguished from restart in that the process P, instead of being killed and recreated, is simply returned to some previous state. Process P retains its process-ID and all associated kernel resources. By retaining P's process-ID and kernel resources, other processes in the system can still communicate with P, its files and pipes stay open, and security issues are avoided.

This appendix describes our checkpoint and rollback facility for UNIX processes. The desired facility is a portable, user-space mechanism for checkpointing the state of a UNIX process in C, and then later rolling the process back to that state. There are numerous checkpoint and restart systems [7, 14, 34, 43, 67] for UNIX processes. There are also numerous specific systems that include rollback, but are confined to some specific environment or domain, such as discrete event simulation [22, 31, 65] or transaction processing [53, 56, 66]. Finally, there are numerous studies examining the theoretical properties of various schemes for maintaining consistency in the presence of distributed rollback [5, 16, 42, 54, 55]. However, an extensive literature search failed to turn up a general purpose rollback facility for UNIX processes. Systems that actually do process rollback seem always to do so in a restricted domain.

We begin in Section A.1 by surveying some checkpoint and restart facilities for UNIX. Section A.2 describes how we adapted such a checkpoint and restart system to implement rollback of UNIX processes. Section A.3 describes the changes necessary to allow for buffered

[1] Abandon all HOPE ye who enter here...

communications used in distributed computing environments such as PVM. Finaliy, Section A.4 describes the special problems that debugging a rollback facility presents, because the rollback action violates most notions of safe programming.

A.1 Related Work

This section surveys several checkpoint and restart systems for their suitability as a basis for a rollback system. By using an existing checkpoint mechanism and adapting the restart mechanism to act as a basis for a rollback mechanism, the work required to implement rollback can be minimized. To be suitable as a basis, the checkpoint mechanism should be portable, it must leave the process running, the source code must be available and the restart mechanism must be adaptable to an already running process.

Kingsbury and Kline [34] describe the effort at Cray Research to add a checkpoint and restart facility to their UNICOS variant of UNIX. While highly sophisticated and robust, UNICOS is also commercial software bound to a specific platform; this is not suitable. Condor [43] is a resource management system that implements process migration, necessitating process checkpoint and restart. Unfortunately, Condor's checkpoint mechanism, a special signal causing the process to dump core, terminates the process, and so it is not suitable for rollback. Recently, the Condor project has undertaken to create a new checkpointing mechanism that does not kill the process [51], but details are not available at this time.

Yee's Save World [67] and Bernstein's Pmckpt [7] are similar to one another, in that they are small libraries of portable software that provide rudimentary process checkpoint and restart. Both save and restore process state by simply reading and writing the appropriate address ranges. They differ in that Save World resorts to a small amount of assembly code to save and restore the stack frame (making it relatively portable), while Pmckpt uses preprocessing of the user's source code. Neither handles open files. The restart mechanism in both cases consists of re-running the user's program with appropriate switches that cause the program to invoke the state restoration mechanism instead of proceeding to the user's code. Because they are portable, publicly available and do not kill the subject process, they are both suitable bases on which to build a rollback facility.

A.2 Basic Checkpoint and Rollback of UNIX Processes

We created this checkpoint and rollback mechanism to support our work in optimism and fault tolerance. The desired facility is a portable, user-space mechanism for checkpointing the state of a UNIX process in C, and then later rolling the process back to the state that was checkpointed. While efficiency was a concern, it was secondary to simplicity. The mechanism should roll back all program-visible aspects of the user data space. Since the mechanism is rollback instead of restart, there is no need to restore code space or kernel space state data. The state of a process comprises four principal components:

- the stack
- the heap
- static data
- the program counter

Both Pmckpt and Save World save this data. Pmckpt offers the advantage of not requiring assembly language code. However, this is at the expense of requiring transformations on the source code of the user's program. Save World requires no such transformations, but uses a modicum of assembly language code to manipulate the stack pointer and program counter.

Both make the assumption that each of the three data areas is contiguously mapped, and checkpoints them by taking the start and end address of each and saving each to disk with a single `write`. This assumption, while in violation of the ANSI C Standard [3], is nonetheless true of a large number of C/UNIX implementations.

In both cases, the restart mechanism is to re-execute the user's program with special switches that cause the program to invoke the restart mechanism instead of proceeding to the user's code. Pmckpt provides a meta-`main` function and a shell script wrapper to make the restart command switch transparent, and so requires some adapting to be transformed into a rollback facility. Save World just provides a `restore_world` function, which can simply be called to produce a rollback effect.

We built rollback mechanisms using both checkpointing systems, but found Save World to be adequately portable and less troublesome.

A.3 I/O and Distributed Computing

Checkpointing and restoring the state of a single process is relatively simple, but processes mostly interact with external entities such as files, devices, or other processes. Saving and restoring the state of these external entities is often difficult or impossible, because the state of the entity is large, or impossible to access.

There are a number of ways to cope with external entities when performing checkpoint and restart or rollback. UNICOS checkpoints and restores groups of cooperating processes as a single job, and because it is only doing restart and not rollback, it need not be concerned with rolling back write operations on files. Condor, like UNICOS, relies on the files not changing between checkpoint and restart so that a reopen and `lseek` will suffice to re-establish contact with the files. Condor copes with external processes by simply prohibiting them; only single process jobs are migrated.

In general, while write operations on internal state are recoverable by recovering the state of the process, general write operations to external entities are considerably more difficult to recover. In a limited domain, such as transaction processing, write operations to files can be made recoverable by logging writes to the file and only applying the writes once the transaction has committed. Such a technique is not applicable to general purpose process rollback, because it is assumed that the transaction will not attempt to read back the changes that it has written to its files. This assumption is invalid with respect to user processes.

If the external entity is a process, then the state changes imposed (in the form of messages) can take on arbitrary form. The only way to recover the state of the external process is to force it also to roll back. Such a strategy requires either a globally consistent snapshot [34, 40], or a dependency tracking scheme such as HOPE. If the external entity is a device, such as a `tty`, then write operations can only be rolled back with specific knowledge of what the operation was and the device's resultant behavior. If the device was a printer or a bank machine issuing money, then the write operation cannot be recovered at all. Thus we consider all write operations to external entities to be unrecoverable for the purposes of rolling back a single UNIX process, unless the external entity is also rolled back.

The best that can be done is to preserve the consistency of the state of the connection to these external entities. Because connections to files are in part stored in kernel space, systems such as UNICOS and Condor must take special action to preserve and re-establish

(reopen) connections to these files. In a rollback environment, we have the advantage that the kernel state for open files still exists, so our remaining concern is consistency with respect to files that are opened or closed between checkpoint and rollback. Specifically, between checkpoint and rollback:

- Open files will remain open.
- Closed files will remain closed.
- Files opened between checkpoint and rollback are simply reopened. This consumes a file handle that is not recovered by the program, but it will be closed by default when the process exits.
- Files closed between checkpoint and rollback are problematic. For our purposes, we consider closing a file to be an unrecoverable write operation on the file.

For unbuffered I/O, the above suffices to maintain consistency. Since the file position is stored in user space, it is recovered by user space rollback and no further action is required to maintain the consistency of the file handle.

Buffered I/O, as used both in level 3 library functions (for both pipes and files) and distributed communications systems such as PVM, is more challenging. For instance, the PVM system will down load messages into a user process's address space and buffer them there until they are actually received by the process using a receive primitive. However, rollback of the process would roll back the *entire* address space, including the buffer holding these pending messages. The pending messages would be lost before they were ever received.

Because buffered communications of this kind move data asynchronously between kernel and user space, steps must be taken to ensure consistency of the user and kernel space versions of the connection to the external file or entity. To perform rollback when buffered I/O connections are open, the following exemption procedure must be used:

1. save the user space buffer to storage not subject to rollback, i.e., write to a non-buffered file
2. roll back the user state of the process
3. free the (now stale) user space buffers that were restored when the process was rolled back

4. load the user space buffer saved in step 1

This procedure effectively exempts the buffered I/O connection from the effects of rollback, allowing it to remain consistent with the state of the external entity.

While write operations to external entities are not recoverable, read operations are recoverable. For files, repositioning the file position pointer suffices to allow the process to reread the data that it read between checkpoint and rollback. The same technique, however, does not suffice for connections with other processes. The data provided by another process, through a connection such as a pipe, cannot be rewound for re-reading.

To allow a process P to reread data that it received from an external process when P is rolled back, it is necessary to log the data that P received as it is read from the buffer, e.g., log the messages that the process receives or the segments of the byte stream that the process reads from a socket. To restore the logged messages, we add a fifth step to the exemption procedure:

5. load the logged data ahead of the restored buffered data

Thus when P requests data from the buffer, it re-reads the data that it was given on the previous attempt.

PVM buffers messages that have arrived at a process P, but have not yet been read with the `pvm_recv` primitive, in P's address space. Our modifications to PVM for rollback include the exemption procedure, and logging of messages received so that they can be re-received on rollback.

Similar modifications can be made to support other forms of buffered I/O, such as pipes or files. The exemption procedure will maintain consistency between the user space buffer and the external entity. The message logging procedure will allow re-reading of data for entities, such as pipes, that cannot be `seek`'d. We have not made these modifications to the buffered file I/O system because they were not pertinent to our research and because they would require access to source code for the buffered I/O libraries.

A.4 Debugging Rollback

Debugging an implementation of rollback presents some special problems, especially when the rollback is implemented from within the process itself. This section describes the problems and solutions we encountered while debugging rollback.

First, because the checkpoint and rollback mechanisms regard all of the process's stack, heap, and static data as single objects, attempts to read and write these objects constitute access violations of the abstract data types contained within. When pointer problems arise it is difficult to tell whether they result from the usual reasons for pointer problems or if the problem is that either the pointer or the data object itself has been corrupted by rollback.

The solution to the problem of data corruption due to checkpoint and rollback was to perform careful comparisons on the checkpoint files. In principle, a checkpoint followed by a rollback and another checkpoint of the same state should produce two identical checkpoint files. Examining the differences between such files can lead to sources of corruption.

Second, because rollback is modifying the entire addressable state of the process, it will also modify any structures in the process's space placed there by debugging utilities. Debuggers invariably store some of their debugging information in the user process's address space. Hence a buggy rollback mechanism can corrupt the debugging structures as well. Some debugging tools such as Purify [27] go to considerable effort to avoid this problem by mapping in their debugging structures in the middle of the address space, well away from the stack and data areas. As a result, the debugging information is not rolled back with the rest of the process. While this has the benefit of protecting the debugging structures from the rollback mechanism being debugged, it also means that the debugging information describing the state of the process's dynamic memory structures is now out of date. The result is that it reports numerous access violations. We did not find any good solution to the problem of corrupted debugger structures, other than to resort to the most primitive of debugging tools: `printf`.

The problem of whether or not debugging information is rolled back along with the process hints at a more general problem: Is all of the process's state being captured and rolled back? We discovered the hard way that finding the start and end of the stack, heap, and static data areas does not always suffice. Our rollback mechanism suffered a subtle problem with data that was frequently `malloc`'d and `free`'d across multiple rollbacks. The problem manifested itself under SunOS 4.1.x on both Sun 3 and SPARC machines, using both Pmckpt and Save World. The problem did *not* manifest itself under any kind of `malloc`-debugger, such as Sun's `malloc_debug` or Purify. The problem also did not manifest itself when ported to other platforms such as a Sequent, or when `malloc` libraries other than the SunOS-provided `malloc` were used, such as NetBSD `malloc` or GNU `g++` `malloc`. Thus we concluded that there was some subtle form of state that SunOS `malloc`

was storing outside of the discovered range of the stack, heap, and static data areas. The specific data being lost cannot be determined without source code to SunOS `malloc`.

Uncaptured state, as manifested by `malloc`, proved to be the most difficult problem to debug. We spent considerable effort examining all aspects of the checkpoint and rollback system, the buffered I/O exemption system, as well as the application being tested. We were searching for flaws that could cause corruption of `malloc`'s data structures. We applied the most powerful `malloc` debugging tools we could find [27], to no avail; the problem always vanished in the presence of `malloc` debuggers. While trying to symbolically trace the precise location of the corruption, it was only accidentally discovered that alternative `malloc` libraries did not manifest the problem, and that the solution was to replace SunOS's `malloc`. The only solution to this problem that we have found is to anticipate it, and be certain that all relevant state is represented in a checkpoint.

Appendix B

Alternative Semantics[1]

The semantics in Chapter 4 specify that the rollback of a speculative **affirm** execution are equivalent to a **deny**. This is a conservative approximation. It is difficult to completely undo the effects of a speculative **affirm**, so once an **affirm(X)** has been executed and rolled back, it is difficult to determine whether X has been affirmed or not in a consistent manner. Since it is hard to continue to track X consistently, Chapter 4 conservatively applies a **deny(X)** at the time **affirm(X)** is rolled back to maintain the consistency of the program.

If this solution is unsatisfactory, it is possible to completely roll back the effects of a speculative **affirm**. However, the cost is that **affirm** becomes weaker. In particular, **affirm** no longer detects the existence of cyclic dependencies. Cyclic dependencies do not arise often in optimistic programming, but it is necessary to detect cyclic dependencies if HOPE is to be used to model the atomicity of transactions.

This appendix provides an alternative semantics to Chapter 4. The semantics presented here provide for the complete rollback of speculative affirms, but do not provide for the detection of cyclic dependencies. All other linguistic properties remain the same, as shown in the variants of the proofs.

B.1 Semantics of the HOPE Constructs

This section defines the HOPE constructs in terms of the abstract machine described in the previous section by defining the effects of each HOPE primitive on the execution sequences. All sequences are initially null and all sets are initially empty.

[1] Array of HOPE

B.1.1 Guess

Process P executing a **guess(X)** primitive in state S_i asserts an optimistic assumption and associates the assumption identifier X. The creation of the new interval, A, requires that a checkpoint be taken of the current state, including the program counter. This is represented as follows:

$$A.PS \leftarrow S_i \tag{B.1}$$

$$A.PID \leftarrow P \tag{B.2}$$

We should note that the process identifier (in PID) is also recorded. The recording of the process identifier is a naming convenience.

The interval A is speculative, and is dependent on the optimistic assumption associated with the assumption identifier X. In other words, the interval A is rolled back if the optimistic assumption associated with X is discovered to be false. If the interval preceding A is speculative then the interval A is also dependent on the optimistic assumptions that the preceding interval is dependent on. For interval A, the set $A.IDO$ is used to track A's dependencies. At the beginning of interval A (i.e., when the **guess** primitive is executed), the dependencies are as follows:

$$A.IDO \leftarrow (S_i.I).IDO \cup \{X\} \tag{B.3}$$

Associated with each assumption identifier X is the set $X.DOM$ that tracks the intervals that are dependent on X. Thus X records its new dependent interval A as follows:

$$X.DOM \leftarrow X.DOM \cup \{A\} \tag{B.4}$$

The state S_{i+1} is constructed from S_i as follows:

$$\begin{aligned} S_{i+1} &\leftarrow S_i \\ S_{i+1}.I &\leftarrow A \\ S_{i+1}.IS &\leftarrow S_{i+1}.IS \cup \{A\} \\ S_{i+1}.G &\leftarrow \textbf{True} \end{aligned} \tag{B.5}$$

The state variable, IS, is the set of all speculative intervals leading to state S.

The **guess** primitive initially returns a value of **True**, as recorded in $S_{i+1}.G$. If rollback occurs, execution resumes by resetting the PC to the point where **guess** was called, and returning a G value of **False**. Thus the effect of the **guess** primitive on the program counter is merely to increment it to the next operation.

The execution sequence associated with P is updated as follows:

$$H_P \leftarrow H_P S_{i+1} \tag{B.6}$$

B.1.2 Affirm

The **affirm(X)** primitive asserts that the optimistic assumption associated with X has been determined to be true. There are two cases. In the first case, the **affirm(X)** was executed in process P in state S_i where $S_i.I = \emptyset$. Therefore, the execution of the **affirm(X)** cannot be undone (**definite affirm**). Thus, all the intervals that depend on X can try to decide on whether to become definite or not as follows:

$$\begin{aligned}
&\text{If } S_i.I = \emptyset \text{ then } \forall B \in X.DOM: \\
&\quad S \leftarrow last(H_{B.PID}), \\
&\quad\quad ((S.IS).B).IDO \leftarrow B.IDO \backslash \{X\}, \\
&\quad\quad H_{B.PID} \leftarrow H_{B.PID} S && \text{(B.7)} \\
&\quad \text{If } B.IDO = \emptyset \text{ then } \textbf{finalize(B)} && \text{(B.8)} \\
&\quad X.DOM \leftarrow X.DOM \backslash \{B\} && \text{(B.9)}
\end{aligned}$$

finalize(B) transforms B from a speculative to a definite interval. **finalize** is not a part of the user's programming model, and is just used here as a shorthand notation for the effects described in Section B.1.5.

In the second case, an **affirm(X)** was executed in a process P in state S_i where $S_i.I \neq \emptyset$. Therefore interval A may be rolled back, thus the execution of **affirm(X)** may be undone. This is called a **speculative affirm**. The interval A records the set of assumption identifiers that it has affirmed in the set $A.IHA$, and applies them when A is finalized, as follows: [2]

[2]This doesn't even handle simple self-affirm.

$$\begin{aligned}&\text{If } S_i.I \neq \emptyset \text{ then}\\ &\quad S_{i+1} \leftarrow last(H_P),\\ &\quad\quad ((S_{i+1}.IS).A).IHA \leftarrow A.IHA \cup \{X\},\\ &\quad\quad H_P \leftarrow H_P S_{i+1}\end{aligned} \tag{B.10}$$

Only one **affirm** or **deny** primitive may be applied to a given assumption identifier. Multiple **affirm** primitives are redundant; once affirmed, it's affirmed. Similarly, multiple **deny** primitives are redundant. Conflicting **affirm** and **deny** primitives have no meaning: An assumption cannot be both true and false. To eliminate this kind of confusing and conflicting notation, more than one **affirm** or **deny** primitive applied to a single assumption identifier, in any combination, is a user error, and the meaning is undefined.

B.1.3 Deny

The **deny(X)** primitive asserts that the optimistic assumption associated with X has been determined to be false. There are two cases. In the first case, the **deny(X)** was executed in process P in state S_i of interval A where $S_i.I = \emptyset$ or $X \in A.IDO$. This implies that P is not dependent on any other assumption identifiers except perhaps X. Therefore, the execution of the **deny(X)** cannot be undone by another process (**definite deny**). Thus, all the intervals that depend on X must roll back. This is expressed as follows:

$$\text{If } S_i.I = \emptyset \vee X \in A.IDO \quad \text{then} \quad \forall B \in X.DOM : \textbf{rollback(B)} \tag{B.11}$$

In the second case, a **deny(X)** in an interval A dependent on other assumption identifiers other than X implies that A may be rolled back; thus **deny(X)** may be undone. This is called a **speculative deny**. The set IHD records assumption identifiers that have been denied, to be applied when the interval is made definite.

$$\begin{aligned}&\text{If } S_i.I \neq \emptyset \wedge X \notin A.IDO \text{ then}\\ &\quad S_{i+1} \leftarrow last(H_P),\\ &\quad\quad ((S_{i+1}.IS).A).IHD \leftarrow A.IHD \cup \{X\},\\ &\quad\quad H_P \leftarrow H_P S_{i+1}\end{aligned} \tag{B.12}$$

deny(X) becomes definite when the interval A is made definite, i.e., when $A.IDO = \emptyset$.

B.1.4 Free_of

The execution of a **free_of(X)** primitive in process P in state S_i of interval A means that interval A must not depend on the optimistic assumption associated with the assumption identifier X. The execution of **free_of(X)** in an interval A inspects the $A.DOM$ set for the assumption identifier X. If X is not found, then the requirement is satisfied and the optimistic assumption associated with X has been confirmed: the specified ordering constraint was not violated, and the equivalent of an **affirm(X)** is executed. However, if the assumption identifier X was found in $A.DOM$, then the ordering constraint associated with X was violated, and the equivalent of a **deny(X)** is executed. The formal specification is as follows:

$$\text{If } S_i.I = \emptyset \text{ then definite } \textbf{affirm(X)} \tag{B.13}$$

$$\text{Else if } X \notin A.DOM \text{ then speculative } \textbf{affirm(X)} \tag{B.14}$$

$$\text{Else speculative } \textbf{deny(X)} \tag{B.15}$$

Like **affirm** and **deny**, **free_of** "consumes" it's argument: it is an error for more than one **affirm**, **deny**, or **free_of** primitive to be applied to the same assumption identifier.

B.1.5 Finalize

Finalizing interval A makes A a permanent part of the history $H_{A.PID}$ by transforming A from a speculative to a definite interval. Finalizing A has the precondition:

$$A.IDO = \emptyset \tag{B.16}$$

The effect of **finalize(A)** on process $A.PID$ is as follows:

$$\begin{aligned} S &\leftarrow last(H_{A.PID}), \\ S.IS &\leftarrow S.IS \backslash \{A\}, \\ H_{A.PID} &\leftarrow H_{A.PID} S \end{aligned} \tag{B.17}$$

$$\forall X \in A.IHA, \forall B \in X.DOM$$
$$S \leftarrow last(H_{A.PID}),$$
$$S.IS.B.IDO \leftarrow S.IS.B.IDO \backslash \{X\},$$
$$H_{A.PID} \leftarrow H_{A.PID} S \tag{B.18}$$
$$\text{If } B.IDO = \emptyset \text{ then } \textbf{finalize(B)} \tag{B.19}$$
$$X.DOM \leftarrow X.DOM \backslash \{B\} \tag{B.20}$$

$$\forall X \in A.IHD, \forall B \in X.DOM \quad \textbf{rollback(B)} \tag{B.21}$$

$$\text{If } last(H_{A.PID}).IS = \emptyset \text{ then}$$
$$S \leftarrow last(H_{A.PID}),$$
$$S.I \leftarrow \emptyset,$$
$$H_{A.PID} \leftarrow H_{A.PID} S \tag{B.22}$$

Speculative execution of HOPE primitives now become definite. In particular, the speculative execution of any **affirm(X)** and **deny(Y)** primitives now become definite, and so all of the intervals dependent on the assumption identifiers $X \in A.IHA$ lose their dependency on X as specified by Equation B.18, and all intervals dependent on assumption identifiers $Y \in A.IHD$ are rolled back by Equation B.21.

If A was the last and only interval in the history of process $A.PID$ to date, then $A.PID$ no longer has a current interval, and becomes definite as specified by Equation B.22.

B.1.6 Rollback

Rolling back interval A truncates the history of $A.PID$ immediately before the start of A, and all subsequent states in the history are discarded. Process $A.PID$ then resumes from the **guess** primitive, but returning **False** instead of **True**.

$$H_{A.PID} \leftarrow Del(H_{A.PID}, A),$$
$$S \leftarrow A.PS,$$
$$S.G \leftarrow \textbf{False},$$
$$H_{A.PID} \leftarrow H_{A.PID} S \tag{B.23}$$

In addition, speculative execution of **affirm**, **deny** and **free_of** primitives must be undone. Speculative execution of **affirm** and **deny** primitives are trivially undone because they are never applied: They die with the interval inside the IHA and IHD sets. Speculative execution of the **free_of** primitive translates into speculative execution of **affirm** and **deny** primitives, and so **free_of** also does not require undoing.

The following Lemma shows that for all intervals and assumption identifiers, if an interval A records that it is dependent on an assumption identifier X, then the assumption identifier X also records that A is dependent on X. This Lemma is used in proving subsequent theorems.

Lemma B.1 *For all intervals A and assumption identifiers X, $X \in A.IDO$ if and only if $A \in X.DOM$.*

Proof: This proof is based on demonstrating that if an assumption identifier X is inserted into $A.IDO$, then A is also inserted into $X.DOM$, and that if X is taken out of $A.IDO$, then A is also taken out of $X.DOM$.

Equation B.3 is the only operation where assumption identifiers are inserted into IDO sets. Equation B.3 inserts assumption identifier X into $A.IDO$, and symmetrically Equation B.4 inserts A into $X.DOM$. Thus, in all cases, if an assumption identifier is inserted into an interval's IDO set, then the interval is inserted into the assumption identifiers DOM set.

Similarly, Equations B.7 and B.18 are the only two operations that remove assumption identifiers from IDO sets. In both cases, assumption identifier X is removed from $B.IDO$, and interval B is symmetrically removed from $X.DOM$. Therefore, since in all cases, insertion into an IDO set symmetrically necessitates insertion into a DOM set, and removal from an IDO set symmetrically necessitates removal from a DOM set, we have for all intervals A and assumption identifiers X, $X \in A.IDO$ if and only if $A \in X.DOM$. □

The rollback of an interval implies that all intervals occurring after A also are rolled back. Therefore **rollback** truncates history. This is shown by the following theorem:

Theorem B.1 *If an interval A is rolled back in a process P then for all intervals B, where B occurs after A in H_P, B is also rolled back.*

Proof: We will first show that for all intervals that occur after interval A in H_P, $A.IDO \subset B.IDO$. There are two cases to consider that correspond to IDO updates. The first corresponds to the initial creation of an interval, and the second corresponds to the effects of execution of **affirm** primitives.

This is done by induction on the number of intervals that occur after interval A. By definition, when the first interval B following immediately after A is created, $B.IDO = A.IDO \cup \{X\}$ where X is the assumption identifier in the **guess** that created B. We can immediately conclude that $A.IDO \subset B.IDO$. Assume that the first k intervals to occur after A in H_P are such that $A.IDO$ is a subset of the IDO sets associated with each of these intervals. Let B' be the k^{th} interval and B be the $k+1^{th}$ interval. By definition, when interval B is created, its associated IDO set is $B'.IDO \cup \{X\}$. Thus $B'.IDO \subset B.IDO$. By the inductive hypothesis, we have that $A.IDO \subset B'.IDO$, thus we can immediately conclude that $A.IDO \subset B.IDO$.

However, the execution of **affirm** primitives also cause changes in interval's IDO sets, thus we need to show that it is always the case that $A.IDO \subset B.IDO$. By Lemma B.1, if an assumption identifier X is in both $A.IDO$ and $B.IDO$, then both A and B are in $X.DOM$. Thus, all of the other operations that affect IDO sets (Equations B.7 and B.18), which always use DOM sets to select the interval IDO sets to manipulate, will affect A and all of the intervals that follow A in H_P in the same way. Thus it is always the case that $A.IDO \subset B.IDO$ for all intervals B that follow A in H_P.

By definition, the only primitive that can cause A to be rolled back is a definite **deny** of some assumption identifier $X \in A.IDO$. Since $A.IDO \subset B.IDO$ for all B that follow A in H_P, $X \in B.IDO$ for all B that succeed A in $H_{A.PID}$. Therefore, the same definite **deny(X)** also will cause all following intervals to be rolled back. Therefore, rollback of A will result in a truncation of H_P. □

The following Theorem shows that once an interval's IDO set becomes empty, it can never be rolled back.

Theorem B.2 *For any interval A, if $A.IDO = \emptyset$ then* **rollback(A)** *will never occur.*

Proof: From Equations B.11 and B.21, an interval A can only be rolled back if $A \in X.DOM$ for some assumption identifier X and a definite **deny(X)** has occurred. But from Theorem B.1, if $A.IDO = \emptyset$, then there does not exist an assumption identifier X such that $A \in X.DOM$. Therefore, no **deny(X)** can affect A, and A cannot be rolled back. □

B.2 Semantic Implications

The preceding specifications describe the meaning of the HOPE primitives in detail, but we would like to have some assurance that they will behave in a way that the programmer would intuitively expect. If **affirm** is asserted for all of the assumption identifiers that an interval depends upon, then the programmer expects that the interval will be made definite. Conversely, the programmer expects that the interval will be made definite only if **affirm(X)** is asserted for all of the assumption identifiers X that the interval depends on. The theorems presented in this section prove that these expectations are preserved by the definitions of the HOPE primitives.

Lemmas B.2 and B.3 are combined together in Theorem B.3 to show that if all of the optimistic assumptions associated with the assumption identifiers that an interval B depends on are confirmed to be true by intervals that are themselves eventually made definite, then B will be made definite. Lemma B.2 below shows that the effects of a speculative **affirm** primitive are identical to the effects of a definite **affirm** if the asserting interval is eventually made definite.

Lemma B.2 Affirm Transitivity: *Let B be an interval that depends on an assumption identifier X. The effect of executing* **affirm(X)** *within a speculative interval A upon $B.IDO$ and $X.DOM$, followed by A eventually being made definite, is the same as the effect of a definite* **affirm(X)**.

Proof: Equations B.7 and B.9 define the effect of a definite **affirm(X)** as removing X from $B.IDO$, and removing interval B from $X.DOM$.

Let A be a speculative interval that executes **affirm(X)** for some $X \in B.IDO$. Equation B.10 records that assumption identifier X was affirmed by interval A. When A is subsequently finalized, Equations B.18 and B.20 perform identical manipulations on all of the intervals $B \in X.DOM$, removing X from $B.IDO$, and removing B from $X.DOM$. Thus the impact of A executing **affirm(X)** followed by **finalize(A)** is the same as a definite **affirm(X)**. □

Lemma B.3 proves that if definite **affirm(X)** is executed on all of the assumption identifiers X that an interval B depends on, then B will become definite.

Lemma B.3 *For any interval B, if definite* **affirm(X)** *is applied to all assumption identifiers $X \in B.IDO$, then* **finalize(B)** *will result.*

Proof: Let B be an interval with an initial set of assumption identifiers $\gamma = B.IDO$.

Equation B.7 expresses that for each $X \in \gamma$, each definite **affirm(X)** will remove X from $B.IDO$. Equation B.9 will induce **finalize(A)** when the last assumption identifier in γ is affirmed. □

Using Lemmas B.2 and B.3, we can conclude that if all of the assumption identifiers that an interval B depends on are affirmed by intervals that eventually become definite, then B will become definite.

Theorem B.3 *For any interval B, if* **affirm(X)** *is applied to all assumption identifiers $X \in B.IDO$ by intervals that eventually become definite, then* **finalize(B)** *will result.*

Proof: Lemma B.3 shows that if all of the **affirm(X)** primitives are executed by definite intervals, then **finalize(B)** will result. Lemma B.2 shows that speculative **affirm** primitives that are eventually made definite have the same effect as definite **affirm** primitives, and so **finalize(B)** will result in either case. □

Theorem B.4 shows that **finalize(B)** will occur if and only if **affirm(X)** is executed on all of the assumption identifiers X that interval B depends on.

Theorem B.4 *For all intervals B,* **finalize(B)** *occurs if and only if* **affirm(X)** *is applied to all of the assumption identifiers $X \in B.IDO$ by intervals that eventually become definite.*

Proof: From Theorem B.3, we have the case that if **affirm(X)** is applied to all assumption identifiers $X \in B.IDO$, then **finalize(B)** will result.

We prove that **finalize(B)** implies that **affirm(X)** has been applied to all assumption identifiers $X \in B.IDO$ using proof by contradiction. Assume that **finalize(B)** has occurred, and that there exists an assumption identifier $X \in B.IDO$ that has not had **affirm(X)** executed.

Since **affirm(X)** has not occurred, no operation will have removed X from $B.IDO$. Therefore $B.IDO \neq \emptyset$, and by Equation B.16, **finalize(B)** cannot have occurred. This contradicts the assumption that **finalize(B)** has occurred.

Therefore, for all intervals B, **finalize(B)** occurs if and only if **affirm(X)** is applied to all of the assumption identifiers $X \in B.IDO$ by intervals that eventually become definite. □

Finally, Theorem B.5 shows that if an interval A executes **free_of(X)**, then $A.IDO$ does in fact remain free of X.

Theorem B.5 *For any interval A that executes* **free_of(X)**, *then either interval A does not depend on on assumption identifier X, or interval A is rolled back.*

Proof: We use proof by contradiction. Let A be an interval such that $X \in A.IDO$, and let the statement **free_of(X)** be executed within interval A. By inspection, Equation B.15 will execute **deny(X)**, which will in turn apply Equation B.11, which will roll back interval A. Thus if A depends on X and executes **free_of(X)**, then A is rolled back.

If A does not depend on X at the time it executes **free_of(X)**, then A can never become dependent on X. To become dependent on X, A would have to depend on some other AID Y that was speculatively affirmed by some interval B that depended on X. However, since the speculative execution of **free_of(X)** resulted in a speculative **affirm(X)**, there cannot exist another interval B that depends on X, because B's dependence on X will have been replaced with a dependence on $A.IDO$. Thus A can never depend on X. □

B.3 Summary

This appendix has formally defined the meaning of the HOPE primitives using an operational semantics, in such a way as to allow the full rollback of the speculative execution of **affirm** primitives. The operations are defined on **intervals** in the execution history of user processes, and **assumption identifiers** that identify particular optimistic assumptions.

HOPE tracks which intervals depend on which optimistic assumptions by maintaining dependency sets in the intervals and assumption identifiers. The dependency sets of an interval are as follows:

IDO	I Depend On
IHA	I Have Affirmed
IHD	I Have Denied

The dependency set of an assumption identifier is as follows:

DOM Depends On Me set of intervals contingent on this assumption identifier

The HOPE operations are specified in terms of manipulations on the dependency sets, and specific actions (**finalize** and **rollback**) that are taken when certain properties of these sets occur.

REFERENCES

[1] Jonathan R. Agre and Divyakant Agrawal. Recovering From Process Failures in the Time Warp Mechanism. In *8th Symposium on Reliable Distributed Systems*, pages 53–61, Seattle, WA, October 1989.

[2] G. M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proc. AFIPS 1967 Spring Joint Computer Conference 30*, pages 483–485, Atlantic City, New Jersey, April 1967.

[3] American National Standards Institute, Inc. *Programming Language - C, ANSI Standard X3.159.* American National Standards Institute, Inc., 1989.

[4] J.S. Auerbach, D.F. Bacon, A.P. Goldberg, G.S. Goldszmidt, M.T. Kennedy, A.R. Lowry, J.R. Russell, W. Silverman, R.E. Strom, D.M. Yellin, and S.A. Yemini. High-Level Language Support for Programming Distributed Systems. In *Proceedings of the 1991 CAS Conference*, pages 173–196, Toronto, Ontario, November 1991.

[5] David F. Bacon and Robert E. Strom. Optimistic Parallelization of Communicating Sequential Processes. In *Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, April 1991.

[6] Rajive L. Bagrodia and Wen-Toh Liao. Maisie: A Language for the Design of Efficient Discrete-Event Simulations. *IEEE Transactions on Software Engineering*, 20(4):225–238, April 1994.

[7] D. J. Bernstein. Poor Man's Checkpointer. Public Domain Software, 1990.

[8] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, Massachusetts, 1987.

[9] R.G. Bubenik. *Optimistic Computation.* PhD thesis, Rice University, May 1990.

[10] Rick Bubenik and Willy Zwaenepoel. Semantics of Optimistic Computation. In *10th International Conference on Distributed Computing Systems*, pages 20–27, 1990.

[11] Rick Bubenik and Willy Zwaenepoel. Optimistic Make. *IEEE Transactions on Computers*, 41(2):207–217, February 1992.

[12] F. J. Burkowski, C. L. A. Clarke, Crispin Cowan, and G. J. Vreugdenhil. Architectural Support for Lightweight Tasking in the Sylvan Multiprocessor System. In *Symposium on Experience with Distributed and Multiprocessor Systems (SEDMS II)*, pages 165–184, Atlanta, Georgia, March 1991.

[13] N. Carriero and D. Gelernter. Linda in Context. *Communications of the ACM*, 31(4):444–458, April 1989.

[14] NASA Ames Research Center. Network queueing system.

[15] Crispin Cowan. Optimistic Replication in HOPE. In *Proceedings of the 1992 CAS Conference*, pages 269–282, Toronto, Ontario, November 1992.

[16] Crispin Cowan and Hanan Lutfiyya. Formal Semantics for Expressing Optimism. Report 414, Computer Science Department, UWO, London, Ontario, March 1994. Submitted for review, and available via FTP from `ftp://ftp.csd.uwo.ca/pub/csd-technical-reports/414/report414.ps.Z`.

[17] Crispin Cowan, Hanan Lutfiyya, and Mike Bauer. Increasing Concurrency Through Optimism: A Reason for HOPE. In *Proceedings of the 1994 ACM Computer Science Conference*, pages 218–225, Phoenix, Arizona, March 1994.

[18] J. de Kleer. A Assumption-based Truth Maintenance System. *Artificial Intelligence*, 28:127–162, 1986.

[19] J. de Kleer. Extending the ATMS. *Artificial Intelligence*, 28:163–196, 1986.

[20] J. Doyle. A Truth Maintenance System. *Artificial Intelligence*, 12:231–272, 1979.

[21] A. Ege and C.A. Ellis. Design and Implementation of GORDION, an Object Base Management System. In *Procedings of the International Conference on Data Engineering*, pages 226–234, Los Angeles, California, February 3-5 1987.

[22] Richard M. Fujimoto. The Virtual Time Machine. *Computer Architecture News*, 19(1):35–44, March 1991. Re-printed from the Symposium on Parallel Algorithms and Architectures 1990.

[23] D. Gifford. Weighted Voting for Replicated Data. In *7th ACM Symposium on Operating System Principles*, pages 150–162, 1979.

[24] Arthur P. Goldberg. *Optimistic Algorithms for Distributed Transparent Process Replication.* PhD thesis, University of California at Los Angeles, 1991. (UCLA Tech. Report CSD-910050).

[25] I. Greif, R. Seliger, and W. Weihl. Atomic Data Abstractions in a Distributed collaborative Editing System. In *13th Annual ACM Symp. on Principles of Programming Languages*, pages 160–172, January 13-15 1986.

[26] Anurag Gupta, Ian F. Akyildiz, and Richard M. Fujimoto. Performance Analysis of Time Warp with Multiple Homogeneous Processors. *IEEE Transactions on Software Engineering*, 17(10):1013–1027, October 1991.

[27] Reed Hastings and Bob Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In *Proceedings of the Winter USENIX Conference*, 1992.

[28] Matthew Hennessy. *The Semantics of Programming Languages.* John Wiley & Sons Ltd., Baffins Lane, Chichester West Sussex PO19 1UD, England, first edition, 1990.

[29] Maurice Herlihy. Apologizing Versus Asking Permission: Optimistic Concurrency Control for Abstract Data Types. *ACM Transactions on Database Systems*, 15(1):96–124, March 1990.

[30] International Standards Orgainzation. *Ada 9X Reference Manual (ANSI/ISO DIS 8652 Draft International Standard)*. Intermetrics, Inc., Cambridge, Massaachusetts, June 1994.

[31] D. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 3(7):404–425, July 1985.

[32] D. Jefferson and A. Motro. The Time Warp Mechanism for Database Concurrency Control. Report Technical Report TR-84-302, University of Southern California, January 1984.

[33] D.B. Johnson and W. Zwaenepoel. Recovery in Distributed Systems using Optimistic Message Logging and Checkpointing. *J. Algorithms*, 11(3):462–491, September 1990.

[34] Brent A. Kingsbury and John T. Kline. Job and Process Recovery in a UNIX-based Operating System. In *Proceedings of the Winter USENIX Conference*, pages 355–364, 1989.

[35] Puneet Kumar. Coping with Conflicts in an Optimistically Replicated File System. In *1990 Workshop on the Management of Replicated Data*, pages 60–64, Houston, TX, November 1990.

[36] H.T. Kung and John T. Robinson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[37] L. Lamont, G. Henderson, and N.D. Georgannas. A Multimedia Real-time Conferencing System: Architecture and Implementation. In *Proceedings of the 1993 CAS Conference*, pages 64–71, Toronto, Ontario, October 1993.

[38] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[39] Jonathan I. Leivent and Ronald J. Watro. Mathematical Foundations for Time Warp Systems. *ACM Transactions on Programming Languages and Systems*, 15(5):771–794, November 1993.

[40] Juan Leon. Fail-Safe PVM. TR CMU-CS-93-124, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA, 1993.

[41] B.T. Lewis and J.D. Hodges. Shared Books: Collaborative Publications Management for an Office Information System. In *Procedings of the Conference on Office Information Systems*, pages 197–204, Palo Alto, California, March 23-25 1988.

[42] Yi-Bing Lin and Edward D. Lazowska. Optimality Considerations of the 'Time Warp' Parallel Simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 29–34, San Diego, CA, January 1990.

[43] Michael Litzkow and Marvin Solomon. Supporting Checkpointing and Process Migration Outside the UNIX Kernel. In *Proceedings of the Winter USENIX Conference*, 1992.

[44] Hanan Lutfiyya and Crispin Cowan. The Application-Oriented Fault Tolerance Paradigm with Rollback. To be submitted for review.

[45] M. Marcotty, H.F. Ledgard, and G.V. Bochmann. A Sampler of Formal Definitions. *Computing Surveys*, 8(2):191–276, 1976.

[46] Sun Microsystems. TMPFS - Memory Based File System. section 4 of the SunOS 4.1 manual.

[47] C. Papadimitriou. The Serializability of Concurrent Updates. *Journal of the ACM*, 24(4):631–653, October 1979.

[48] J. Peterson and A. Silberschatz. *Operating System Concepts*. Addison-Wesley, Reading, Massachusetts, 1983.

[49] Gerald J. Popek, Richard G. Guy, Jr. Thomas W. Page, and John S. Heidemann. Replication in Ficus Distribued Files Systems. In *1990 Workshop on the Management of Replicated Data*, pages 5–10, Houston, TX, November 1990.

[50] B. R. Preiss, W. M. Loucks, and V. C. Hamacher. A Unified Modelling Methodology for Performance Evaluation of Distributed Discrete Event Simulation Mechanisms. In *Winter Simulation Conference*, 1988.

[51] Jim Pruyne. Process Checkpointing. Personal Communications, May 1994. University of Wisconsin, Madison.

[52] William Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communications of the ACM*, 33(6), June 1990.

[53] Erhard Rahm and Alexander Thomasian. Distributed Optimistic Concurrency control for High Performance Transaction Processing. In *International Conference on Databases, Parallel Architectures, and their Applications (PARBASE-90)*, pages 490–495, March 1990.

[54] Parameswaran Ramanathan and Kang G. Shin. Use of Common Time Base for Checkpointing and Rollback Recovery in a Distributed System. *IEEE Transactions on Software Engineering*, 19(6):571–583, June 1993.

[55] Peter Reiher, Richard Fujimoto, Steven Bellenot, and David Jefferson. Cancellation Strategies in Optimistic Execution Systems. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 112–121, San Diego, CA, January 1990.

[56] A. S. Spector, J. L. Eppinger, D. S. Daniels, J. J. Block R. Draves, D. Duchamp, R. F. Paush, and D. Thompson. High Performance Distributed Transaction Processing in a General Purpose Computing Environment. Report, Computer Science Department, CMU, Pitsburgh, PA, September 1987.

[57] R.E. Strom and S. Yemini. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.

[58] Rob Strom. The Need For Optimism in Hermes. Personal Communications, August 1991.

[59] Rob Strom and Shaula Yemini. Synthesizing Distributed and Parallel Programs through Optimistic Transformations. In Y. Yemini, editor, *Current Advances in Distrubuted Computing and Communications*, pages 234–256. Computer Science Press, Rockville, MD, 1987.

[60] Thomas Strothotte. *Temporal Constructs for an Algorithmic Language*. PhD thesis, McGill University, 1984.

[61] Thomas Strothotte. Temporal Constructs for an Algorithmic Language. Report SOCS-84.20, McGill University, Montreal, Quebec, December 1984.

[62] V.S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice & Experience*, 2(4):315–339, December 1990.

[63] Gerard Tel and Friedmann Mattern. Derivation of Distributed Termination Detection Algorithms from Garbage Collection Schemes. *ACM Transactions on Programming Languages and Systems*, 15(1):1–35, January 1993.

[64] P. Triantafillou and D.J. Taylor. A New Paradigm for High Availability and Efficiency in Replicated and Distributed Databases. In *2nd IEEE Symposium on Parallel and Distributed Processing*, pages 136–143, December 1990.

[65] Brian Unger, John Cleary, Alan Dewar, and Zhong e Xiao. Multi-lingual Optimistic Distributed Simulator. *Transactions of the Society for Computer Simulation*, 7(2):121–151, June 1990.

[66] Kun-Lung Wu and W. Kent Fuchs. Twin-page Storage Management for Rapid Transaction-undo Recovery. In *14th Annual International Computer Software and Applications conference (COMPSAC 90)*, pages 5–10, Houston, TX, November 1990.

[67] Bennet Yee and David Applegate. Save World. Public Domain Software, available via FTP from `ftp://Play.Trust.CS.CMU.Edu/usr/ftpguest/pub/save_world.tar.Z`., 1993.

[68] Philip Yu and Daniel M. Dias. Analysis of Hybrid Concurrency Control Schemes for a High Data Contention Environment. *IEEE Transactions on Software Engineering*, 18(2):118–129, February 1992.