JPL D-9516

Technology and Applications Programs

Time Warp Operating System Version 2.7
Internals Manual

February 1992

Prepared for

U.S. Army Model Improvement and Study Management Agency
U.S. Army Concepts Analysis Agency

Through an Agreement with
National Aeronautics and Space Administration

by

JPL

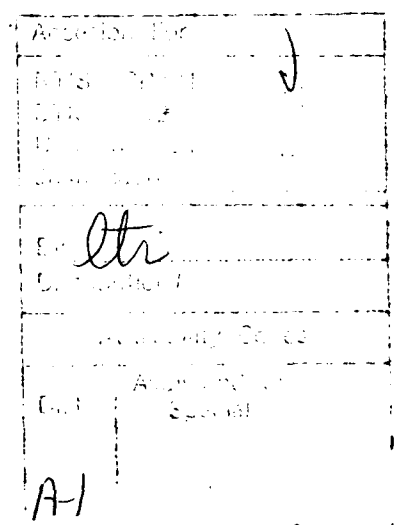
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

JPL D-9516

Technology and Applications Programs

Time Warp Operating System Version 2.7
Internals Manual

February 1992



Prepared for

U.S. Army Model Improvement and Study Management Agency
U.S. Army Concepts Analysis Agency

Through an Agreement with
National Aeronautics and Space Administration

by

JPL

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

The research described in this publication was carried out by the Jet Propulsion Laboratory, California Institute of Technology, and was sponsored by the United States Army Model Improvement and Study Management Agency and the Concepts Analysis Agency through an agreement with the National Aeronautics and Space Administration, under contract NAS7-918.

PREFACE

This manual describes internal details of the Jet Propulsion Laboratory (JPL) implementation of the Time Warp Operating System (TWOS) version 2.7. It is meant to complement the TWOS User's Manual, JPL document number D-6493 Rev. C.

This manual is meant for the use of those trying to understand the internal workings of TWOS. Users of TWOS should generally consult the TWOS User's Manual, instead. This manual is not meant to serve as a C language tutorial, nor as an introduction to UNIX systems, nor as a manual for the BBN GP1000 parallel processor or Sun workstations. The manual is written assuming that its readers have a working knowledge of and familiarity with the basics of computer programming; with basic systems programming and data structures; with the C programming language; and with the hardware on which they intend to run TWOS. This manual is not meant to serve as an introduction to parallel programming, systems programming, or operating systems.

The Time Warp Group at JPL would appreciate any questions, comments, and suggestions, Send all correspondence to:

Time Warp Operating System
High Performance Computing Group
Mail Stop 525-3660
Jet Propulsion Laboratory
4800 Oak Grove Drive
Pasadena, CA 91109

Trademarks

GP-1000 is a trademark of BBN Advanced Computers, Incorporated.

Excel is a trademark of Microsoft Corporation.

UNIX is a registered trademark of AT&T Bell Laboratories.

Ethernet is a trademark of the Xerox Corporation.

Sun-3 and Sun-4 are trademarks of Sun Microsystems, Inc.

Chapter 1: Overall System Structure

1.1 Introduction

The Time Warp Operating System (TWOS) is an implementation of the Time Warp synchronization method proposed by David Jefferson. In addition, it serves as an actual platform for running discrete event simulations. The code comprising TWOS can be divided into several different sections.

Figure 1 graphically describes the Time Warp Operating System. Starting from the center of the figure and working out, at the very core of the system is another operating system. TWOS typically relies on an existing operating system to furnish some very basic services. This existing operating system is referred to as the **Base OS** in figure 1-1. The existing operating system varies depending on the hardware TWOS is running on. It is Unix on the Sun workstations, Chrysalis or Mach on the Butterfly, and Mercury on the Mark 3 Hypercube. The base OS could be an entirely new operating system, written to meet the special needs of TWOS, but, to this point, existing systems have been used, instead. The base OS's used for TWOS on various platforms are not discussed in detail in this manual, as they are well covered in their own manuals. Appendix G discusses the interface between one such OS, Mach, and TWOS.

The next layer out is the **Standard OS Services** layer. This layer of code provides certain services to TWOS that are common to most operating systems. They include context switching, simple memory management, queueing primitives, fast copy routines, object location, and interrupt handling. These services could usually be provided by the underlying operating system in the **Base OS** layer, but TWOS needs to have them performed in a somewhat different way than most systems. Therefore, they are implemented in a manner suitable for TWOS. The TWOS code providing these services is discussed in Chapters 17-24 of this manual.

The third layer is **Basic TW**. This layer is the implementation of the fundamental code that makes Time Warp synchronization work. It includes scheduling code and support for rollback, message cancellation and commitment. This is the code that is typically thought of as "Time Warp" by those reading theoretical papers on the subject. This code is discussed in Chapters 1-8 of the internals manual.

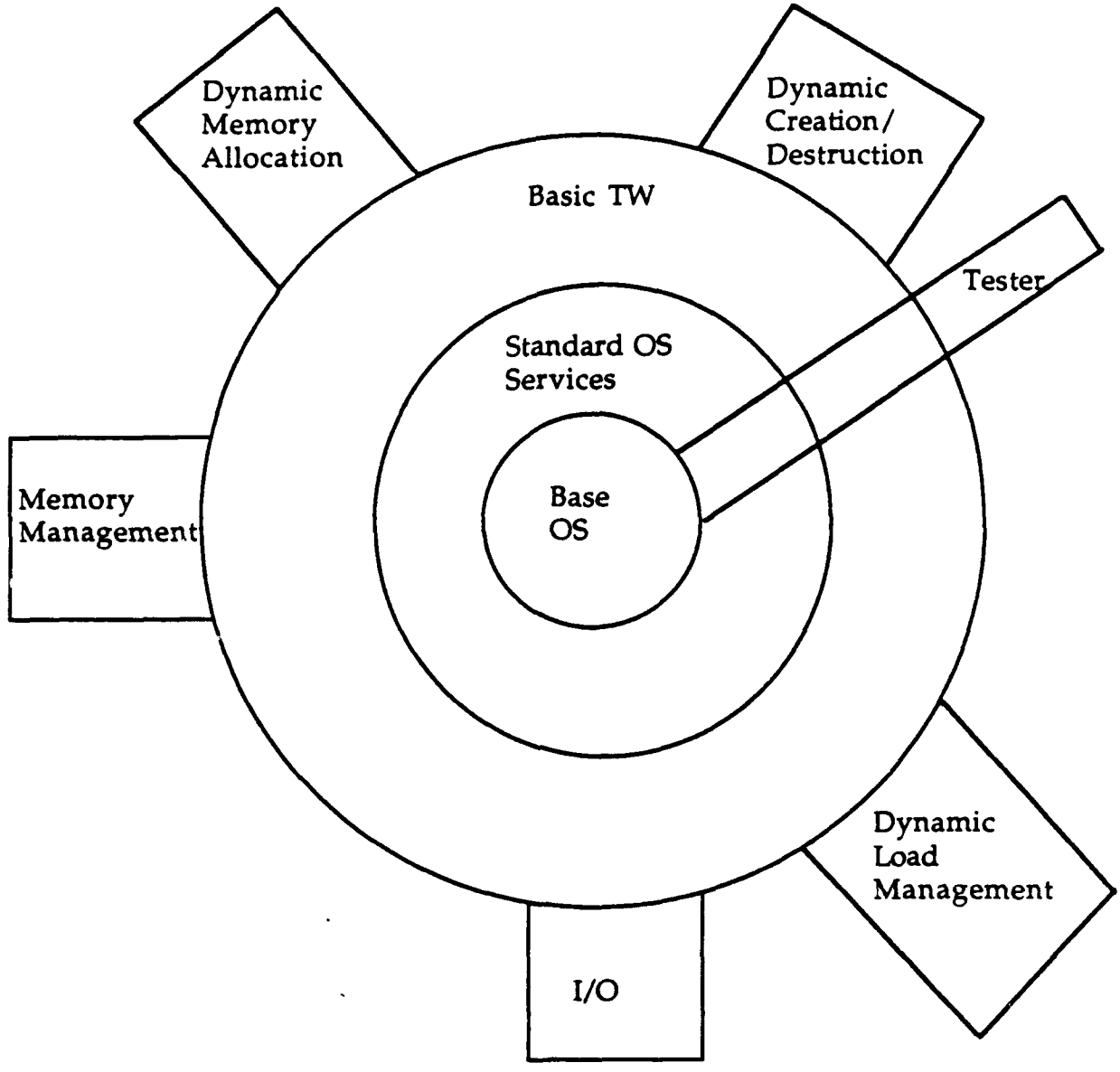


Figure 1-1: The Structure of TWOS

Surrounding the **Basic TW** layer is a set of additional services, above and beyond the basic Time Warp synchronization code. Correct Time Warp programs can be written and run without any of these services, but they are required if one wants to do real work on top of the system. However, not all programs use all of these services. The services include dynamic creation and destruction of objects, dynamic memory allocation, message sendback, I/O, dynamic load management, and critical path computation. This code is discussed in Chapters 9-16 of the internals manual, except for the material on I/O, which is covered in Chapter 20.

Finally, there is the **tester**. The tester is not really a functional part of TWOS. It serves two purposes. Primarily, it is used for debugging purposes. It allows a knowledgeable user to probe various internal data structures in the middle of the run. Its secondary purpose is to serve as a command interpreter, in which role it reads the configuration file and parses the commands found there, passing them along to TWOS. (The tester performs this second function only because its primary function required all code necessary to parse user commands. Rather than duplicate that code for the purpose of reading configuration files, the tester serves double duty.) The tester is discussed in Chapter 18 of this manual.

The remainder of this chapter will discuss each of these parts of TWOS in further detail. Later chapters of the TWOS Internals manual will give complete details on how each of these parts of Time Warp works, and why they are implemented as they are.

1.2 The Base OS

TWOS performs most operating system functions on its own. However, there is one fundamental service that is usually provided by some other operating system. That service is message routing. TWOS is meant to run on parallel hardware. Information must be sent from one node of the hardware to another all the time. Some of this information is in the form of user-level messages. Some of it is systems messages. In either case, information must be transported from one node to another.

Every different architecture TWOS runs on handles the passing of information from one node to another in a different way. On the Butterfly, information is passed through a switch that simulates shared memory. On the Mark 3 Hypercube, information is routed along a limited number of communications paths, passing from one node to another until it reaches its destination. On a network of Sun machines, information is sent out on a local area network, usually an ethernet. The code necessary to make the information move is thus highly dependent on architecture.

On any parallel or distributed system, the designers of the system must have made some provision for moving information from node to node. Typically, they have provided software that performs this function. Therefore, some message passing system usually exists before the port of TWOS to that hardware begins. To the extent that this code is reusable, rewriting it inside of TWOS is wasteful.

A second point in favor of using the base operating system is the complexity of message passing software. TWOS requires a relatively modest message passing capability. Any message presented to the message passing system must be delivered to the requested node reliably, without corruption of the

contents. Messages need not be delivered in the order presented to the message passing system, nor is there a hard deadline on how quickly they must travel, nor need every message take approximately the same amount of time. Despite the relatively primitive message handling support required, the complexity of code required to provide this service is great. The issues of writing the code are fairly well understood, but the amount of code is still substantial. Also, message passing code offers many possibilities for rarely-occurring bugs.

With only small exceptions, there is nothing Time Warp specific about message passing at this level. At the level of node-to-node transportation, the system need not worry about virtual time issues, nor about message cancellation. TWOS probably performs better if messages are given priority by order of their virtual times, but will work correctly with any reasonable ordering. Performance might also improve if messages and antimessages could cancel in internal message passing queues, but, again, doing so is not necessary for correctness. In some cases, giving priority to TWOS system messages over TWOS user messages is important, and in other cases negative messages must have priority over positive messages. But, other than these small exceptions, some of which can usually be handled with existing features of the existing message passing system, nothing about this functionality is any different under TWOS than under any other distributed system.

Thus, if TWOS did not rely on an underlying operating system to pass messages from node to node, a new message passing subsystem would have to be written for each new architecture TWOS was ported to. The code required would be voluminous, complex, and probably filled with bugs. On the other hand, existing systems already provide stable, efficient message passing systems. With only minor exceptions, they offer all of the functionality TWOS needs in its message passing system. The choice of whether to use the existing message passing system or writing a new one is thus easily made, in most cases.

TWOS sometimes uses other base OS services at very low levels. For instance, while TWOS has its own I/O facilities, many of those are built on top of I/O primitives provided by the base OS. TWOS' memory management system sometimes relies on the underlying memory management system, since TWOS itself does not handle virtual memory or page tables. Like message passing, these underlying OS services are used because they closely match what TWOS needs to do, anyway, thereby avoiding the complexity of correctly coding the services.

1.3 Standard OS Services

TWOS is required to perform a number of actions typical to almost all operating systems. For instance, it must allocate and deallocate memory. It must switch contexts to permit user jobs to run, or to return to the operating

system. It must support internal queues. It must copy data from one place to another. And, like most distributed systems, it must be able to map a user-meaningful object name to the object's physical location in the system.

The obvious question is, why not use the underlying operating system to provide these services, just as it provides message passing? The answer is that the underlying system either cannot provide the exact functionality that TWOS requires, or cannot provide it efficiently enough, or cannot give TWOS sufficient control over the results.

Another issue is that trapping to an underlying operating system is often quite expensive, in time. If the operation being performed is already fairly heavyweight, like moving a message over a network, the cost of trapping to the underlying operating system is dwarfed. If the operation being performed is not particularly expensive itself, however, then the cost of the trap may be much more than the cost of the actual work being done.

For instance, TWOS uses memory very extensively. The system is forever requesting memory for messages and states, and returning that memory to the pool of unused bytes. This operation must be very quick if TWOS is to run fast. Also, TWOS knows that it will make many memory requests of a certain size, those matching the length of message buffers in the system. TWOS can handle its own memory management much more efficiently than the underlying system can.

TWOS also needs complete control of context switches. It must arrange that parameters are properly placed, and context switches must be cheap. Given that they are being performed in certain stylized ways, TWOS can switch contexts more cheaply than the underlying system.

Queues are ubiquitous in TWOS. Every process has three of them attached, a queue of input messages, a queue of output messages, and a queue of states. The queueing discipline in TWOS is somewhat different than that of most systems, in that messages must be ordered by their virtual times, and, within a virtual time, by their selectors, and, within a selector, in an unspecified but deterministic way. The possibility of message cancellation between two identical messages of opposite signs is also not usually supported in existing queueing packages.

TWOS copies data frequently. The system must make a copy of every message an object sends out to save in the object's output queue. Events are presented copies of messages, rather than the originals, so that they may modify them without corrupting them for rollback purposes. Every event must be given a fresh copy of the state produced by the last event. Copying occurs many other places in the system. A TWOS port usually requires the writing of an assembly language copying routine to ensure that the operation

is fast enough. An assembly language data comparison routine is often written, for the same reason.

TWOS needs to translate user-level object names to physical locations. Within a node, the physical location is a pointer to the control block for the object. If the object is not located on the local node, then TWOS must be able to find out which node does host the object. Existing systems often either do not provide this functionality, or do not provide it at a reasonable cost, or do not provide it in a scalable manner. The dynamic load management capability in TWOS adds a number of complexities to the problem that many existing systems would not handle well. Thus, TWOS performs name mappings itself.

On any particular underlying system, one or more of these services might be sufficiently well matched to TWOS' needs that putting the service into TWOS wouldn't be necessary. However, experience shows that this set of services is usually not provided by underlying operating systems well enough to use their existing code. But there is nothing fundamentally specific to TWOS that requires these services to be part of the TWOS kernel.

1.4 Basic TW

For most people familiar with the Time Warp paradigm, Time Warp is all about scheduling, rollback, and message cancellation. That is the base of the idea. So these are the services found in the **basic TW** layer of TWOS. In conjunction with the less TWOS-specific services offered by the lower layers, this layer comprises the minimal base for running a Time Warp system. Provided users don't need special services, and provided the system has access to infinite memory, no more is necessary. TWOS could perform correctly on certain types of applications with no other capabilities.

TWOS scheduling is by lowest virtual time first, independently calculated on each node of the system. Each node maintains a queue of objects, ordered by the next virtual time at which they have work to do. Whenever a node completes a piece of work, it selects the object at the head of this queue to serve next, without interacting with any other node. TWOS must maintain this queue in the proper order. As new messages arrive, rollbacks may change the virtual times at which objects need to execute, causing the scheduler queue to be reordered. Also, when an object completes execution, the system must determine when it should execute next, and move it to its proper position in the queue.

Whenever an out-of-order message arrives, or whenever a message is cancelled, TWOS may need to roll back. Rollback is only needed if work has been done by the object receiving the out-of-order message at a later virtual time, or if the cancelled message had already been processed. In either case,

TWOS must determine if a rollback is necessary, and, if so, determine the correct time to roll back to.

As a result of rolling back incorrect computation, TWOS may need to cancel some messages sent by the rolled back object. Merely delivering the negative copy of the message to the same place the positive copy went will achieve cancellation. There are some complexities involved with whether the message was to the object itself, or to another object on the same node, or to another object on a different node. Also, since the underlying message delivery subsystem does not guarantee ordered delivery of messages, TWOS must be prepared to handle a negative message that beats its positive copy to the destination.

While perhaps not a theoretical necessity, in any real system only a finite amount of memory is available. Even in virtual memory systems, while the system may not run out of memory for a long time, using vast amounts of memory can be costly. The Time Warp commitment mechanism permits a Time Warp system to free some of the memory it is using, throwing away messages and states that the system is through with, and will never need again. If a state or message will never be needed to support a rollback, it can be discarded.

TWOS commitment works in two parts. First, TWOS must determine which items can be safely deleted. This determination is made by calculating Global Virtual Time (GVT), defined to be the time of the earliest unprocessed event in the system. Since events cannot change other events at earlier virtual times, no event with a time earlier than GVT will ever be needed for rollback, so any information associated with such events can be discarded. Finding the states and messages that can be discarded and deleting them is the second part of commitment.

There are a surprising number of tricky issues that must be dealt with at this layer. In addition, the special services provided at higher levels tend to cause further complexities in this code. None the less, the code at this level is actually of modest size. Performing theoretically correct Time Warp synchronization does not require that much code. Performing it efficiently, with the functionality real users need, is the source of most of the system's complexity.

1.5 Special Services

While the **Basic TW** layer provides the minimal necessary services for TWOS synchronization, it has many shortcomings. For instance, it does not permit dynamic creation and destruction of objects. Many applications require the creation of objects not known about at the beginning of the simulation, and many applications are able to discard objects that they know they will not need any longer. TWOS provides these capabilities.

In a very basic TWOS system, all data local to an object is kept in a statically sized state. This method makes some common programming methods, such as the use of lists and trees, very difficult and expensive, as the user always requires as much space for each copy of a state as the largest size his data structures may ever achieve. TWOS relieves this problem by permitting the dynamic allocation and deallocation of memory segments.

Real applications require input and output from the system. TWOS supports some basic I/O operations. The I/O capabilities of the underlying system cannot be used because their I/O operations cannot be rolled back. TWOS provides a form of I/O that works correctly in the face of rollback.

Applications that are irregular in their performance behavior require dynamic load management to achieve their best performance. In addition, a good dynamic load management system makes good static load balancing unnecessary, saving the user from a great burden. TWOS contains a dynamic load management capability, as well.

The critical path of a simulation is a limitation on how fast it can be run in parallel. Users wishing to speed up their programs may want to examine the program's critical path. TWOS is able to calculate the critical path of a simulation by running it, memory permitting.

1.6 The Tester

The tester is used for debugging the TWOS system. It has hooks into all layers of the system except the Base OS. The tester is able to parse debugging commands and call the appropriate debugging routine in the system. It also provides the ability to access debugging commands on any node in the TWOS run. Approximately 100 debugging commands are available from tester, allowing users to examine objects' input and output queues, individual messages and states, and internal TWOS data structures.

The tester parser is also used to read the configuration file. All commands in the configuration file are comprehendible by the tester. The tester can create objects, send them messages, adjust system parameters, turn dynamic load management on and off, suppress output, and cut off the run at a particular point. Typical users interact with the tester only through the configuration

file, as they should have no need to debug TWOS itself. In a few cases, the tester debugging facilities can help in debugging a user-level problem.

Chapter 2: Basic TWOS Data Structures

TWOS applications are decomposed by their designer into units called *objects*. An object should be some identifiable entity in the simulated world that performs substantial amounts of work independently. TWOS is capable of making a further division of objects along the temporal axis of virtual spacetime into units called *phases*. For instance, one object could be divided into three phases, one handling all activity during the first third of the simulation, another handling the activity during the second third, and the final one handling the activity during the last third. TWOS automatically assures that consistency is maintained among the various phases of an object, despite any rollbacks that may occur in any of the phases.

All TWOS objects consist of one or more phases, each of which is an independent process. A phase can be located on any node of the system, without regard for the location of any other phases of the object. Each phase is has its own controlling data structure that is represented in one of the nodes' scheduler queues.

Every object is defined as being of some type, similar to the declaration of class in object oriented programming. For example, in a simulation of pool balls rolling and colliding on a table, some objects will be of type `pool_ball`, others will be of type `table_sector`, and others will be of type `cushion`. Associated with each type is executable code describing how objects of that type handle event messages, plus some initialization and termination code. All objects of a given type on a given node share the same code, so this code must be reentrant. Also, objects of each type have a set of local variables specific to that type. Every object gets its own copy of these local variables, called a *state*. The state correspond to instance variables in object oriented programming.

2.1 Object Control Blocks

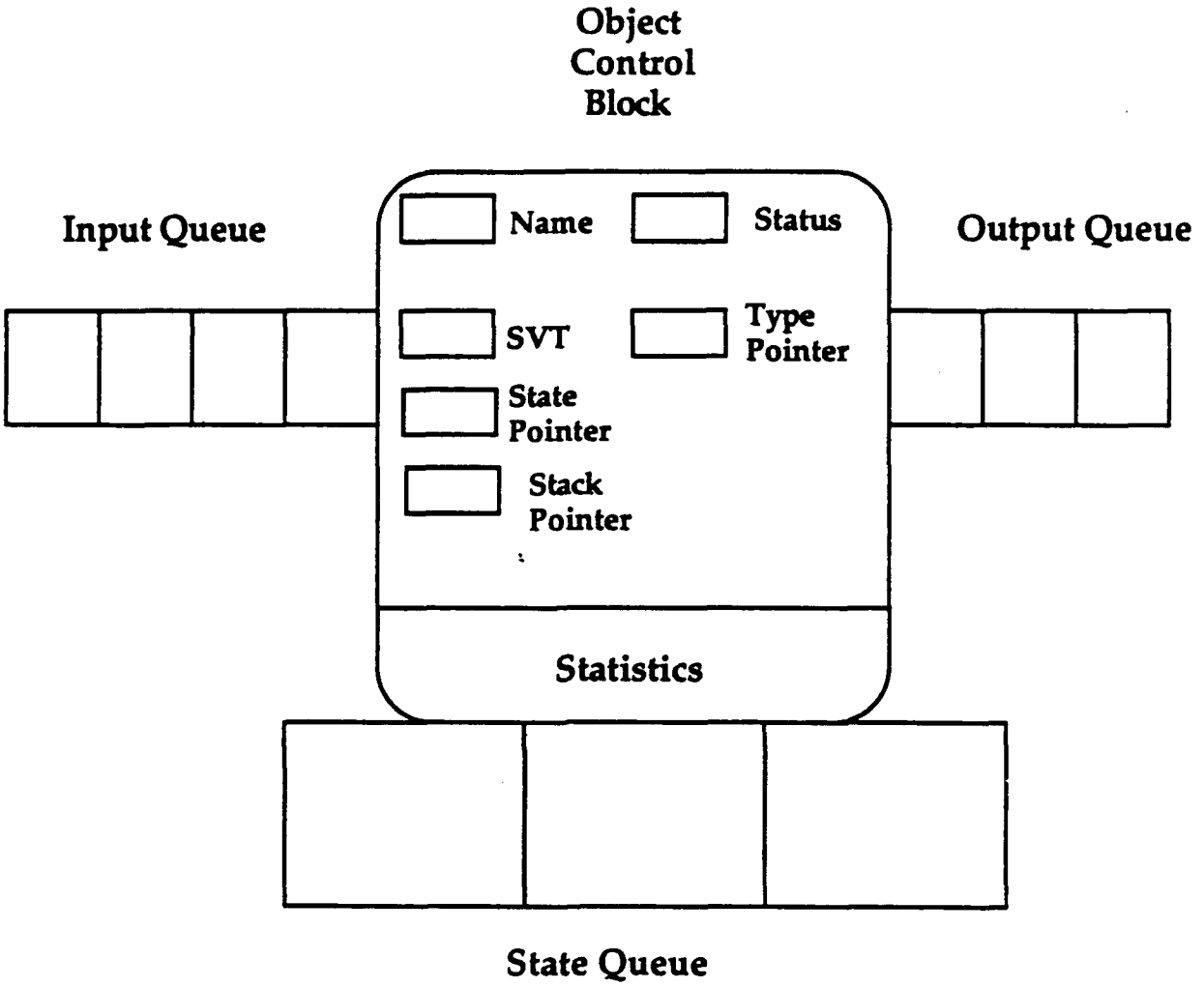


Figure 2-1: Structure of a TWOS process

Figure 2-1 shows a conceptual picture of a TWOS process (phase). The central structure represents the executable and control portion of the process, and is known as the *object control block* (OCB). (It should be called the phase control block, but the existing terminology predates the invention of phases, and has not yet been removed from the code. OCB will be used throughout, but readers should remember, when relevant, that a given object can consist of several phases, each with its own OCB.) The process' name (consisting of the object's name and the phase delimitation times) is stored in its OCB. The process' simulation virtual time (SVT), the virtual time at which the process is to execute next, is also kept here. The OCB's type pointer indicates the user-defined type of this object, and provides a means of getting at the code for the object. The state pointer points to the state being used by the process'

current event, and the stack pointer points to the stack for that event. Also, the OCB contains all of the statistics kept by TWOS for this process.

The object control block also contains pointers to three queues, shown in Figure 2-1. To the left of the object control block is the process' *input message queue*. This queue contains copies of messages sent to the process, ordered by their virtual receive times. Unlike most systems, TWOS cannot discard messages it has already handled until it can be certain that they will not be needed again, since a rollback may cause them to be re-executed. Therefore, at any moment in the TWOS run, some of a process' messages will have already been handled, while others are awaiting handling. A pointer in the object control block indicates the latest message so far handled.

To the right of the object control block is the process' *output message queue*. This queue contains copies of all messages sent by the process to itself or to any other process, ordered by their virtual send times. In case of a rollback that requires the process to cancel output messages, the copies stored in the output queue will be used for that cancellation.

Below the object control block is the process' *state queue*. This queue contains copies of the process' local state information. If the process needs to roll back to a particular simulation time, the state queue entry with the next earlier time will be restored as the process' current state. Since every event performed by the process is likely to cause a change in some of that local information, typically one state is saved for every event performed. Correct operation does not require all states to be saved, but doing so has proved to yield better speedup than saving only some of them. Since states can contain dynamic memory segments that may not change from event to event, saving a state does not always require saving all pieces of the state. The state queue is ordered by the time of the event producing the copy of the state.

Object control blocks are kept in one of two places. Normally, they are stored in a node's scheduler queue. When a process is migrating, however, its object control block on the sending node is moved into a queue of migrating processes until it has been completely received on the destination node. (See Chapter 13 for details on process migration.)

2.1.1 Objects, Phases, and Processes

Users writing simulations for TWOS define objects. These objects are the units of parallelism, from the user's point of view. A given object has a locally available state that it can easily access, but other object's cannot access if and it cannot access other object's states easily. The user never needs to concern himself with any decomposition below the level of processes.

TWOS, however, can further automatically divide objects into smaller entities called phases. Chapter 12 fully describes phase decomposition. For

the moment, it is sufficient to understand that a given object is composed of one or more phases. Each phase can be independently located on its own node, or several phases of the same object can share a single node. Phases are the unit of processor scheduling – each has its own OCB, each has its own time at which to be scheduled, and so on. From a scheduling point of view, in fact, TWOS does not recognize any relationship between phases of the same object.

To avoid confusion, this manual will speak mostly of *processes*. A process is one phase, which may or may not be an entire object. The term "object" will be used when referring to the collection of one or more phases making up the object. "Phase" will be used only when dealing with the specific characteristics of a process that relate to temporal decomposition.

2.2 States

Generally, processes use a state as an input to an event, and produce a state as a result of the event. The state used as input must be timestamped earlier than the event, while the state produced will have the same timestamp as the event.

A diagram of the state data type is shown in Figure 2-2. The state has a header storing control information, such as its timestamp, whether the event producing the state is in error or not, and a pointer to the object control block that owns the state. The state also contains pointers for files the process currently has open. In addition, there is a table of dynamic memory segments. Processes can allocate and deallocate dynamic memory segments in user code (as explained in chapter 9), and this table keeps track of them. Finally, there is a statically sized data area. The size of this data area is dependent on the type of the process.

States normally reside in a process' state queue. Each process requires only two states for correct execution – one to handle the next event, and one to handle any possible rollbacks. The state used to handle rollback is the latest state for the process that can be proven to be correct. However, if only two states are saved, any rollback must roll all the way back to the provably correct state and go forward from there, re-executing every subsequent event. If more states are saved, a rollback need only go back to the latest state at an earlier time than the rollback. In practice, TWOS tries to retain all states produced by a process until they are known to be correct. [Bellenot 92] describes the performance implications of saving fewer than every state.

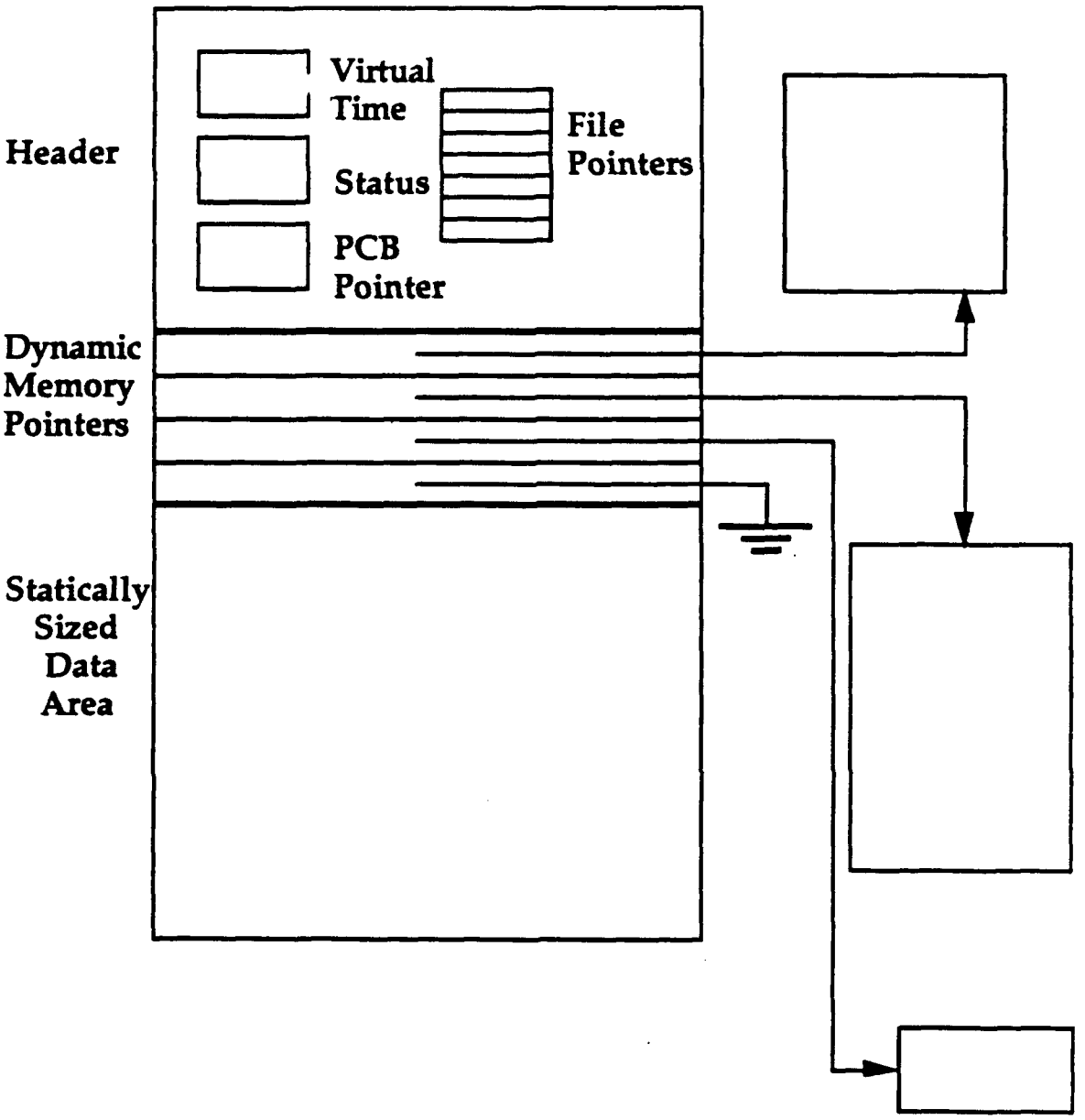


Figure 2-2: Structure of a TWOS State

When processes migrate, their states must be shipped to other nodes. States can be fairly large, complex data objects, however, because of the attached dynamic memory segments, all of which must also be shipped to the destination. Therefore, TWOS has a special queue in which to keep states that are being moved to other nodes. Only when all pieces of a state have arrived at the destination is the state removed from this queue.

2.3 Messages

TWOS messages come in two types – user messages and system messages. User messages are sent by processes and are delivered to processes at the explicit request of user code. System messages are sent from one node's copy of TWOS to another's. Users cannot send system messages and system messages are never visible to user code. The two types of messages share much of the same delivery mechanisms, but get very different types of treatment.

User messages come in several subtypes. The most common is the *event message*, which causes an event to occur at the destination process. Also, the user can send dynamic creation messages and dynamic destruction messages, which are discussed in Chapter 14.

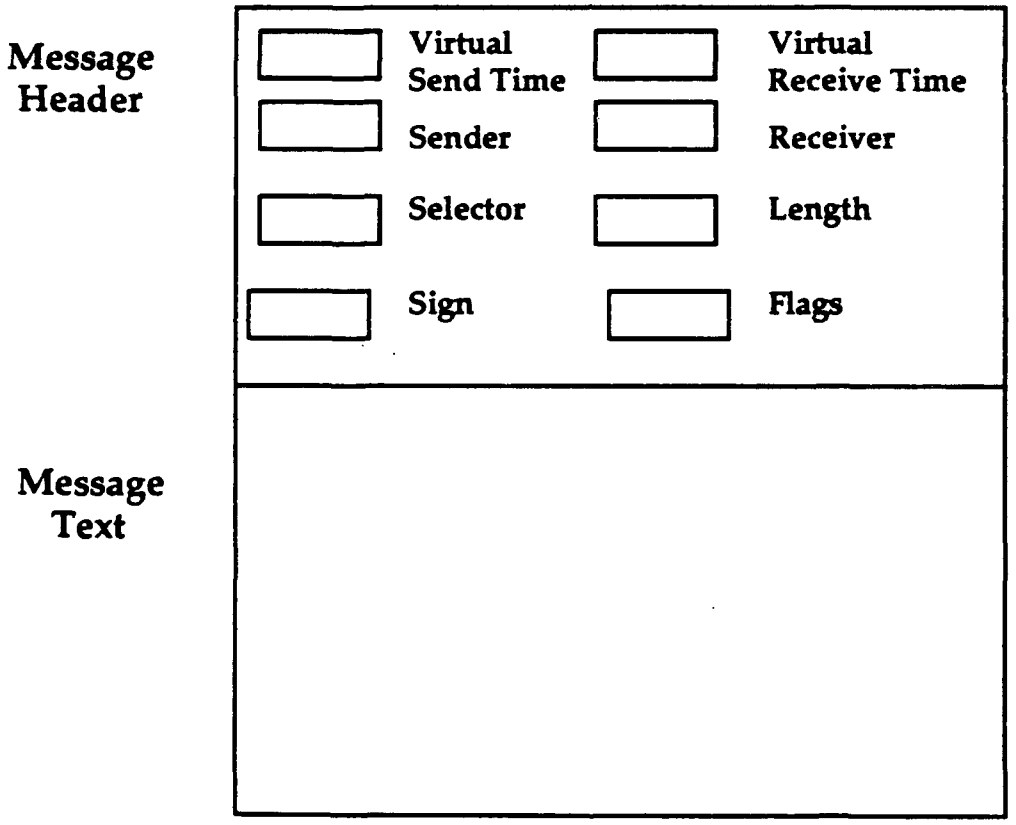


Figure 2-3: Structure of a TWOS Message

Figure 2-3 shows an event message structure. It consists of a header and text. The header contains control information, including the identity of the sending and receiving processes, the sending and receiving time (virtual times, not real time), the length of the message's text, a user-specified message selector, the sign of the message, and flags. The flags are used to indicate whether a message is travelling forward or backward, whether the message is

being migrated, whether an input message has had its event run yet, and whether an output message is a candidate for lazy cancellation. The header information generally cannot be altered or examined by an application, other than through certain special interface calls. The message text, on the other hand, is completely available to the receiving process. The user can include any type of information in the text area.

2.4 Virtual Times

TWOS supports a complex virtual time data structure. It consists of three fields. The first field is the simulation time field, the second is called sequence1, and the third is called sequence2. Users are encouraged to use the simulation time field to encode passage of simulated time, while the two sequence fields are meant to be used to encode sequentialities within a single simulated time. Virtual times are ordered by their simulation time fields first. Two virtual times with identical simulation times are further ordered by their sequence1 fields. If those fields are also identical, the virtual times are ordered by their sequence2 fields.

TWOS currently implements the simulation time field as a double length floating point number, taking 64 bits, on the BBN GP-1000. Both sequence fields are implemented as unsigned long integers. TWOS uses the particular data representations of these types of data items on the GP-1000 to implement fast comparison macros, but TWOS also contains slower, more portable, comparison routines.

Chapter 3: Sending and Receiving TWOS Messages

The easiest way to explain the Time Warp message sending and receiving code is to use an example of the handling of a single message. The message followed in this example is a message sent from process A, on node 0, to process B, on node 1, with a send time of 100 and a receive time of 200. Process A is running an event at time 100, when the process issues a `schedule()` command, as shown below:

```
schedule ( "B", 200., 0, 5, "hello")
```

This command means Time Warp is to send a message to the process named "B" to be received at time 200. (A simple floating point time is used in th example. In actual practice, an abstract data type of type `Vtime` is required f the receive time parameter to `schedule()`.) The message's selector is 0. It contains five bytes, the word "hello".

TWOS runs as a single load module, with user code and system code linked together. `schedule()` is a routine defined internal to TWOS. It makes a check to determine whether the message to be sent will be received after a user-provided cutoff time. If so, the simulation will end before the message is received, so actually sending the message would simply waste time and memory. Therefore, in this case the system merely returns, doing nothing with the message in question. Since the effect of the message is invisible until it is received, and this message would never be received, the user cannot tell that the system did not actually send this message. Messages with receive times after the user-specified end of the simulation are also not counted in statistics.

If the message is to be received before the cutoff time, the system calls `switch_back()`, an assembly language context switching routine. `switch_back()` is meant to switch control to the Time Warp Operating System, to a routine called `sv_tell()`. (The actual mechanism for getting to `sv_tell()` is a little more complicated than this, but the additional details are not relevant to this discussion. `schedule()` used to be called `tell()`, which is why `sv_tell()` has the name it does.) Time Warp maintains separate stacks for the operating system itself and for any process that is in the middle of running an event. Until `switch_back()` is called, TWOS is still running off the stack of the process requesting the `schedule()`. `switch_back()` saves the context for the process' stack, restores the context for TWOS' own stack, and arranges for control to be transferred to `sv_tell()`.

Once `sv_tell()` is called, TWOS is running again under its own stack. Error checking is the first thing done by `sv_tell()`. The routine makes sure that the length of the message to be sent is less than or equal to the length of a

packet. Then it makes sanity checks on the receive time of the message. The receive time may not be less than the current send time. If the receive time and send time are the same, the process receiving the message cannot be the process sending the message. (This restriction helps avoid some forms of infinite loops.) Finally, TWOS checks if the user is trying to issue a receive time that is greater than $+\infty$. Such receive times are illegal. The system also makes sure that the user has not given a null string for the receiver's name.

Assuming that the message passes all error checks, the message is copied into the process' argument block, a structure kept in the object control block. The process' status is set to indicate that the object is trying to send a message, and `dispatch()` is called.

`dispatch()` is the main scheduling routine in TWOS. Many things can happen once `dispatch()` is called. Other processes may have services performed for them first, before the system gets around to this `schedule()`. For instance, a process may need to determine what virtual time it will execute at next. Or some other process' message may be sent off before process A's. `dispatch()` may even decide to run another process before it gets around to actually sending the message we are following. The ordering of actions is purely by the earliest virtual time first, at this stage. The virtual time of process A's request is the send time on the message, which is also the virtual time of the event process A is executing. (Further details of the workings of `dispatch()` can be found in chapter 4.)

Assuming that process A is still the process on this node with work to do at the earliest virtual time, `dispatch()` will deal with this message immediately. It calls a routine named `sv_doit()`. `sv_doit()` has the responsibility for actually sending a message.

TWOS must have two copies of each message. First, it must have a copy to send to the receiving process' input queue. Second, it must have a copy to save in the sending process' output queue, in case the message needs to be cancelled. From TWOS point of view, it does not have any copies of the message at all, yet, since a TWOS message is a combined structure containing both header information used by the operating system to handle the message and the data that the user wanted to transmit to the receiver. The copy of the message data provided in the `schedule()` call cannot be used as one of the copies, first, because it has no header attached, and second, because it is in the process' memory space and may be written by the process at any time. Therefore, `sv_doit()` must make two copies of the requested message. It calls a routine called `mkomsg()` twice. If either call fails due to lack of message memory, then TWOS must delay sending this message. Chapter 10, on message sendback, discusses steps that would be taken if there were not sufficient memory to make one of the copies of the messages. If those steps

failed, the event trying to send the message would be rolled back. In the case of this example, there is no memory problem.

`mkomsg()` allocates a full-sized message buffer of the maximum size of any message that can be sent, plus extra bytes for header information. The message buffer is first entirely cleared. (If the buffer is not zeroed, garbage bytes may cause misordering of the message in the receiver's input queue, leading to non-deterministic results.) Then some of the header fields are filled in, such as the identities of the sender and receiver.

Assuming that `mkomsg()` succeeds in finding sufficient memory for both copies of the message, `sv_doit()` copies in the text of the message. One copy is marked as negative and immediately stored in the sending process' output queue. `deliver()` is called to handle the positive copy that is to be sent to process B.

`deliver()` is a general purpose function that is called whenever a message has to be sent somewhere. It is used not only for normal message delivery, but for message cancellations, message sendback for memory management, and process migration. Generally, `deliver()` is called whenever a node has a message that is supposed to go into some queue at some process. At this point, the system does not know whether the destination process is local or remote, nor whether the message should go into that process's input or output queue, nor whether the message requires any kind of special handling. `deliver()` is the routine that finds the answers to those questions.

The first task of `deliver()` is to determine what sort of handling the message requires. `deliver()` looks at bits in the message header to determine whether the message is going forward for delivery, or backward for memory management purposes. Based on this information, it calls the *phase location facility* to determine where the process receiving the message resides. This chapter will not go into the details of process location; see chapter 17 for further information on process location. In the normal course of events, the phase location facility will return a location data structure immediately. This data structure contains a field identifying the node hosting the destination process. If that node is the local node, another field contains a pointer to the process.

In this case, the destination process is not on the local node. Process A, the sender, is on node 0. Process B, the receiver, is on node 1. `deliver()` is currently running on node 0. Therefore, the only information in the process location data structure is that the destination process is on node 1. `deliver()` is prepared to handle certain complicated circumstances involving migrating processes and misdelivered messages, but those issues don't arise in this case. (See chapter 13 for more information on these issues.) `deliver()` simply sees that the process is not local and calls `sndmsg()` to transport it to the node hosting the receiving process.

`sndmsg()` sets up several message header fields for the use of the underlying message passing facility. Then `enq_msg()` is called to put the message into a queue of messages to be sent off of this node. Each node maintains a single queue of messages to be sent to other nodes. System messages are kept at the head of the queue, to ensure fast delivery. After the system messages are negatively signed messages. These messages will usually cause work being done in error to roll back. Since TWOS runs faster if less work is misdone, the system tries to deliver negative messages faster than normal messages. Also, theoretical correctness requires negative messages to have delivery priority over positive messages. After system and negative messages, the queue contains positively signed user messages, in order of their send times. The message from process A is put into its proper place in this part of the queue.

Once the message is in the queue of messages to be sent, `sndmsg()` calls `send_from_q()`. The basic purpose of `send_from_q()` is to attempt to ship off the first message in the queue of messages to be sent. For a variety of reasons, the attempt might not succeed. If it does not, the system will try again later. The code used by `send_from_q()` and the functions it calls are highly machine-specific. This chapter addresses only hardware-independent issues, so the example message drops out of sight, at this point, being passed off to the underlying message delivery system. (Further details of what happens to the message on a sample architecture, the BBN GP-1000 running the Mach operating system, can be found in Appendix G.)

Eventually, that underlying system will pass the message back up to TWOS, this time on the destination node, node 1. The main loop of TWOS periodically checks for incoming messages using a function called `read_the_mail()`. The implementation of this function is machine-specific, but it typically tries to allocate a local message buffer to hold the message, then enqueues it in the node's receive queue. TWOS will eventually look at the front of this receive queue to handle incoming messages. When it does, TWOS calls `msgproc()`.

`msgproc()` is a general routine to handle any message that arrives at the local node. The message may be a system message or a user message. In this example, it is a user message. `deliver()`, the general purpose user message handler mentioned earlier, is called again, this time on node 1. Node 1 hosts process B, so the path through `deliver()` is a bit different this time.

Again, `deliver()` checks to see if the message is travelling forward or in reverse. In this case, as for all normal event messages, the message is travelling forward. Then `deliver()` calls the phase location facility. Since the receiving process is local, the phase location facility will return an answer immediately. This answer will consist of a location data structure with one field showing that the process is local, and another holding a pointer to the

process' control block. `deliver()` calls `nq_input_message()` to put the message in the receiving process' input queue.

`nq_input_message()` makes some preliminary checks to make sure that the message is being correctly delivered to the proper process. This is one example of the TWOS strategy of multiple redundant checks, a strategy that has proven very helpful in finding system bugs. `nq_input_message()` then calls `find()`, an extremely general purpose function that finds the position for an element in a sorted queue. In this case, `find()` is looking for the correct position in the receiving process' input queue. That queue is ordered first by receive time, then by message selector, then by a byte-by-byte comparison of the texts. Having a fully deterministic ordering of input queues is vitally important to TWOS. If the queues are not deterministically ordered, TWOS cannot guarantee the same results as a sequential run.

`find()` returns a pointer indicating where to add the incoming message in the linked list that forms the input queue. In the normal case, the message will simply be slipped into place in the queue. (If this were a cancellation message, the results of `find()` would indicate which message the cancellation will annihilate in the queue.) At the moment, however, the message is occupying one of a limited number of message buffers available to the system. So `nq_input_message()` will attempt to allocate space for the message, copy it from the buffer, and return the buffer to the buffer pool. If there is no memory shortage, this operation will succeed.

The arrival of the message will also cause process B to re-evaluate when it needs to run next. B may have been scheduled to run after time 200, the receive time on the message. In this case, B's control block must be changed to indicate that B needs to run at time 200, to handle this new message, which will also cause the scheduler queue to be reordered. Alternately, perhaps B was already scheduled to run at time 150. The new message at time 200 will be handled after the event at time 150, so the control block need not be changed.

A more complex possibility is that process B has already run at some time after time 200, at time 250, perhaps. In this case, TWOS needs to execute a rollback to ensure that the work done prematurely at time 250 is undone. Rollback is not covered in this chapter, so we will presume that process B has not executed any events later than time 200, and that it is currently scheduled to perform its next event at time 150.

Node 1 must check to see if the arrival of this message caused a rollback, then. A routine called `rollback()` does just that. `rollback()` checks the time of the operation that might cause a rollback (in this case, the arrival of a message at time 200) against the process' current simulation time. Process B is currently at simulation time 150, so this new message will not cause process B to run at an earlier time. Thus, no rollback is needed and `rollback()`

returns immediately. The input message has now been completely enqueued, and delivery of the message is complete.

Once node 0 has had the message put into the queue of messages to be sent off node, the remainder of the event that requested the message can be run. Process A is marked as being ready to run again, and sooner or later the scheduler will return control to it.

At the destination end, the user's process cannot tell that a message has arrived until the resulting event is scheduled. (User code is not permitted to examine the future event list.) When the event caused by the message is run, the message becomes visible to user code. At some point in the application code that handles this event, process B will almost certainly need to examine the message that caused the event. The process must be given a copy of the original message, not the message itself, because the process may modify the message's contents. If this event needs to be rolled back and re-executed, the process will need a pristine, unaltered copy of the original message. Therefore, the copy of the message stored in the input queue is never actually accessed by the process. Instead, a function called `message_vector()` makes a copy of the message for use during the event.

TWOS then sets up control structures to allow a context switch to the code that will run process B's event at time 200. It also puts the copy of the message in a place where the event code can find it. Then context is switched to the event. At some point during the event, the message causing it will probably need to be examined. The user code executes a macro called `msgText()` to get a pointer to the message. It can then do whatever it needs to with this message.

Chapter 4: Event Scheduling

Each node used in a TWOS run has its own local scheduler queue containing all processes handled by the node. Figure 4-1 shows a closeup of the scheduler queue. The scheduler queue is a linked list of phase control blocks ordered by the next virtual time at which the processes are to execute. Whenever TWOS decides to schedule a process to run, the `dispatch()` routine selects the first process in the scheduler queue, which is also the process with the earliest unrun event on the node. In Figure 4-1, the process with an event at time 100 will run next, unless some other work is scheduled for an earlier virtual time before the scheduler next selects a process.

Processes can be in a variety of conditions. A process in the `GOFWD` state has rolled back, and needs to be examined to determine the time of its next event. A process that is `BLKINF` is blocked at infinity, and currently has no more work to do. Only a process that is `READY` is eligible for immediate running. If the process at the front of the scheduler queue is `READY`, `dispatch()` calls `load_obj()` to prepare that process for running an event.

A `READY` process is either `EDGE` or `NON-EDGE`. `EDGE` status means that the process has not yet started the event it wants to run. `NON-EDGE` status means that the event has already been partially run, and will be resumed in its middle when this process is selected for execution. Events can be interrupted by the system when there is more important work to be done. In the BBN GP-1000 version of the system (and the Sun workstation version, as well, though not in all previous versions), TWOS will not automatically interrupt a running event, but when the event requests certain system services TWOS can choose to switch to a different event.

When `load_obj()` is called for a process with `EDGE` status, TWOS must completely set up all data structures the process needs to run its event. None of this work is actually done in `load_obj()`. Rather, `load_obj()` calls `objhead()` to set up the process to run. Doing so requires several memory allocations. Should any of those allocations fail to be satisfied, TWOS will invoke message sendback (as described in Chapter 10) to free the necessary memory. If message sendback cannot free the required memory, the event cannot be run at this time. TWOS frees any memory already set aside for this event and attempts to run the event again, later.

Before any data structures can be set up, TWOS must determine the *type* of the process. A process' type is defined by the user when the process is created. In a simulation of colliding pool balls, the processes might have types "ball", "sector", and "cushion". In a military simulation, the processes might have types "division", "corps", and "air mission". A process' type defines what sort of object in the simulation world it represents.

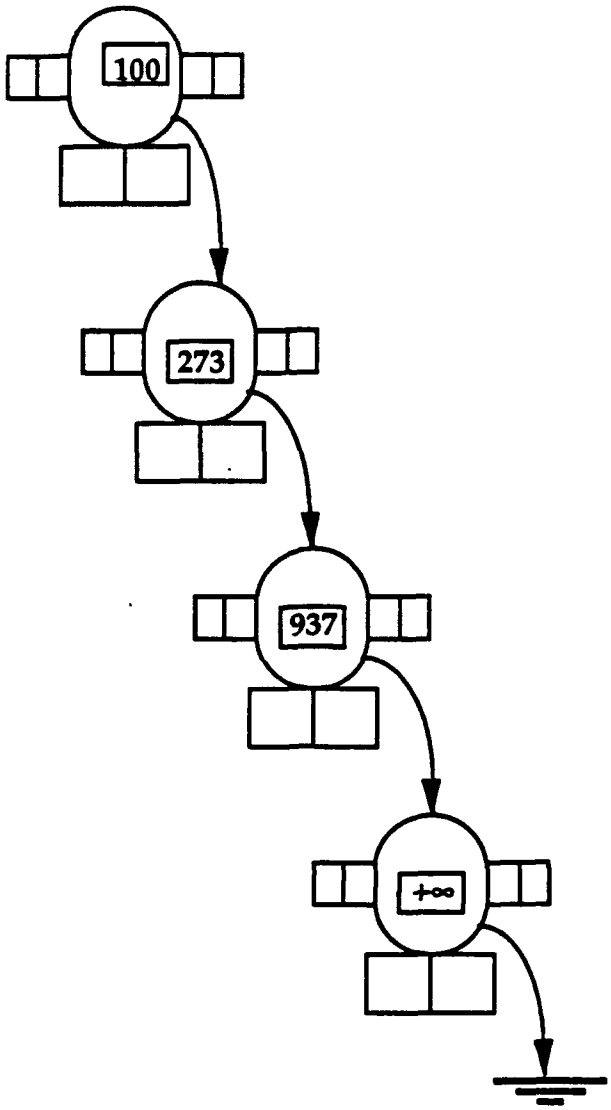


Figure 4-1: The Scheduler Queue

The scheduled process' type is indexed into a table to determine where to find the code necessary to execute an event for that type. Also, the table contains information about the size of that type's state. TWOS uses this information to attempt to allocate a new state for this process' event, calling a routine named `loadstatebuffer()`. TWOS must also allocate a stack for this event's local use, which is also done in `loadstatebuffer()`.

If the state allocation is successful, TWOS must copy the contents of the previous state into the newly allocated memory. The previous state may have had dynamic memory segments attached. If so, TWOS does not yet allocate or copy for those segments.

After `loadstatebuffer()` returns successfully, TWOS tries to get memory for copies of the messages that have caused this event. A single event can be caused by an arbitrary number of messages, all of which must have identical receive times. The process already has copies of all of the messages, but those copies cannot be given to the receiving process' event. This execution of the event may turn out to be premature, or erroneous, and if it is, the messages need to be preserved in their original form to permit correct re-execution. Also, if the event changed these copies of the messages, message cancellation would not work if one of them proved erroneous. The antimessage used to cancel it would not completely match the altered positive message, and cancellation would fail. TWOS does not have sufficient control of the physical memory on all platforms to force the copies to be read-only for the user, so the only choice is to make a new copy and permit the event to do whatever it wants with those copies. A routine called `message_vector()` handles making fresh copies of the messages for the event.

At this point, `setctx()` is used to arrange to pass control to the process' stack. When control passes back to TWOS' main loop, as it soon will, that code will notice that an event has been set up to execute. (For more details about the main loop of TWOS, see Chapter 21.) The main loop may choose to do some other system business first, but eventually the main loop will use the routine `switch_over()` to actually pass control to the user code written to handle this event. This user code will continue to execute until the event is completed, or the code requests a TWOS system service. If, for whatever reason, the event does not run to completion, TWOS puts the process' OCB into `NON-EDGE` status.

Scheduling a process with a `NON-EDGE` status is much easier. The data structures that the event requires have already been completely set up, so TWOS merely uses `setctx()` to alert the main loop that control should be switched to the process at the point at which it halted.

When an event completes, TWOS releases the memory used for the event's stack (using `l_destroy()`, a general purpose deallocation routine) and the copies of its messages (using `destroy_message_vector()`). TWOS then calls `go_forward()` to change the process' SVT to the receive time of that message and reorders the scheduler queue to reflect the process' new SVT. (This process is covered more fully in Chapter 5.) This process' next event will occur at a virtual time equal to the virtual receive time of the earliest unprocessed message in the input queue. TWOS then calls `dispatch()` to schedule the next event.

In certain cases, TWOS must back out of an event before it completes. For instance, the event may be rolled back in the middle of performing it, or it may request a memory allocation that cannot currently be satisfied. In such cases, TWOS deallocates all memory attached to the event (state, stack, and

message vector) and changes the status from NON-EDGE to EDGE. TWOS will reschedule this event at some point in the future.

Chapter 5: Rollback

TWOS can roll back for a number of reasons, but the most basic, and probably most common, reason is that a message arrives at a process with a virtual time lower than that process' *SVT* (simulation virtual time). Such an arrival means that the process has processed one or more events at virtual times later than that of the newly arriving message. Any such prematurely processed events must be rolled back. (Keep in mind while reading this chapter that rollback can occur for other reasons, since some code discussed is intended to handle such cases.)

The explanation of a rollback will be easier if a particular case with an actual process and sample virtual times is used as an example. Figure 5-1, below, shows the case. Process Y is at SVT 677, but has not yet started executing the event for time 677 when it receives a new message for time 520.

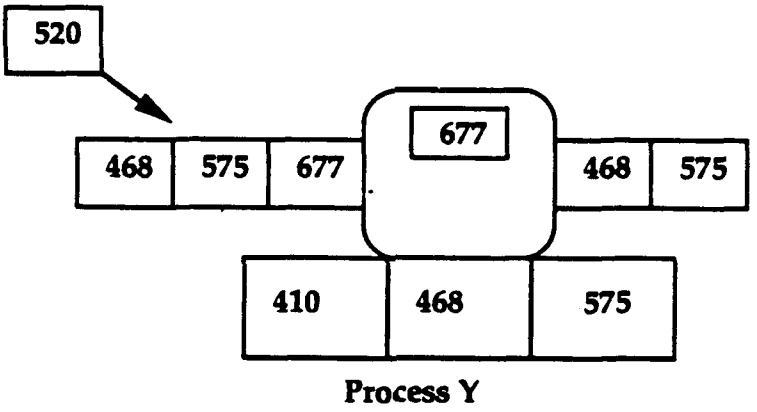


Figure 5-1: Process Y Is About To Roll Back

TWOS, at this instant, is in a routine called `nq_input_message()`. This routine is used to put incoming messages into processes' input queues. In this case, the positive enqueueing message for time 520 is to be placed in process Y's input queue. In addition to enqueueing the message, this routine calls `rollback()`. Note that `rollback()` is called before determining whether the message's receive time is later than process Y's SVT.

`rollback()` accepts two parameters, an OCB pointer and a virtual time. In this case, the pointer points to Y's OCB and the virtual time is 520. The first action performed by `rollback()` is to check if Y is a standard out object. Standard out objects are used solely for output purposes, and never roll back, since they only perform output for a virtual time once GVT has passed that virtual time, so if Y is a standard out object (STDOUT, in Unix terms), `rollback()` returns immediately. (For more details on standard out objects, see Chapter 20.) Otherwise, `rollback()` goes ahead.

After making a sanity check to ensure that Y is not going to be rolled back to a time before GVT, Y is put into run status GOFWD. This run status indicates to the scheduler that Y does not, at the moment, know what time it should run at next. The scheduler, in dispatch(), must examine Y to set its new SVT when it is in status GOFWD, using a routine called go_forward(). In this example, Y will next execute at time 520, assuming no further messages come in. However, the rollback may occur because of an annihilation, or for some other reason. In such cases, there may be no actual work remaining to be done at the rollback time, after rollback() exits. Handling such cases is the reason for using the GOFWD run status.

After setting Y's run status to GOFWD, rollback() will reorder the scheduler queue, if necessary, since Y's SVT may have changed. In our example, it changed to 520 due to the arrival of a new message. Y's SVT is set to 520 and the scheduler queue is reordered.

Process Y might have been in the middle of an event at the time rollback() was called. In this case, Y has a state that was being used to run that event, and that state should be discarded. Since the process was in the middle of an event, this state is not yet in the state queue, but a pointer to it would be stored in the OCB's state buffer field. Therefore, after setting Y's SVT, rollback() next checks the state buffer field. If it is non-null, rollback() destroys the attached state, along with the stack being used for the interrupted event and the copy of the input messages provided for that event. Also, the effective work of the process is adjusted to reflect that any work done by the interrupted event must be considered work rolled back. (Effective work is used by the dynamic load management system, as described in Chapter 11.) In our example, the process was not in the middle of an event, so these actions are not taken.

Next, rollback() calls cancel_states() to get rid of any states with times later than the new SVT. TWOS no longer uses the jump forward optimization, so states later than the rollback time must be discarded. After cancel_states() returns, rollback() will call either cancel_all_output() or unmark_all_output(), depending on whether aggressive or lazy cancellation is in use. Then rollback() returns.

cancel_states() takes an OCB pointer and a virtual time as parameters. These have the same values as rollback()'s parameters. cancel_states() starts at the end of the state queue, with the state having the latest virtual time, and works backwards until it finds a state earlier than Y's new SVT. Each state with a later time is removed from the state queue. If Y's current state pointer points to the removed state, that pointer is nulled. Y's effective work is adjusted to reflect the rolled back event, then the state may be destroyed. (In some cases, it cannot be destroyed, due to optimizations

involving temporal decomposition and pre-interval states. For each process, there is at most one such state at any given moment.)

One of the states just deleted by `cancel_states()` might have been associated with a dynamic creation. In this case, the process should resume the type it had before the creation message was processed. This information is in the last state that is not discarded, so the process' type pointer will be set to the type pointer in that state.

In our example, `cancel_states()` would discard only the state at time 575. It would set Y's type to the type in the state for time 468.

When `cancel_states()` returns to `rollback()`, the next step is to arrange for the cancellation of any messages that may have been sent by events that were rolled back. Depending on whether lazy or aggressive cancellation is in effect, `rollback()` calls a routine to handle all necessary cancellation business. The process of message cancellation is fully described in Chapter 6, using the same example used here.

At this point, `rollback()` returns. However, the conceptual process of rollback is not complete. `cancel_states()` and `rollback()` undid any work that needed to be rolled back, but `rollback()` did not necessarily set the process' SVT to the proper time. In the example, `rollback()` set it correctly. Y's SVT is set to 520, and that is when it should next execute. But if the rollback were caused by the annihilation of a message in the input queue, the rolled back process would have its SVT set to the time of the annihilated message. If the annihilated message were the only message for that virtual time at that process, there would be no event to be run, and SVT would be improperly set. Other circumstances can cause `rollback()` to set SVT to an improper value.

The first action taken in `rollback()` was to set the process' run status to `GOFWD`. This status signals the scheduler that the process in question needs to be examined to determine the virtual time it will run at next. Eventually, this process will get to the head of the scheduler queue and `dispatch()` will call `go_forward()` for it. `go_forward()` takes advantage of the fact that all states at times greater than or equal to the rollback time have already been discarded. Whenever the process plans to execute next, it will start with the last state in its state queue.

Next, `go_forward()` must find an input bundle to work with. An input bundle is one or more messages with the same receive time in the process' input queue. A routine called `earliest_later_inputbundle()` searches the input queue for this bundle. If one is found, SVT is set to the receive time of that bundle. The next event this process runs (in the absence of more rollbacks) will be associated with this bundle. `go_forward()` sets the run

status to READY and the control to EDGE, indicating that an event for the process is just about to start.

One possible rollback pattern is if an input message that already ran an event is cancelled, then some other event at a later time is run next. If lazy cancellation is being used, any output messages associated with the cancelled event are still in the process' output queue, unmarked, waiting to be either cancelled or confirmed. `cancel_omsgs()` is used to handle this situation, and is covered in Chapter 6. One result of `cancel_omsgs()` is that messages may be removed from the output queue.

Something rather odd can happen at this point. `go_forward()` has a pointer to a state and an input bundle, and one might think that the event can go ahead, no matter what. However, the cancellation that was just done by `cancel_omgs()` may cause a problem, since all messages in that event's input bundle might have been cancelled. The event TWOS is considering running may have been scheduled by an earlier event at the same process, an event that was cancelled. The symptom of this case is that the run status of the process is no longer READY. If the cancelled messages annihilated the input bundle selected earlier, then `rollback()` would have been called and that routine would have changed the run status of the process, probably to GOFWD. In this case, we need simply exit from `go_forward()`. Sooner or later, in its virtual time order, the process will re-enter `go_forward()` and try to find another bundle to execute.

There is another path through `go_forward()`. In some cases, there are no more events to be performed by a process after a rollback. In such cases, when `go_forward()` searches for an unprocessed input bundle, it does not find one. In this path, again, `cancel_omsgs()` must be called to get rid of any lazily cancelled output messages for cancelled events. Also, the run status of the process should be set to BLKINF, indicating that it has no work to do, currently. Finally, if this is a phase, rather than a full object, and it is not the last phase of the object, being put into the BLKINF state indicates that it has completed all work for this phase, in which case it should ship its final state to the next phase. `send_state_copy()` is called for this purpose. (Chapter 12 discusses the purpose of `send_state_copy()` and its effects.)

The exit from `go_forward()` completes the process of rollback. The process in question has undone the effects of the premature execution and has prepared for running at its new virtual time. From this point, the process is simply scheduled to run its event just as if the message causing it had arrived in the proper order. Figure 5-2 shows process Y after the rollback has completed. The new SVT is time 520, the input message for time 520 has been inserted into the input queue and the state for time 575 has been discarded. The output message for time 575 remains in the queue, assuming

we are using lazy cancellation. Chapter 6 discusses what happens with this message.

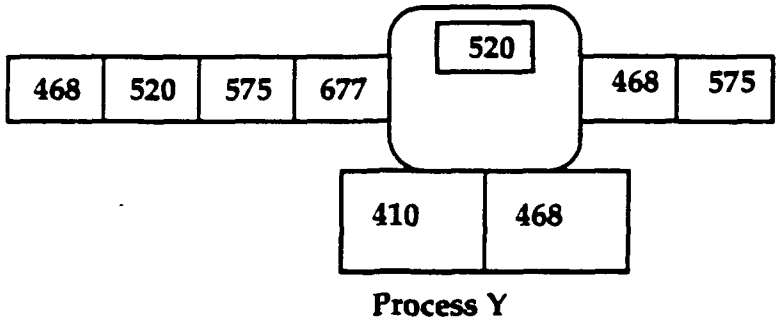


Figure 5-2: Process Y After Rollback

Chapter 6: Message Cancellation

TWOS' optimistic execution sometimes causes messages to be sent in error. When TWOS detects this problem, it must be prepared to cancel such messages. This chapter discusses how TWOS goes about cancelling messages.

Messages are cancelled as a result of a rollback. The example used in Chapter 5 illustrates the situation. Process Y has executed an event at time 575, resulting in sending a message (to process Z). Now process Y rolls back to time 520, as a result of the arrival of a new message. How will the message sent at time 575 be cancelled? Figure 6-1 illustrates the situation.

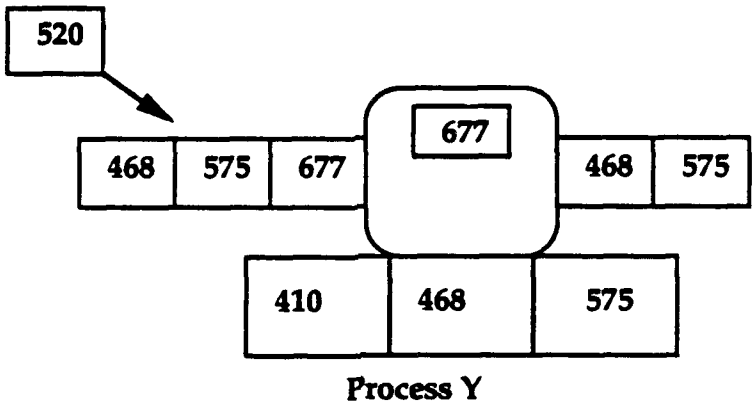


Figure 6-1: Process Y Needs To Cancel a Message

TWOS can run with either lazy or aggressive cancellation. The default is lazy cancellation. Aggressive cancellation can be used, instead, via a switch set in the application's configuration file. All processes in the simulation must use the same style of cancellation. The style of cancellation can be reset by the user through a tester command in the middle of the run (see Chapter 18 on the use of the tester), but no guarantees can be made about correct execution if this is done, so users should avoid doing so.

As mentioned in Chapter 5, cancellation during rollback is handled in the routine `rollback()`. `rollback()` has set the process' run status to `GOFWD`, and has cancelled any states that should be discarded. `rollback()` now calls a routine to deal with cancellation. Different routines are called depending on whether lazy or aggressive cancellation is in force.

In the case of aggressive cancellation, `cancel_all_output()` is called. Any antimessages in Y's output queue will be sent off right away. That is about all `cancel_all_output()` does. It runs through the output queue of the process, looking for any messages with a send time not less than the rollback time. All such messages are dequeued and submitted to `deliver()` for

routing to the destination process, where they will cancel with their matching positive copies.

In the case of lazy cancellation, actual cancellation must wait to determine if the messages will or will not be sent by a subsequent running of their rolled back event. Normally, when messages are sent their negative copies are stored directly in the output queue at the instant of sending, not at the end of the event. Thus, if an event does re-execute in a different way, it will start putting messages in the output queue in the middle of the event's execution. However, the messages from the previous, rolled back invocation cannot be certainly cancelled until the event completes. When it does, the process' output queue may contain a mixture of messages sent from the rolled back event that should be cancelled, messages sent only by the correct execution of the event, and messages sent by both versions of the event. At the end of the event, TWOS must examine these messages to see which should be cancelled.

`unmark_all_output()`, which is called by `rollback()`, makes sure that these types of messages can be distinguished. Each message can be either marked or unmarked. Messages put into the output queue during a normal send are marked. `unmark_all_output()` unmarks any output queue messages with a time greater than or equal to that of the rollback. When an event re-executes, it may send messages again. If a message sent during this second execution of the event matches a message sent during the previous execution of the event, TWOS will find the message copy in the output queue, and will realize that the message has already been sent. Lazy cancellation permits TWOS to avoid resending that message. Instead, the process simply marks the unmarked message. If the second execution of the event sends some messages not sent during the previous execution of the event, matching copies of these messages will not be found in the output queue, and TWOS will treat these sends normally, sending off the positive copies of the messages and enqueueing the negative copies in the sender's output queue. The negative copies so enqueued will be marked, just as if the event had never executed before.

At the end of the correct execution of the event, the output queue is examined for messages with the send time of the event. Any such messages that are marked are left alone. Such messages were sent by the correct execution, and may or may not have been sent by the earlier execution. Any unmarked messages in the process' output queue for that send time were sent by a rolled back version of the event, and not by the correct version. Such messages must be cancelled, by sending out the negative copy in the output queue to catch up with the positive copy in the receiving process' input queue, where the two copies will annihilate.

In the example, if aggressive cancellation is used, the output message for time 575 would immediately be shipped to its destination for cancellation

purposes. If lazy cancellation is used, that message would be unmarked and saved until the event at time 575 was re-executed. If the message is sent again during the execution, the copy in the output queue will be saved. If the message is not sent again, the copy in the output queue will be delivered for cancellation purposes. Figure 6-2 shows the situation after the rollback, if lazy cancellation is in force. (The shading of the output queue copy of the message sent at time 575 indicates that it has been unmarked.)

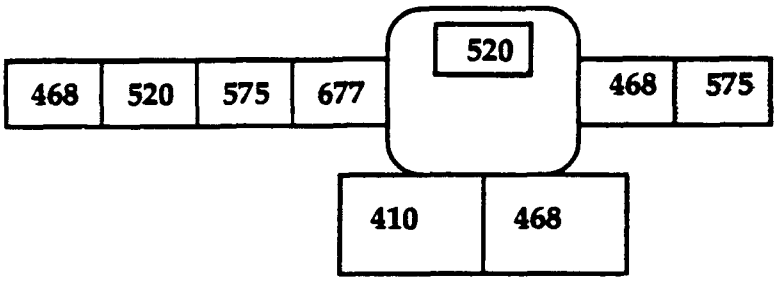


Figure 6-2: Process Y Has Chosen the Message To Cancel

But what would happen if the event for process Y at time 575 were itself cancelled? Figure 6-3 shows this case, assuming that first the event for time 575 was rolled back, then Y's input message for that time was cancelled. In this case, since no event for time 575 will be performed by Y, the mechanism described above will not cause the incorrect message earlier sent at time 575 to be cancelled.

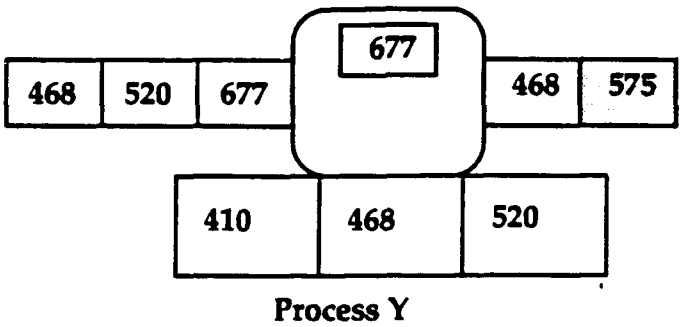


Figure 6-3: The Message At 575 Has No Event

This case is handled in `go_forward()`, a routine discussed in detail in Chapter 5. `go_forward()` is called by `dispatch()` when a process has been put into the `GOFWD` run status, indicating that the system is not yet sure of the virtual time of the next event the process needs to perform. After determining the time of that event, `go_forward()` calls `cancel_omsgs()` to get rid of any lazily cancelled messages for which events no longer exist.

`cancel_omsgs()` takes three parameters – the OCB pointer, the time at which to begin cancelling (the time of the last non-rolled-back event, which is the same as the time of the last state), and the time at which to stop (the time of the event about to be scheduled). Starting at the beginning of the output queue, it looks for any output messages in the range of times, non-inclusive. Any unmarked message it finds in the range of times is the output of an event that has been cancelled and will not be rerun; all such messages must themselves be cancelled. `cancel_omsgs()` calls `dqomsg()` and `deliver()` to send them to their destinations for annihilation, much as `cancel_all_output()` did.

In the example of Figure 6-3, the start time provided to `cancel_omsgs()` is 520, and the end time is 677. Running through Y's output queue, the only unmarked message for those times that `cancel_omsgs()` will find is the one for time 575, so that message will be dequeued and delivered.

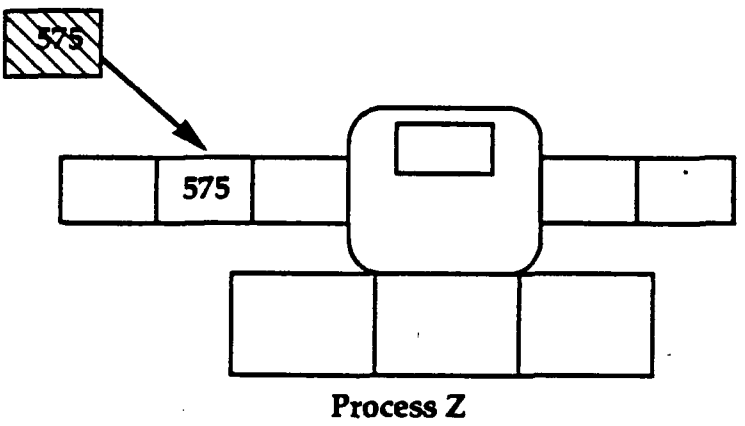


Figure 6-4: The Negative Message Arrives At Process Z

Delivery of a negative message as a result of cancellation is handled in much the same way as delivery of a normal positive message, as outlined in Chapter 3. The only important difference is that, at the destination end, instead of being enqueued, the negative message will usually be annihilated. When `nq_input_message()` calls `find()` to find the proper place in receiving process Z's input queue, a perfect match but for the sign, will be found. Figure 6-4 illustrates the situation. (Note that this figure has no timestamps for most items, in the interests of simplifying the example and avoiding confusion between send and receive times. In this figure, the crosshatching indicates a negative message.) Process Z has received the message sent at time 575 and has enqueued it. Process Z may even have performed the associated event.

At this point, `nq_input_message()` calls `annihilate()`, with both the positive and negative copies of the message as parameters. A routine called

`accept_or_destroy()` makes the memory used by the incoming copy available to the system again, and `l_destroy()` is used to dequeue the copy in the input queue and release its memory to the system. `rollback()` is then called to deal with any necessary rollback associated with this annihilation.

What happens if the positive copy of the message sent at time 575 is not in process Z's input queue? This can happen for two reasons. First, the positive copy of the message might still be in transit. Second, message sendback might have returned it to process Y. (Message sendback is discussed in Chapter 10.) In either case, the proper thing to do with the negative copy of the message is to enqueue it in Z's input queue in the normal place, ordered by receive time. Figure 6-5 shows this situation, with the positive copy of the message eventually arriving. When the positive copy does arrive, `nq_input_message()` will call `annihilate()` just as in the situation shown in Figure 6-4, with the same results.

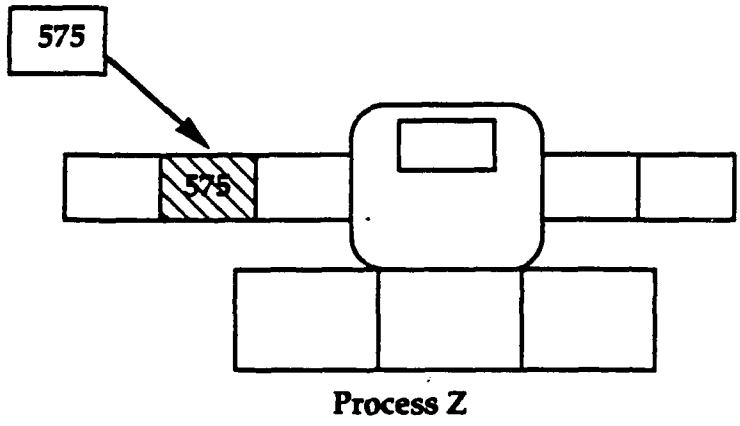


Figure 6-5: The Positive Message Catches Up To the Negative Message

Chapter 7: Global Virtual Time Computation

Computation of global virtual time (GVT) is necessary for any optimistic system in order to determine when work can be committed. Any uncommitted work may still be rolled back, so output associated with it cannot be released to the I/O device and memory used to support it cannot be freed. Also, Time Warp based systems typically use the progress of GVT to determine when the computation is completed. When GVT progresses to positive infinity, the computation is done.

In theory, the correct value of global virtual time is simple to compute. All nodes stop processing events and each node then finds the virtual time of the earliest unprocessed local event. That time is GVT. In practice, such a GVT computation method is unsatisfactorily inefficient, as all nodes must halt processing to calculate GVT. If the computation is not halted, however, generally the true value of GVT cannot be computed. Instead, a conservative estimate is computed, an estimate guaranteed not to be any larger than the true GVT. While such an estimate may not permit commitment of all possible work, if it is close enough it is also good enough. The estimation procedure must also guarantee that eventually it will correctly detect the end of the computation, of course.

The challenge, then, is to produce a GVT algorithm that is correct (never permits commitment of an event that might be rolled back), efficient (does not slow down the overall progress of the simulation more than necessary), and accurate (gives a value very close to the true GVT). TWOS' GVT algorithm meets all of these criteria.

This chapter first presents the rough outlines of the TWOS GVT algorithm, then discusses its implementation in detail. The design of this algorithm and its performance are covered in detail in [Bellenot 90].

TWOS uses an algorithm that embeds a graph structure on the nodes of the system. The actual structure of the graph depends on the number of nodes being used, but Figure 7-1 shows a typical sample for six nodes. A GVT master, the leftmost node, periodically starts the computation. It informs one or two successor nodes, each of which in turn informs one or two other nodes. This spreading message tells the nodes to start calculation of GVT, meaning that they must consider any messages they send to other nodes for GVT purposes. Note that the connections shown in Figure 7-1 are purely virtual – they have no relationship to any physical communications link in the machine actually running TWOS. Note also that the general structure of one node connecting to exactly one other node, except on the ends of the graph, does not hold for all numbers of nodes.

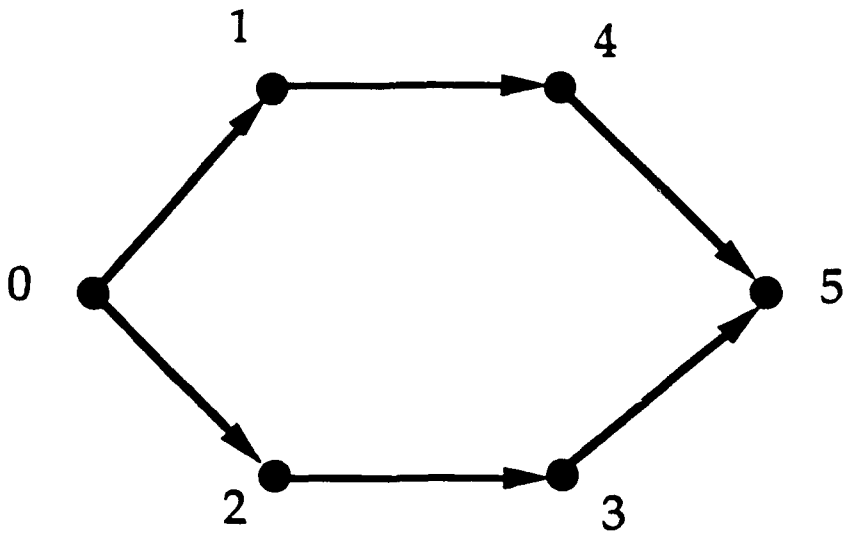


Figure 7-1: A Six Node GVT Communications Graph

Once the GVT start message has reached the node on the right side of the graph, that node knows that all nodes have started calculation of GVT. It sends a message in the opposite direction containing its own local estimate of GVT, requesting that its neighbors combine that estimate with their own estimates. The graph is traversed in the reverse direction, gathering all local GVT estimates into a single minimum. By the time the GVT master node is reached, all estimates have been considered and the master has a correctly conservative estimate of GVT. The master can start committing local information on that basis. The master informs the other nodes of the new GVT estimate, using a binary tree structure to disseminate the information. Each node can start its own commitment procedures when it receives the estimate. This algorithm uses fewer messages than many other GVT algorithms, and does not burden any single node with very many messages.

GVT calculation must consider all possible events not yet processed, including any that exist only in the form of messages in transit. Therefore, TWOS must always be able to access the virtual time of any message in transit in order to perform correct GVT estimations. In the case of distributed memory machines where messages may be travelling through a communications medium, a message may be neither at the sending node nor the receiving node when those nodes make their local GVT estimations, but the message in transit must be included in the GVT calculations. In such systems, TWOS must send acknowledgements of messages for GVT purposes. The sending node must keep track of any unacknowledged message's GVT contribution, thus ensuring that it is included for GVT purposes. At worst, a message will be considered for GVT purposes both at the sending and receiving end, in which case the GVT estimate will still be correctly conservative.

In a shared memory machine, or a pseudo-shared memory machine like the BBN Butterfly, TWOS need not send acknowledgements, as the message is never unavailable for GVT purposes. But TWOS must still be very careful to consider all possible messages. Each node may perform its part of the GVT calculation at a different time, so a message's GVT contribution could be missed if its sending node computes a GVT estimate before sending the message and the receiving node calculates its estimate before receiving the message. Further care is necessary because TWOS messages can be kept in several different places, due to process migration, process location, and low-level buffering. TWOS must be sure to examine all places a message might be stored, or the message to schedule the earliest system event might be overlooked and TWOS might improperly commit information needed to run that event.

This rough outline is sufficient for most purposes, but a more detailed knowledge of GVT computation might be necessary for debugging TWOS or for altering the algorithm. Full details are supplied below.

TWOS' GVT algorithm runs at varying intervals, but is controlled only by the passage of real time. It is not invoked, for instance, when some node runs low on memory, or needs to perform commitment. No matter what the state of the system or its nodes, the GVT algorithm is started only by the expiration of a timer. The time between timer ticks can be set by a configuration file command (see the Time Warp User's Manual, Section 2.2.3) to any positive integral number of seconds.

Only the node numbered 0 keeps this timer. This node is called the GVT master. When the timer expires, the GVT master will start up a GVT computation. In actuality, this timer does not interrupt any action being taken at the moment it expires, so the system might continue to deliver a message, run an event, or do any of a number of other things after the timer has expired. The timer's expiration causes a flag to be set, and the main loop of TWOS checks that flag's value periodically (see Chapter 21). When the flag is set, the main loop calls `gvtinterrupt()`, which calls `gvtstart()`.

`gvtstart()` can be called either by `gvtinterrupt()` (on the GVT master node), or because of the arrival of a `GVTSTART` message (on any other node), so it is somewhat general purpose. As Figure 7-1 shows, some nodes (such as node 5, in the figure) will be informed of the start of the GVT algorithm by two other nodes. Such nodes wait for all `GVTSTART` messages to arrive before proceeding any further. Once a node has received all its `GVTSTART` message (or, in the case of the GVT master, immediately), a routine called `logmsg()` is called. `logmsg()` takes the minimum virtual time of any unacknowledged message. This minimum is taken immediately, rather than waiting for the next phase of GVT computation, because some of the unacknowledged messages may be acknowledged by the time the return wave of GVT

computation reaches this node, but may not have been acknowledged by the time the receiving node performed its part of the wave. (On the BBN Butterfly, where no message acknowledgements are required, logmsg() calls butterfly_min(), which examines messages in certain low level buffers. This action is analogous to examining unacknowledged messages.)

In Figure 7-1, for example, a message may have been sent from node 1 to node 5. Node 1 lists it as unacknowledged. When node 5 starts the next wave of GVT calculation, the message still might not have arrived, so node 5 does not consider its virtual time. By the time this wave of GVT calculation reaches node 1, perhaps the message has been acknowledged, so there would be no entry for it in the list of unacknowledged messages at node 1. This message would make no GVT contribution, causing potential errors in GVT computation. By considering the unacknowledged message's time at once, before node 5 can possibly have started his GVT computation, node 1 ensures that the message will be included in GVT by at least one node.

After calling `logmsg()`, `gvtstart()` calls `gvtmessage()` to send `GVTSTART` messages to any nodes it must inform. In Figure 7-1, for instance, node 0 sends `GVTSTART` messages to nodes 1 and 2, while node 3 sends a `GVTSTART` message only to node 5. `gvtmessage()` is a general purpose routine for sending all kinds of GVT related system messages.

If the node running `gvtstart()` is the last node in the graph, with no responsibilities for sending `GVTSTART` messages to any other node, then the first wave of GVT calculation is complete. (In Figure 7-1, node 5 is the last node.) In this case, `gvtstart()` calls `gvtlvt()` to initiate the next wave of GVT calculation.

`gvtlvt()`, like `gvtstart()`, is a general purpose function. Not only is it called by the last node in the GVT chart when it has received all of its `GVTSTART` messages, but it is called by all the other nodes when they receive a `GVTLVT` message. Some nodes may receive more than one `GVTLVT` message (such as node 0, in Figure 7-1). Unlike `gvtstart()`, such nodes must perform actions each time they get a `GVTLVT` message, because these messages carry GVT estimates from upstream nodes. For instance, node 0 in Figure 7-1 receives `GVTLVT` messages from both nodes 1 and 2. The message from node 1 contains a combined GVT estimate for nodes 1, 4 and 5, while the message from node 2 contains a combined estimate from nodes 2, 3, and 5. Both of these estimates must be taken into account when node 0 calculates its GVT estimate.

`gvtlvt()` takes a single parameter, a virtual time containing a GVT estimate. In the case of the last node in the graph, this estimate is simply the minimum virtual time computed by the recent invocation of `logmsg()`. For any other case, it is an upstream GVT estimate passed in a `GVTLVT` message. In either case, the first thing done by `gvtlvt()` is to check to see if `gvtlvt()` has

already been called by this node for this GVT cycle. If it has not, `lvtstop()` is called to combine the incoming GVT estimate with all local information.

`lvtstop()` runs through the scheduler queue, looking for the minimum SVT of any local process. (The `STDOUT` object is not examined; see Chapter 20 for an explanation.) Then `minmsg()` is called. `minmsg()` checks the migration queues for their GVT contributions (see Chapter 12) using `migr_min()`, then checks some low level message queues. `lvtstop()` then calls `MinPendingList()` to look for messages awaiting phase location information (see Chapter 17). Once that information has been minned in, the local estimate of GVT is available. It is saved, for debugging purposes, and returned to `gvtlvt()`. `lvtstop()` must not be called more than once, as some of its actions are not valid if performed more than once per GVT cycle.

`gvtlvt()` passes a variable address to `lvtstop()`, which stores the local minimum in the referenced variable. Whether or not `lvtstop()` was called on this invocation of `gvtlvt()`, the variable's contents is compared to the value passed in to `gvtlvt()`, and the minimum taken. If all incoming `GVTLVT` messages have been received, `gvtlvt()` then checks its node's position in the GVT graph, sending out the appropriate `GVTLVT` messages using `gvtmessage()`. For example, in Figure 7-1, node 4 sends a `GVTLVT` message only to node 1, while node 5 sends GVTLVT messages to both node 4 and node 3. The `GVTLVT` message contains the local GVT estimate combined with all upstream GVT estimates.

When the GVT master node (node 0, in Figure 7-1) receives all of its `GVTLVT` messages, it calls `gvtupdate()` with the final estimate of the minimum unprocessed virtual time in the system. This estimate is the new estimate of overall GVT. `gvtupdate()` does not use the graph structure shown in Figure 7-1 to distribute the new GVT, but uses a binary tree to get the value to all nodes as quickly as possible. (The binary tree is not used for previous GVT cycles, as it does not provide an easy method for determining if all nodes have been contacted, unlike the graph structure shown in Figure 7-1.) `GVTUPDATE` messages are sent to any nodes that are below this node in the tree, again using `gvtmessage()`. `gvtupdate()` also contains substantial code to perform all sorts of secondary functions, such as timing and converting the new GVT estimate from a complex data structure to a set of strings. It also takes some data for use by the experimental throttling code (see Chapter 15), determines if the simulation has ended (possibly starting critical path computation if it has, see Chapter 16), schedules the next GVT cycle (if this node is the GVT master), and calls `gc_past()` to perform commitment (fully covered in Chapter 8). When `gvtupdate()` completes, the local node's participation in this cycle of the GVT algorithm is done, and it will do no further GVT work until the next cycle starts.

There are some other GVT issues. The GVT distribution graph, an example of which is shown in Figure 7-1, is used for two of the three cycles of GVT calculation. This graph is not trivial to generate, in all cases, so TWOS figures it out once, at the beginning of the run, and saves the result for use by all future GVT cycles. This graph is calculated by a routine called `gvtcfg()`, a routine that also pre-calculates the binary tree used to distribute the new GVT estimate. The graph is stored in data structures called `to[]` and `from[]`. The binary tree information is kept, in distributed fashion, in local variables called `Out0` and `Out1`.

A relatively low level routine called `msgproc()` is used to determine which functions to call to handle various incoming system messages. For all incoming GVT messages, `msgproc()` calls `gvtproc()`, which sorts them out and calls the appropriate function.

One complexity for any GVT protocol that doesn't stop all nodes to compute its estimate is handling messages sent by nodes that haven't made their contribution to nodes that have. Referring back to Figure 7-1, consider this example. Node 1 is sending a message with virtual time 100 to node 5. Before the message is sent, node 5 calculates its GVT contribution, say, 200. After the message has been sent and received, node 1 calculates its GVT contribution, say, 120. Since the delivery of the message for time 100 has been completed, node 1 has no record of it. No other node calculates a smaller contribution, so GVT is set to 120. Now node 5 has an event to run at time 100, which is pre-GVT, so the system has made an error.

The TWOS protocol handles this problem by keeping a variable called `min_msg_time`. When `logmsg()` is called from `gvtstart()`, `min_msg_time` is reset. (What it is reset to depends on the hardware in use, but is never too optimistic.) Whenever a message is sent off node, `min_msg_time` is set to its receive time if that time is lower than the current minimum. In the example described above, `min_msg_time` would have been set to 100. `min_msg_time` is later used in `minmsg()`, called out of `gvtlvt()`, to make its contribution. In the example, node 1 would thus have a GVT contribution of 100, not 120, since `min_msg_time` would hold it back.

Many low level details of keeping data for GVT calculation purposes are machine dependent. On the BBN Butterfly, which is a pseudo-shared memory machine, a message is always either at the sending node or the receiving node, whereas on a network of Sun workstations or a Mark 3 Hypercube, the message can also be somewhere in transit, either travelling on a communications link or hidden away in a low level store-and-forward buffer. These machine-specific details are not covered here, but should be kept in mind when porting TWOS to a new machine.

Chapter 8: Commitment

Commitment in TWOS refers to the process of dealing with portions of the computation that have been determined to be correct. That determination can only be made once TWOS is certain that a piece of work will never again be altered due to rollback. Each piece of user work done in a TWOS run has some virtual time associated with it. Once GVT has exceeded that time, the work can be committed.

Commitment has two possible consequences. First, memory is saved to support rollback of events and cancellation of messages. Committed events will never be rolled back, and committed messages will never be cancelled, so this memory need no longer be saved for committed states and messages. Second, since hardware and software outside of that under control of TWOS cannot be rolled back, output to such devices cannot be done until the system is certain that it will be committed. Thus, writes to files are only performed once they have been committed.

The commitment process is closely linked to GVT computation. As discussed in Chapter 7, one of the final actions each node takes in the GVT protocol is to call the routines that commit local data. The main routine for commitment is called `gc_past()`, standing for "garbage collect past". This routine runs through every process in the scheduler queue, examining each individually. `objpast()` is called for each process. After `objpast()`, which does the bulk of the work associated with commitment, returns, `count_queues()` is called to gather statistics. If the `SHORTEN_LIST` compilation flag is used in the TWOS' makefile (it is, by default), TWOS will then consider whether to split the process in two, depending on the length of its input queue. (TWOS uses doubly linked lists ordered by receive times to store its input messages, and searching that data structure when it becomes long can lead to very high overheads. This splitting mechanism is a quick and dirty method of shortening the search time; the appropriate solution, which would be to use a better data structure, could not be added in the time available.) `gc_past()` then goes on to the next process, and, after all processes are considered, returns.

`objpast()` does the majority of the work of commitment for an individual process. If the process is of type STDOUT, messages in its input queue represent output to be written to a file. The routine `commit()` is called for such processes. (For further details on output, see Chapter 20.) Next, `objpast()` finds the last state in the process' state queue. `objpast()` then performs the first of two passes through the process' state queue, back to front. The purpose of the first pass is to determine whether this process has any committed errors. If it does, TWOS will stop before any of the states have been fossil collected, leaving more debugging information for the user. The user error message associated with the error is printed, and TWOS jumps to

tester. This pass through the state queue also finds the earliest (in virtual time) state that is to be fossil collected. The time attached to that state becomes the garbtime for this process.

It is possible, due to phase decomposition, that the current value of GVT is greater than the last virtual time this phase processes. (See Chapter 12 for details of phase decomposition.) In such cases, garbtime is set to the end of the phase.

The `objpast()` routine contains a great deal of conditionally compiled code next. This code is only compiled into the system if the associated ifdef flags are defined. Some of this code is tagged with the `PLR` flag and some with the `EVTLOG` flag. The former is for debugging, the latter for using the event log (see Chapter 19). Neither flag is defined, by default, so neither body of code will be covered, here. Also, there are frequent conditionally compiled fragments of various lengths tagged with the `RBC` flag. These relate to experimental rollback hardware, and are not normally compiled in.

`objpast()` now makes its second pass through the state queue, this time looking for states to fossil collect. Starting with the first state in the queue, it looks for a state with a time greater than or equal to the previously calculated garbtime. Once it finds such a state, it breaks out of the loop and moves on to deal with other items. If the loop isn't broken, this state is to be fossil collected. To fossil collect a state, TWOS first checks to see if it has any dynamic memory segments attached. (See Chapter 9 for details on dynamic memory.) If the state does have dynamic memory segments, all of them must be deallocated before the state itself can be freed.

Deallocating dynamic memory segments is complicated by the deferred copying of memory segments used by TWOS. TWOS only copies memory segments that have been accessed, so a segment attached to an earlier state may still be valid for a later state, in which case a single physical copy is used for both. Clearly, simply deallocating such a memory segment would be a mistake, so TWOS must check to see if the segment is needed by the next state before releasing it. If it is needed, TWOS shifts the pointer to the next state.

TWOS' critical path computation facility (see Chapter 16) often requires that states be retained for longer than would otherwise be necessary. However, this facility does not require that the entire state be retained, merely a stub. If this facility is being used, instead of deallocating a state at this point, TWOS calls `truncateState()`. (More details on truncated states are found in Chapter 16.)

Normally, however, TWOS is now ready to destroy this fossil state. It calls `l_remove()` to pull it cleanly out of the state queue, then `destroy_state()` to free its memory. `destroy_state()` is careful to deallocate any memory segments and deallocate the memory segment table, if one exists, before

deallocating the state itself. (`destroy_state()` is used in other places, which is why the earlier code checked for deferred memory segments, rather than doing it in `destroy_state()`.) TWOS then moves on to the next state in the queue.

Once all fossil states have been handled, TWOS goes on to fossil collecting other items. If the critical path computation facility is in use, there may be some truncated states that can be deallocated, so TWOS checks for those next.

Input messages are handled next. `stats_garbtime()` is called to collect statistics on each fossil input message. Then TWOS checks for illegal patterns of dynamic object creation and destruction. (See Chapter 14.) If critical path computation is being used, some fossil input messages might still be needed. For such messages, only their header is needed, not their data, so `truncateMessage()` is called to free the data area, and the messages are marked as being fossils, so they can be cleaned up later. (See Chapter 16.)

More normally, fossil messages can simply be deallocated using `delimsg()`.

Once all input message fossils are handled, `objpast()` moves on to output messages. After making some checks for error conditions, `stats_garbouttime()` is called for each fossil output message. Then `delomsg()` is called to free it. The critical path code makes no use of output messages, so it adds no extra complexity to this part of `objpast()`.

`objpast()` next contains a good deal of dead code, kept only for future reference. This code attempted to fossil collect phases and entire objects that were no longer needed. This code is in two parts, separately conditionally compiled out. The first part is truly dead. The second part is largely correct, but had been causing problems that had not been fully diagnosed by the time TW 2.7 was frozen. This later code may be reintroduced into the system in the future, so it will be covered here. It is phase-related, so see Chapter 12 for clarification of any concepts particular to phases.

This code recognizes that a phase whose end time is before GVT will never run again, and is taking up a lot of space needlessly. The only value it still has is that it stores some statistics that will be needed by the `XL_STATS` file. `objpast()` allocates a small data structure to hold anything needed locally. Then this code tries to find the location of the next phase of the object and send the statistics to that phase. If it can find it without going off-node for information (see Chapter 17 for details on why it might need to go off-node), the local node packages up the statistics into a system message and sends them to the later phase. `l_remove()` then pulls the dead phase out of the scheduler queue, `nukocb()` releases its memory, and puts the shortened dead OCB data structure in a special queue.

The last actions performed by `objpast()` relate to certain forms of throttling. (See Chapter 15 for details on throttling.) Some forms of throttling supported by TWOS rely on committed information to determine how much optimism a given process is permitted. `objpast()` has earlier been accumulating a count of how many events this process committed, and how much processing time it committed. The code here adds up the time required to create all of the process' uncommitted states and counts the number of those states. It then uses a multiplicative parameter and an additive parameter to compare the committed and uncommitted statistics and determine how many more events (or how much more processing time) this process should be permitted.

Chapter 9: Dynamic Memory

The TWOS dynamic memory allocation and deallocation system permits users to expand their process' states by requesting additional memory. Since the request expands the state, and the expanded area is available to the user until he deallocates the space (or destroys the object), TWOS must maintain multiple versions of the space to properly and efficiently support rollback.

TWOS does not have access to page tables of any hardware it runs on, so the simple solution of managing single pages is not available. Similarly, TWOS does not have access to dirty bits[1] , nor the ability to swap memory segments to disk, nor the ability to trap writes. All of these limitations prevent some of the more simple and straightforward implementations of efficient multi-version dynamic memory allocation. The scheme used by TWOS must work within these limitations.

TWOS has the further limitation of being unable to map addresses issued by the user. Therefore, TWOS cannot present the user with a dynamic memory segment at the same address for every event. Instead, as described in the TWOS User's Manual, Section 4.6, the using process must keep an indirect pointer to each dynamic memory segment. Each event needing access to the segment must request a translation of that indirect pointer into a physical pointer, using the `PointerPtr()` call. This limitation actually permits TWOS to make up for some of the other limitations described above.

Each TWOS state may need its own set of dynamic memory segments. Some states will not need any segments at all. Some will need many segments, but will access only a few of them in any given event. Some will need many segments and will access all of them within an event. The TWOS dynamic memory system tries to provide the minimum use of memory and as little data copying as possible. The strategy is to make events that use dynamic memory segments extensively pay a little extra cost to avoid taxing the events that do not use dynamic memory at all.

Figure 9-1, a reproduction of Figure 2-2, shows how dynamic memory is attached to a TWOS state. Each state has a pointer to a dynamic memory table. If the previous event did not have any dynamic memory segments attached to it, this state starts off with a null pointer to its dynamic memory table. If the previous event had one or more segments allocated, even if it didn't access them, this state starts off with a pointer to its own copy of a dynamic memory table. Processes that never use dynamic memory thus do not incur the overhead of maintaining a dynamic memory table they never use.

---

[1] Dirty bits are a hardware supported method to determine whether a particular piece of memory has been written or not.

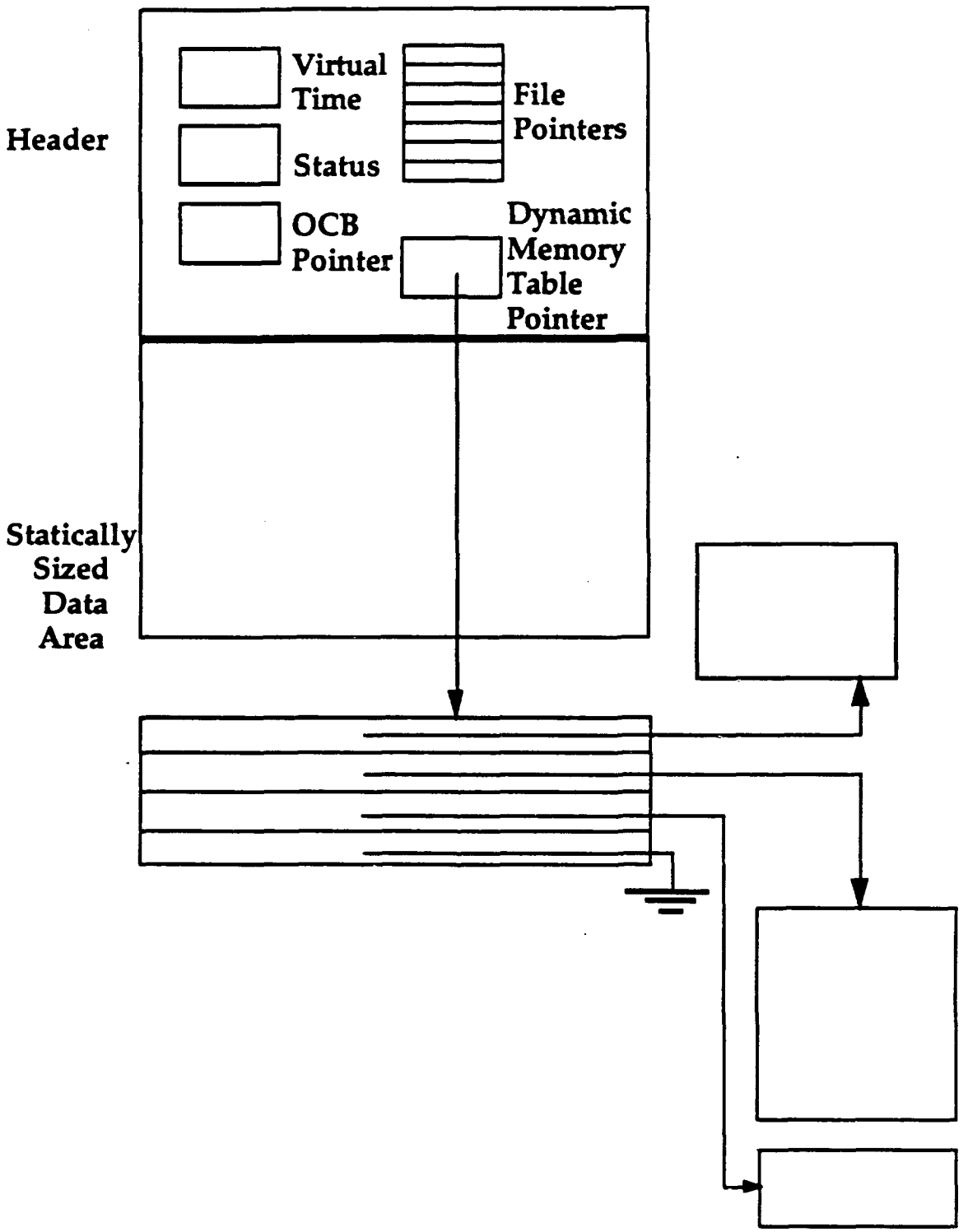


Figure 9-1: A TWOS State

The dynamic memory table shown in Figure 9-1 contains pointers to three dynamic memory segments and an unused entry. A newly allocated dynamic memory table for an event that had not yet started executing would not actually look like that. Figure 9-2 reproduces the dynamic memory table for this state, as it would actually appear when the state and memory table were first allocated.

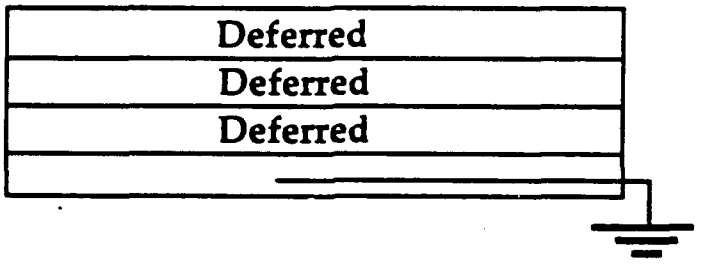


Figure 9-2: A Newly Allocated Dynamic Memory Table

Instead of having pointers to three differently sized memory areas, the table contains three markers indicating that the associated segments are deferred. The null, unused fourth segment is just as it was in Figure 9-1. TWOS does not allocate memory for any dynamic memory segments and does no copying when an event is started. Instead, the deferred markers indicate that, for the moment, the segment has not been accessed by this event, and the previous event's version of the segment is still the most recent version.

Figure 9-3 shows the same dynamic memory table further along in the event. The user has called `PointerPtr()` on the first and third segments, but not on the second. Note that the second segment is still in a deferred status, while actual memory has been allocated for the two segments that were accessed. The memory is allocated even if it develops that the user only reads the segment in question, and does not alter it. In principle, no new copy would be needed, in this case, since the contents have not changed at all. The limitations on TWOS that prevent trapping of writes and access to the dirty bits make it difficult to take advantage of this situation, however. The originally planned TW 2.7 would have contained the ability to request read-only copies of dynamic memory segments, which the user would have to treat in a disciplined way, since TWOS could not enforce read-only access. This method could have provided great advantages for some applications, but time was too short to include it.

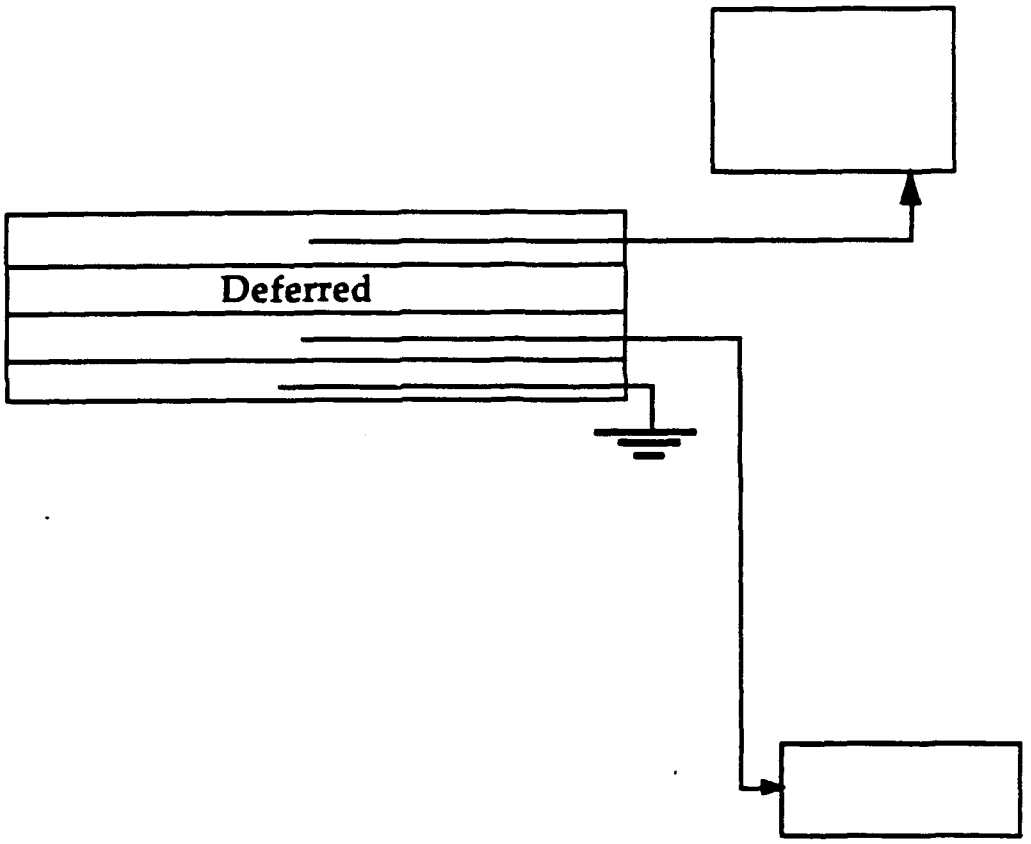


Figure 9-3: A TWOS Dynamic Memory Table In the Middle of an Event

TWOS' deferred copying method has two advantages. First, less time is spent at the beginning of events allocating and copying memory segments that frequently are not used. Second, less memory is used on unnecessary copies of dynamic memory segments. The cost is somewhat higher overhead for copying an event's required memory segments piecemeal. This added cost is usually quite low, being no more than a context switch into and out of the kernel for each memory segment. If a state had a large number of memory segments, and used all of them in an event, copying them all at the beginning of the event would be more efficient. Experience with simulations run under TWOS suggests that most events access only a subset of their process' allocated memory segments, so the solution chosen is more efficient for the applications of interest.

The remainder of this chapter will trace the life cycle of a typical dynamic memory segment from its allocation, through several events, until its deallocation.

The most basic way to allocate a dynamic memory segment is the system call `newBlockPtr()`, described in the TWOS User's Manual, section 4.6. This call

allocates a memory segment of a size requested by the user and returns an indirect pointer to it. This indirect pointer is actually an offset in the calling process' dynamic memory table. (More precisely, it is an offset into the dynamic memory table of the state for the event currently running for the calling process, but the order of segments in a given process' dynamic memory table is never changed, so the imprecision is not significant.)

All `newBlockPtr()` does is perform certain timings and set up a context switch from the user stack to the system stack. `newBlockPtr()` sets up this context switch using `switch_back()`, providing `newBlockPtr_b()` as a parameter telling `switch_back()` which system routine to perform. When TWOS returns control to `newBlockPtr()`, that routine can return the indirect pointer to the user, since control won't come back until the segment has been allocated.

`newBlockPtr_b()` does the work of setting up the segment. First, it checks for valid sizes. TWOS does not permit allocations of more than 100,000 bytes, and negative sizes are also illegal. Also, TWOS permits only a certain number of dynamic memory segments for each state. This limit is set at compile time, and is 100, by default. An attempt to allocate a 101st segment is flagged as an error, just as trying to allocate an illegal size is. `object_error()` is used to flag both these conditions. Since the error might be caused by incorrect optimistic execution, TWOS should not shut down immediately. Instead, the state of the process currently running is marked as being in error and the process is not scheduled again unless the error is rolled back. If the error is committed (see Chapter 8), then TWOS will halt and print an appropriate message, since at that point TWOS can be certain that the error has not occurred because of optimistic execution, but because the user actually made it.

If the request is valid, `newBlockPtr_b()` must first check to see if this state has an address table attached or not. If no other dynamic memory segments have been requested by this event, and the previous event performed by this process (if any) did not have a dynamic memory table, the state for this event does not have a table, either. Before a memory segment can be allocated, the state must have a table to store its pointer.

TWOS does not have a single static size for dynamic memory tables. Some processes need only one or two segments, while others need dozens. Allocating a table of the largest size any process will need for all processes is wasteful of memory, which is always in short supply when running TWOS. Therefore, it is not only possible that the state doesn't have any memory table, but also that the state has a table, but all its entries are full, so a larger one is needed. In either case, `newBlockPtr_b()` figures out how many entries, new or additional, are needed and allocates a chunk of memory of the correct size using `m_create()`. If the memory is not available, the event is rolled back. If

the memory is available, it is cleared. At this point, if a too small table existed, its entries must be copied into the expanded table just allocated, and the smaller table deallocated. Whether a table existed already or not, the new table's address is stored in the state's dynamic memory table pointer.

Next, `newBlockPtr_b()` must find an empty entry in the table. Since users are permitted to deallocate segments, in any order, the entire table is searched, rather than simply maintaining a pointer to the last entry used. Any entry with a null pointer is currently not in use, so `newBlockPtr_b()` finds the first such entry. Then `newBlockPtr_b()` calls `m_create()` to allocate the actual segment. This allocation might also fail, in which case the event must be rolled back. (If this allocation fails, the dynamic memory address table is not deallocated, even if there are no entries in it.) If the allocation succeeds, then the resulting real pointer is stored in the dynamic memory address table entry selected earlier and the new segment is cleared. (Clearing memory provided to the user by TWOS is a common theme in the system, as uncleared memory may contain garbage that user code can accidentally examine. This situation can lead to non-determinism in identical runs, and is very difficult to debug. Hence, TWOS takes care that all memory provided to users is always cleared when the user first gets it.)

`newBlockPtr_b()` now calls `dispatch()`, a routine discussed in Chapter 4 that chooses which process to run next. In most cases, the process requesting the memory allocation is still the process with the lowest virtual time event, so it will be chosen and its event resumed from the point of the allocation. However, if some other process received a message while the first process was running its event, that other process may be chosen. If `dispatch()` does not choose the process that requested the allocation, that process is suspended at the point of its request. This suspension does not cause the process to be rolled back to the beginning of the event, so, when its turn comes again, it can resume the event from the point it left off, unaware that other work has been done while it waited.

The return path to the process' code goes through TWOS' main loop (see Chapter 21), so not only might other processes be scheduled in front of the requesting process, but TWOS may choose to perform a number of system tasks before returning to user code. All of this activity will be utterly invisible to the process, of course.

Once the process regains control, it has an indirect pointer to its memory segment. Now the user's code needs to access that segment. The basic function for translating the indirect pointer to the current physical address of the segment is `pointerPtr()`. This function makes checks on the validity of its parameter (the offset). If the offset is invalid, `pointerPtr()` will return a null value, rather than itself calling `object_error()`. The user can choose to flag an error at that point or continue his code.

If the parameter is valid, `pointerPtr()` performs some timing code and checks to see whether the requested segment is deferred. The segment under discussion is not, since it was just allocated, so `pointerPtr()` uses the offset into the address table to find the proper physical pointer and returns it. Since finding this address was such a simple operation, there is no context switch into system space.

Figure 9-4 shows the condition of the example process' dynamic memory address table. All but one entry is null. The single non-null entry points to the segment we have just allocated.

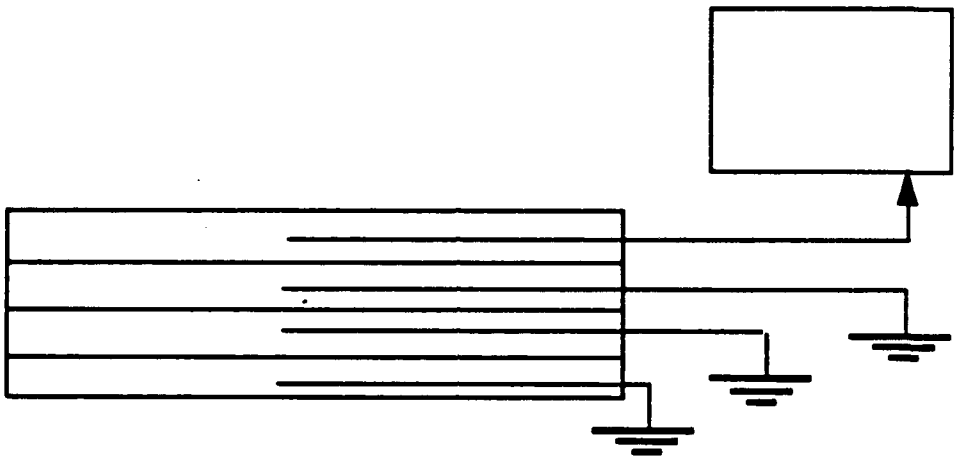


Figure 9-4: After Making the Allocation

Now the process runs a new event. As stated earlier, TWOS does not copy any of the existing memory segments at the start of the event. In `loadstatebuffer()`, a routine used to create a new copy of a state for an event (see Chapter 4), one action taken is to handle any dynamic memory segments. After successfully allocating the state itself, `loadstatebuffer()` checks to see if the previous event had a dynamic memory address table. If it did, `m_create()` is called to allocate memory for a new copy of the table. If the allocation fails, the event is rolled back, just as if there was no memory available for the state itself. If the allocation succeeds, the new table is cleared. Any non-null segments in the previous event's table are marked as `DEFERRED` in the new table. The other segments are left null. Figure 9-5 shows the situation after `loadstatebuffer()` is done.

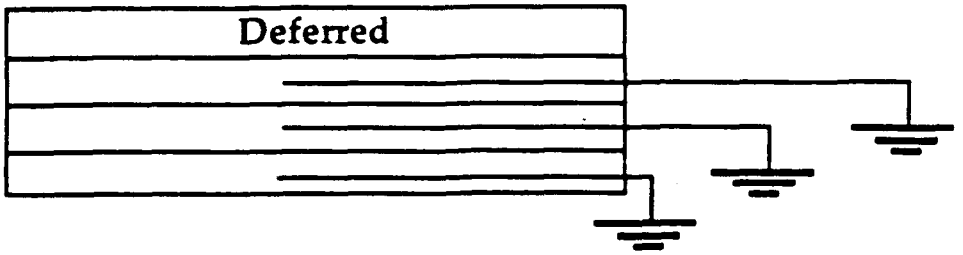


Figure 9-5: A New Table For a New Event

This event does not access the dynamic memory segment, and does not allocate any more segments, so the table looks the same at the end of the event as it did at the beginning. When the next event starts running, its dynamic memory address table also looks like Figure 9-5.

This next event, however, needs to use the dynamic memory segment. So it calls `pointerPtr()` with the proper offset. However, there is, as yet, no valid pointer to return to the user, so instead of simply returning a pointer, `pointerPtr()` uses `switch_back()` to switch into system space and there calls `pointerPtr_b()`. `pointerPtr_b()` is only called when a `DEFERRED` marker is in the requested offset. `pointerPtr_b()` must look at the previous state to find an actual copy of the memory segment. In this example, the previous state does not have such a copy. More generally, the segment might not have been accessed for an arbitrary number of events. If so, every one of those events' states' dynamic memory address tables has a `DEFERRED` marker in that entry, so `pointerPtr_b()` simply goes back through the state queue until it finds a state that has a real copy of the segment. When the last accessed copy is found, `pointerPtr_b()` uses `m_create()` to allocate a new copy, stores the pointer in the current state's proper dynamic memory address table entry, then uses `entcpy()` (a fast copy routine) to copy the values of the old version of the segment to the new version. If the allocation fails, the running event is rolled back. In any case, `pointerPtr_b()` returns through `dispatch()`, giving the system a chance to do something else.

Figure 9-6 shows how the table would look after the call to `pointerPtr()` returns. Note that it looks exactly the same as Figure 9-4.

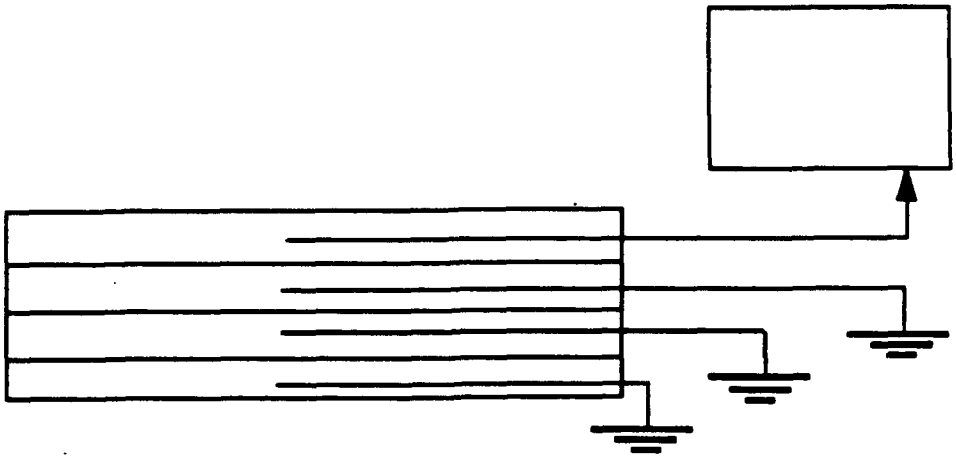


Figure 9-6: A Fresh Copy of the Segment

The next event decides to deallocate this segment. It calls `disposeBlockPtr()` to do so. After making the usual timings, `disposeBlockPtr()` uses `switch_back()` to call `disposeBlockPtr_b()`. This routine performs the error checks and, if they are passed, looks into the dynamic memory address table for the specified entry. If this event has not yet accessed this entry in any other way, all that is there is a `DEFERRED` marker, in which case the entry is simply set to null. On the other hand, if the entry has been accessed by this event, then there is an actual pointer to a copy of the data in the entry, as shown in Figure 9-6. In this case, the memory holding that copy is released before nulling the pointer. Control returns through `dispatch()`, as usual.

When `disposeBlockPtr()` returns, the segment is gone as far as this event and any future events at later virtual times are concerned. The slot in the table may be reused for the next segment allocated, but there is no logical connection between the segment just disposed of and the next segment to be allocated. Figure 9-7 shows the situation after the deallocation. Note that the dynamic memory address table still exists, even though it has no non-null entries. TWOS does not deallocate the table in such cases, so a process that once allocates a dynamic memory segment will have an address table for the rest of the run, even if the segment was immediately deallocated and no other segment is ever allocated by the process.

Deallocation has no effect on any saved states with earlier virtual times than the deallocating event's. If these earlier states are required due to rollback, they will still have access to their proper copy of the dynamic memory segment.

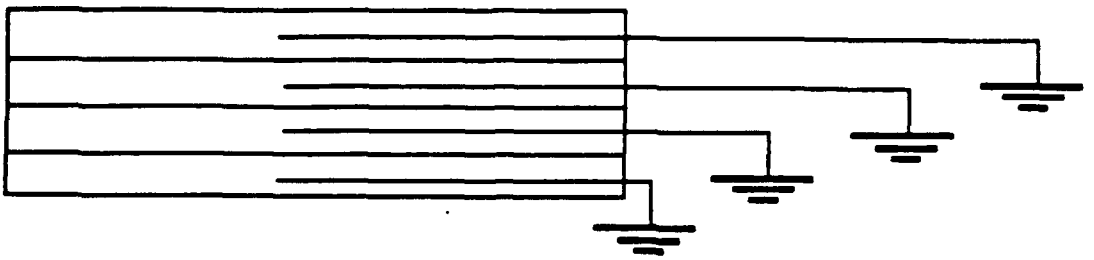


Figure 9-6: A Fresh Copy of the Segment

One other area of importance for dynamic memory segments is commitment, covered in detail in Chapter 8. When committing states for a process, it is important not to release the latest physical copy of a dynamic memory segment. `objpast()`, when freeing dynamic memory segments attached to states being fossil collected, first checks to see if any of the segments are deferred in the next later state of the process. If they are, then the segment is transferred to that state, its physical pointer replacing the `DEFERRED` marker in the proper entry of that state's dynamic memory address table. If the next later state has its own copy of the segment, or if it has a null entry, indicating that the segment was deallocated during that event, `objpast()` simply frees the memory used by this copy of the segment.

Dynamic memory causes substantial complications in phase migration. These complications will be discussed in detail in Chapter 13.

Chapter 10: Message Sendback

The message sendback facility deals with problems of limited memory on nodes running a Time Warp simulation. Message sendback is a limited form of the more general cancelback protocol that was to go in a future version of TWOS. The fundamental idea behind both message sendback and the cancelback protocol is that if the node doesn't have memory to hold a message or state that you need, make room by throwing away something you don't need as much. Message sendback only throws away input messages, while cancelback can discard input messages, output messages, and states. The criteria for determining what is needed most is based on characteristic virtual times for allocations. Discarding data for sendback or cancelback causes rollback. Eventually, the events that were rolled back for this purpose will re-execute, regenerating the discarded data.

TWOS allocates memory for many purposes. Some of these are systems related, and such allocations typically do not invoke message sendback. Sendback is generally used to satisfy allocations made as a result of user requests. These include sending and receiving messages, creating a state to run an event, allocating or accessing a dynamic memory segment (see Chapter 9), and creating a new object (see Chapter 14). Additionally, message sendback is used for certain allocations related to process migration (see Chapter 13). Generally, anywhere in the code where memory is allocated with the routine `m_create()`, message sendback might be invoked. This chapter will not detail every case where sendback might occur, but will cover the most basic case, handling an input message when memory is low. The other cases cause analogous actions.

Time Warp has two major classifications of messages: system messages and user messages. System messages deal with such activities as GVT computation, process migration, phase location, and simulation end. User messages are the more familiar stuff of Time Warp: event messages, dynamic creation messages, and dynamic destruction messages.

Incoming system messages are put into a special buffer. Only one system message is dealt with at a time, and it is dealt with to completion before the system goes on to do something else. Since they exist only in this special permanent buffer, system messages do not require the allocation of any memory. Hence, they do not cause message sendback, and are of no further concern to us here.

User messages are placed in a different buffer, upon arrival. More precisely, when the system determines that it should read a message provided by the low level message system, it copies the message into a pre-allocated piece of memory pointed to by `rm_buf`. When the time comes to actually handle the message, the `rm_buf` pointer is copied to `rm_msg`. `rm_msg` can hold only one

message pointer. When a user message arrives, the system must determine what to do with it, and get it out of the buffer as quickly as possible. Assuming that this is an input message, the routine that makes the determination is `nq_input_msg()`.

In some cases, dealing with the input message can be quite easy, and will require no memory allocation. In particular, if the new message annihilates a message already existing in the destination process' input queue, no new memory will be needed. Rather, the memory of the message in the queue will be freed, and the buffer can be immediately cleared. (This annihilation may cause rollback, which is covered in Chapter 5. Rollback may cause the freeing of further memory.) Annihilation will occur when a new positive message matches an already queued anti-message, or, more commonly, when a newly arrived anti-message matches a queued positive message.

More often, though, the new message is not the negative copy of a queued message. In this more common case, `get_memory_or_denymsg()` is called to put the new message into the destination process' input queue. This routine calls `m_create()` to get a new message buffer. If successful, the incoming message is copied into the new buffer and the `rm_buf` copy is then available for use by another message. The `rm_msg` pointer is nulled (showing that the node is ready to handle another input message), and the input message is linked into its queue.

The problem arises when no memory is available to make a local copy of the message pointed to by `rm_msg`, meaning that no more messages can be accepted until this one is dealt with. Since the local node may be unable to free any memory until the next message arrives, and since that message's virtual time may be earlier than that of the input message currently being handled, the system must not simply link the buffer the message sits in into the receiving process' input queue, as that buffer will be needed to accept the next message. Therefore, if it turns out that there is no space for the next input message, the system must call message sendback.

When message sendback is called to make room for an input message, the system may determine that the best thing to send back is the input message itself. Therefore, one way that space may be freed for the next input message is by rejecting the present one.

Whenever memory is to be freed for message sendback, the system must choose the item that is the best candidate for sendback, based on some characteristic virtual time. For sendback caused by an input message, this time is its receive time. For sendback caused by an output message, this time is its send time. For sendback caused by a state, the time is the time of the event needing the state.

m_create() accepts three parameters. The first parameter is a description of the amount of memory needed. The second parameter is a virtual time characteristic of the allocation about to be made. The third parameter is a flag indicating whether the requested allocation is immediately critical to the success of the run or not. This flag is only set to true if failure of the allocation means that the system can take no further action.

m_create() has all of its important code inside an infinite loop. It tries to allocate memory (using the lower level l_create() routine) until the allocation is satisfied, or no more memory can be freed. When either condition is fulfilled, m_create() returns, with a pointer to the block of memory, if obtained, or a failure code if the memory could not be obtained.

When m_create() is first called, there may be free memory available for use. The first thing m_create() tries to do in its major loop is a simple allocation of free memory of the size requested. If that allocation fails, make_memory() is called. If make_memory() succeeds, then control returns to the top of m_create()'s loop, to re-attempt to allocate the newly freed memory. There may be enough, but, if not, make_memory() will be called again. If make_memory() fails before enough memory is obtained, then, if the critical parameter is set, m_create() traps to the tester. If the critical parameter is not set, m_create() returns failure. (There is other code in m_create() before the failure return. This code is used to help detect fatal memory shortages, and is not of direct relevance to message sendback.)

make_memory() takes a size and time parameter, though the size parameter is ignored. make_memory() starts by calling get_message_to_send_back(). get_message_to_send_back() is the heart of the sendback code, as the sendback policy is administered in this module. This routine is passed a virtual time, the characteristic time of whatever TWOS is trying to allocate memory for. If this time is less than the minimum local virtual time, get_message_to_send_back() looks for something that exceeds that time, rather than the virtual time passed as a parameter. This check ensures that no pre-GVT message will be sent back.

get_message_to_send_back() loops through all processes stored at this node. For each, the routine tries to find in the process' input queue the message whose send time is greatest, such that its receive time is greater than the virtual time we are considering. It does not choose an antimessage. If it finds a message that meets these criteria and is further ahead in its send time than any other message so far seen, the routine registers that message as the current best candidate for sending back.

After looking through all local processes' input queues, the routine will have chosen the best candidate for sendback, if any input message at all qualifies. The message chosen, to recap, should be the positive message with the

maximum virtual send time such that its virtual receive time is greater than the characteristic time of the item we are trying to free space for. That message is removed from its input queue and has its reverse bit set. `rollback()` is called to roll back the process that lost the input message. (For details on the effects of `rollback()`, see Chapter 5.) The process whose input queue held the chosen message may need to roll back to the virtual receive time of the message being sent back, if it had already passed that time. A pointer to the message to be sent back is returned.

The logic behind the choice of message to send back is that, first, it makes no sense to send back a message whose receive time is not greater than the characteristic time of the thing TWOS is making space for. If the system did choose such a message, it would need it again, very soon. Given that the receive time is greater than the characteristic time of what TWOS is making space for, the choice is based on the maximum send time. The rationale behind this criterion is that choosing such a message causes a process that is far ahead to roll back, rather than one that is at nearly the same virtual time. Since receive time can never be less than send time, this criterion will probably not cause TWOS to choose a message needed soon locally over one that will not be needed for a long time. Antimessages are not chosen because they are temporary residents in the input queues. The only reason they are there is to await the arrival of their positive counterpart. Once that counterpart arrives, the antimessage will annihilate, freeing the space it occupies. Sending an antimessage back would delay the process of annihilation, since the message-antimessage pair are, in effect, chasing each other.

Only one message is chosen for sendback at a time. Sendback can cause rollbacks, and certainly will cause duplication of effort, so as few messages should be sent back as possible. When a message is sent back, the amount of memory made newly available may be more than that taken up by the message itself. The message may take up space next to a free block, and the deallocation of that space may cause a free block large enough for the request to be created, even though the message itself was smaller than the size of the request. Also, sending back a message may cause a local rollback that will free up more space. Thus, only one message is sent back, after which the system tries to perform the allocation again. Should the allocation attempt fail again, then another message will be chosen for sendback.

`get_message_to_send_back()` returns the chosen message's pointer to `make_memory()`, which then calls `send_back()` for the message. The actual process of sending a message back is quite simple, especially from the sending end. Each message and antimessage in the system has a direction bit. When the message is going from the original sender to the original receiver, this direction bit is set to `FORWARD`. If the message is ever sent backwards, the

direction bit is flipped to REVERSE by get_message_to_send_back(). send_back() merely calls deliver(), the normal message sending routine.

deliver(), discussed in Chapter 3, is a general purpose routine for routing messages. It checks the direction bit and, seeing that it is set to REVERSE, routes the message to its original sender.

Reverse messages are received in basically the same way as normal messages. When deliver() on the receiving node looks at the message, though, it notices that the message is a reverse message, rather than a forward message, so nq_output_message() is called, instead of nq_input_message().

Reverse messages are not actually put into the output queue, however. If their matching message of the opposite sign is in that queue, the two messages annihilate, possibly causing a rollback. If not, then the negative copy must have been sent to the original receiver to cancel this message being sent back. In this case, the non-annihilating reverse message is re-delivered to the original receiver, in the hopes that it will be able to annihilate when it gets to the receiver, who, presumably, now has the matching message of the opposite sign.

If get_memory_or_denymsg() cannot get a buffer for an incoming message, make_memory() next tries to free up any eligible buffers in the free message pool. TWOS maintains a pool of unused message buffers, in order to make message buffer allocation quick. If memory is very low and no other memory is available, that memory can be freed. If make_memory() can either send a message back or free some buffers from this pool, m_create() gets a success signal. Otherwise, m_create() fails, and returns failure to the routine that called it.

In the case of failure to enqueue an arriving input message, that message itself is a candidate for sendback. Its direction is reversed and it is passed to denymsg(). This machine-specific routine ensures that the message will go back to where it came from. deliver() cannot be used because the message in question is sitting in a special system buffer that requires handling deliver() cannot provide. This incoming message might be a negative message, leading to the only case in which TWOS will send back a negative message. Normally, negative messages annihilate at their destination, so the problem does not arise, but in extraordinary cases, a non-annihilating negative message may have to be sent back.

The process is similar for any other allocation. For states or messages that are sent out of an event, if message sendback fails to get enough memory for the requested allocation, the event making the request is rolled back. TWOS must take care to recognize that certain allocations are linked. For instance, when an event sends a message, TWOS needs to allocate two buffers, one for the positive copy and one for the negative copy. (See Chapter 3 for more

details.) If the first allocation succeeds and the second fails, not only must TWOS roll back the event, but it must deallocate the buffer for the first copy of the message. This theme occurs several places in the TWOS code, including object creation and making a new copy of a state.

Message sendback never considers discarding states or output messages. The correct, more general way to free memory when TWOS runs short is described in the cancelback protocol [Jefferson 90], which has not yet been implemented in TWOS.

Chapter 11: Dynamic Load Management

Dynamic load management relies on process migration, which is discussed in Chapter 13, and on temporal decomposition, discussed in Chapter 12. Given that processes can be split and moved, however, substantial work is still necessary to ensure that the right processes are chosen to best balance the load. This chapter discusses the code that makes those decisions.

TWOS dynamic load management works in cycles. Periodically, one processor, called the load master, starts a dynamic load management cycle. Using a protocol similar to that used to collect GVT information (see Chapter 7), the load master requests that all processors calculate their local load and return that information to the load master. Once the load master has gotten complete load information from all processors, it passes the information back to all processors, again using the method used by the GVT protocol. Unlike the GVT protocol, the dynamic load management protocol works in only two waves, one collecting information, one disseminating it. The load management protocol uses the same graph set up for GVT computation, and does not distribute its final information through the more efficient binary tree set up by the GVT initialization code. (Though it could use the tree.) This chapter will not go through the details of how each routine in the protocol connects to the next routine, or what is done in each routine, as they are so close to the pattern of the GVT protocol, which was discussed in complete detail in Chapter 7.

The basic metric used to calculate relative load of the processors in a TWOS run is *effective utilization*. A complete description of this metric can be found in [Reiher 90a], but, informally, it is the proportion of time a process or node spends performing work that is committed. Processes with high effective utilizations are contributing much to the progress of the simulation, while processes with low effective utilizations are contributing little. If the overall utilizations of the nodes of the processor are highly imbalanced, the system is not applying its resources in the best possible manner, so load should be shifted to bring the nodes into balance.

TWOS uses an estimator of effective utilization, since the system cannot know if an event will be rolled back until it is committed, but events may be committed long after they are processed. The TWOS effective utilization estimator is calculated by keeping track of how much real time was spent running each event. Each process keeps a running total of how much time it has spent running events since the last load management cycle. It also keeps track of the amount of time spent running events that were rolled back during this cycle. Events rolled back during this cycle were not necessarily processed during this cycle, so, in certain cases, the amount of work rolled back in a cycle may exceed the amount of work done in a cycle. A process' effective work is the work done during the cycle minus the work rolled back

during the cycle. Its effective utilization is its effective work divided by the length of the cycle. This is the number reported back to the load master. The load master constructs a table of the effective utilizations of all processors and ships that table out to all other nodes.

As each processor receives the effective utilization table, it scans the table to see if it is an overloaded node. TWOS permits the *n* most overloaded nodes to migrate work off during a given load management cycle, where *n* is a parameter the user can set at runtime. (The default value is 8.) If a node is one of the *n* most loaded nodes, it finds the corresponding underloaded node. The most overloaded node matches up with the least loaded node, the second highest load with the second lowest, and so on. If the difference between the two loads in a particular pair is greater than some lower threshold (also a parameter that can be set at runtime), the overloaded node will consider moving some load to the underloaded processor.

The determination of whether a given processor will migrate load or not, and, if it will, to where, is made in the routine that accepts a table of system loads in the second phase of the load management protocol. This routine is called `loadUpdate()`. `loadUpdate()` runs through the table that has just arrived, marking the *n* highest and lowest pairs of utilizations, until it has found itself as either a high or low utilization node. If it is a low utilization node, it will not also be a high utilization node, so it need not make any further comparisons. If it is a high utilization node, it needs to check to see if it should move some work to its low utilization counterpart.

While dynamic load management is used by default in TWOS, much experimentation with it remains incomplete. The state of the dynamic load management code in TW 2.7 reflects that situation. There are several places in which different methods for how to perform dynamic load management functions are possible. The existing code often supports several of those methods, depending on either compile time or run time switches. One such case is the method used to determine if a given pair of utilizations (high and low) are sufficiently mismatched that a migration should be performed. TWOS supports two methods, one of which looks at the difference between the two utilizations, the other of which looks at their ratio.

Comparison by difference is used, by default. This method subtracts the lower utilization from the higher and compares it to some minimal necessary difference. (This minimal difference can be set at run time from the configuration file, and is .1, by default.) If the difference is greater than the minimal difference, a migration is permitted.

Comparison by ratio can be set from the configuration file. If ratio comparison is used, after making checks to ensure that we are not dividing by zero, and that the possible negative status of one or both of the utilizations will not cause problems, then the ratio of the high utilization to the low

utilization is compared to some minimal ratio (also settable from the configuration file). If the ratio is greater than the minimal ratio, a migration is permitted.

If a migration is permitted, `chooseObject()` is called to determine what to migrate. It takes the underutilized node's effective utilization as a parameter. (Since the code is running on the overutilized node, `chooseObject()` will have the overutilized node's utilization locally available, so it need not be passed as a parameter.)

In certain cases, an overloaded processor will not be permitted to order a new migration. If it is not one of the *n* most overloaded processors, it cannot migrate, no matter how heavily loaded it is. If it cannot find a suitable process to move, no migration will be ordered. Also, TWOS can only perform one outgoing migration per processor at a time. More migrations may be queued up, but only the migration at the head of the queue is active. Leaving many processes lying around in the migration queue can adversely affect performance, so TWOS will not permit an overloaded processor to order another migration if it already has several queued up waiting to move. `chooseObject()` checks for this condition by calling `numMigrating()`, which counts the length of the local migration queue. The result is compared to a value held in `maxMigr`, and, if the queue is already too long, `chooseObject()` returns `NULL`. The maximum permitted migration queue length is another parameter that can be set at run time in the configuration file. The default for this parameter is 1, so at most one process will be migrating off a given node at a time.

`chooseObject()` must next choose what to move. The node will move at most one process, as there is a high per-process overhead for migration, so the overloaded node scans the utilizations of its local processes for the single process whose migration will best balance the two processors' utilizations. TWOS supports two methods for choosing this process. One, called `BEST_FIT`, tries to find the single process whose utilization would most closely balance the two nodes. The other, called `NEXT_BEST_FIT`, skips over the one chosen by `BEST_FIT` and looks for the next most likely process. (The theory is that the process that best fits the needs will frequently be doing a lot of good work locally, usually being the local process with the highest utilization. The next best fit might be a process that is not getting to do as much work as it could, because the best fit process is running, instead. So the next best fit process might do much more work on an underutilized node.) `BEST_FIT` is used by default. `NEXT_BEST_FIT` can be used, instead, by using a configuration file command.

The `BEST_FIT` strategy is checked in a routine called `bestFit()`. This routine looks through all local processes one at a time. It skips over the `STDOUT` process, and also skips any process that is currently being migrated

into this node. For eligible processes, `bestFit()` multiplies the utilization of the process by a small number (currently .1) and compares the result to the difference in the two nodes utilization. If moving this process will not make up for at least this fraction of the difference, then it is not a good candidate for migration. The idea here is that moving a process that accounts for less than 10% of the difference in two nodes utilizations is unlikely to pay off. (This is another experimental feature of dubious value, practically and theoretically. Time did not permit testing without it.) This minimum utilization threshold can only be changed by recompilation. If the process currently being examined passes all these tests, then its utilization is compared to that of all other processes so far examined to determine if this process is the one to best balance the difference between the two nodes.

After looking through all processes, one more test is made. If this process is the only one on the overloaded node to do useful work, migrating it cannot possibly help, so no process will be chosen, in this case. Otherwise, the identity of the best process found (if any) is returned to `chooseObject()`.

`chooseObject()` now decides whether to temporally split that process. TWOS typically tries to avoid migrating whole objects, since they are very large. Instead, TWOS migrates just a single phase of the object. (See Chapter 12.) Therefore, after selecting a process to move, TWOS must next determine where to split it.

TWOS can decide where to split a process in several different ways. The choice of which way to use is determined by a configuration file command. The default is called `NEAR_FUTURE`. This method splits the process near its current SVT, selecting the later phase to migrate. This form of splitting has the effect that the next event the process currently expects to run will be done on the new node. If the process later rolls back before SVT, the piece left behind on the overloaded node will have to do more work. Otherwise, all future work will be done on the underloaded node. Since typically processes have several messages and states with timestamps earlier than GVT, this method of splitting usually moves all data necessary to perform the next event, but does not move any data not necessary to perform that event.

`splitNearFuture()` finds the place to split, for this splitting method. If the current SVT of the process is not `POSINF` (indicating that it currently has no more work to do), `splitNearFuture()` chooses SVT as the split time. Otherwise, `splitNearFuture()` looks for the last state in the chosen process' state queue. If one is found, the split is done at that point. Otherwise, `splitNearFuture()` looks at the last input message's receive time (if any), and chooses that. (This last choice is somewhat archaic, as if SVT is `POSINF` and there are no states, there probably will be no input messages, either.) `splitNearFuture()` does one last check to make sure that it isn't about to order an illegal split, then it calls `split_object()` to divide the existing

process into two processes at the chosen point. (See Chapter 12 for details on the behavior of `split_object()`, and its effects.) If a split was performed, `splitNearFuture()` returns the later phase. Otherwise, it returns `NULL`.

`chooseObject()` can split its chosen process in other ways. `LIMIT_EMSGS` calls `splitLimitEMsgs()` to split the process in such a way that only a small number of input messages will be moved, thereby limiting the size of the migrating process, possibly at the cost of not moving the work that will be done next. `MINIMAL_SPLIT` calls `splitMinimal()` to split the process at the latest time of anything in its input, output, or state queues, thereby moving as little data as possible, again at the probable cost of not moving the work to be done next. `NO_SPLIT` simply breaks, so that the entire process will be moved.

Whatever method is used, `chooseObject()` then returns a pointer to the process to be moved. If no suitable candidate was found, it returns `NULL`, instead. If a non-null pointer was returned, `loadUpdate()` calls `move_phase()` to migrate it to the underloaded node. (See Chapter 13 for details of how the process is moved.) `loadUpdate()` then cleans up in preparation for the next load management cycle. The load master sets up a timer to go off when the next load management cycle should begin.

Chapter 12: Temporal Decomposition

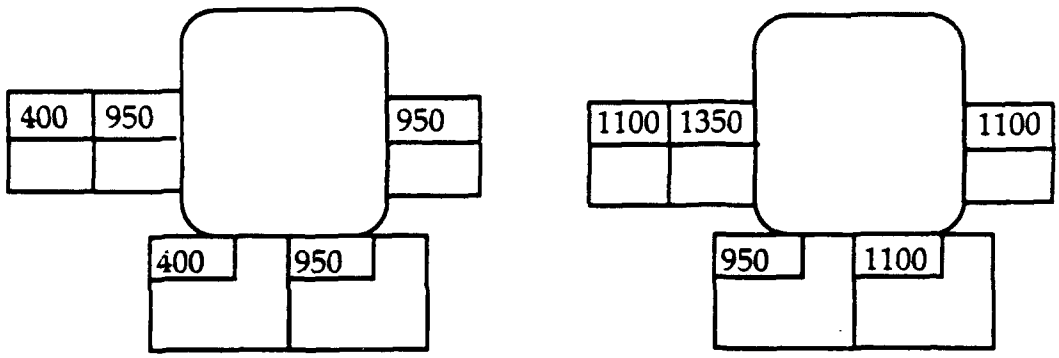
TWOS permits objects to be split into pieces, each of which is treated as a separate process by the operating system. Objects can be divided along virtual time boundaries. For instance, object A could be split at virtual time 1000, resulting in two different processes, one of which handled all events for A from the beginning of the simulation to virtual time 1000, the other of which handled all of A's events from virtual time 1000 to the end of the simulation. Each of these constituent parts of A is called a phase. Every TWOS object consists of one or more phases, each treated as a separate process. An object can be divided into an arbitrary number of phases.

Phases are invisible to the user, unless he examines a TWOS run in the middle of its execution. A simulation will always produce the same results regardless of how many phases its objects have been divided into. Since phases cannot affect the outcome of the simulation, their only purpose is to improve the performance of the simulation. Phases can be used for both static and dynamic load management [Reiher 90a], but the current version of TWOS does not automatically do any static load management. Chapter 11 discusses phases' use in dynamic load management. Phases are also used as a method of ensuring that a particular process' input queue never grows too large. (See Chapter 8.)

Figure 12-1 shows an example of an object divided into phases. Just as above, object A has been divided into two phases, at the virtual time 1000 boundary. Below each phase is its identity. The phase on the left is A(-∞, 1000), while the phase on the right is A[1000, +∞). The phase on the left handles all events for A up to, but not including, virtual time 1000. The phase on the right handles all events for time 1000 (inclusive) and later. Note that the phase on the right has a state for time 950, earlier than any event that phase is permitted to perform. Each TWOS event uses the state from the previous event as input. When A needs to perform its first event at or after time 1000, it will need a state from before time 1000 as input to that event. That is the purpose of the state at time 950. This state must be a completely correct copy of the last state at A(-∞, 1000).

This example suggests some of the difficulties of supporting phases in TWOS. The later phase must have a complete and correct copy of the last state of the earlier phase. (The copy of it at the later phase is called a *pre-interval state.*) If the two phases are on different nodes, then that state must be transported from one node to the other. Moreover, the later phase on the foreign node may have been running before the need to receive the copy of the pre-interval state was known, in which case it has performed events with the wrong state as input. Those events will have to be rolled back when the new pre-interval state arrives. Unless special care is taken to ensure that the two phases can never execute in parallel (and such special care would decrease

parallelism), the earlier phase may roll back and generate new pre-interval states for the later phase many times.



A(-∞,1000) A[1000,+∞)

Object A

Figure 12-1: An Object Divided Into Two Phases

The code specifically dealing with temporal decomposition is of two major classes. One class deals with the mechanics of splitting an object (or a phase of an object) into two phases. The second class deals with handling the special needs of phases during the run, primarily transport and reception of pre-interval states. This class also includes support code, such as object location code that is capable of dealing with phases of objects residing on different nodes, statistics code, and commitment code. This chapter will discuss both classes of code, but much of the code in the second class is discussed in more detail in the chapters devoted to other aspects of the system. For instance, most of the issues regarding phase location are covered in Chapter 17.

Objects and phases are split using the `split_object()` call. This routine takes two parameters, a pointer to the phase to be split and the virtual time of the split. `split_object()` first makes certain sanity checks, such as ensuring that the split time parameter is within the phase boundaries of the phase parameter. Also, `split_object()` will refuse to split any phase that is currently migrating into the local node. It will not split a null phase (see Chapter 14 for a description of a null object), nor a phase that is in the middle of sending a message for the user, nor a phase that has an error state. The latter two cases are prohibited to simplify phase splitting; with sufficient effort, the code could be made to support splitting of these types of phases

`split_object()` splits the later phase from the existing phase. The earlier part of the existing phase is left with the same OCB, while the later phase gets a new OCB. `split_object()` must divide the existing phase's input, output, and state queues between the two resulting phases, making sure that

each phase gets the queue entries relating to events it must handle. First, though, `split_object()` finds the state that is to be the pre-interval state of the later phase about to be split off. TWOS needs two copies of this state, one for each of the phases, so `split_object()` will try to make an extra copy before splitting the phase. `copystate()` is used to make the copy.

`copystate()` is a routine specifically for making copies of states for phase decomposition. It is prepared to allocate all necessary memory for the state itself, its dynamic memory segment address table, and all dynamic memory segments. It must also be prepared to deal with the possibility that the local node does not have enough memory to make a complete copy of the state, however. `split_object()` could simply fail, at this point, as splitting a phase is never of vital importance to the system. However, `copystate()` makes one further attempt before giving up. Generally, the later phase is presumed to need the pre-interval state more than the earlier phase needs its last state, as most often the earlier phase has run all of the events it already knows about. Unless it rolls back, it will never run again, and, hence, never need its final state again. So if `copystate()` cannot make a copy of the earlier phase's last state, it will simply steal the copy that already exists in the sending phase's state queue.

Life, unfortunately, isn't quite as easy as that for `copystate()`, The existing copy may have some deferred dynamic memory segments. (See Chapter 9.) Since the later phase will be off on a different node, it cannot be given a state with deferred segments. It needs actual copies of all dynamic memory segments. So if `copystate()` does steal the last state, it must check to see if there are any deferred segments. If there were, `copystate()` must allocate actual memory for them, and must copy their last valid values into the new memory before handing the state to the new phase. In the case of a stolen state, `copystate()` does not actually steal the state right away. It makes sure that the state is complete, then returns `NULL` to the calling routine, signalling that routine that it should steal the state, if necessary. If `copystate()` cannot even make the existing copy complete, it returns -1, signalling total failure.

If it develops that the earlier phase eventually rolls back and does need the stolen state, that phase can simply rerun the event creating the phase and thus create a fresh, identical copy. There are minor issues of maintaining correct statistics, since TWOS is careful to count states, and careful to realize that it should not count pre-interval states. The OCB of a phase that has lost a state in this way is marked so that the statistics gathering code will notice and adjust the count accordingly.

`split_object()` actually makes its copy of the state before doing any other work. If split_object() is unable to complete, for any reason, any work it does must be backed out. Making this state copy is the operation `split_object()` performs that is most likely to fail, so this operation is done first; if the state

copy fails, little other work has been done, so little time has been wasted on it and little extra effort is necessary to undo it. After successfully making the state copy, `split_object()` goes on to handle the rest of the split.

The first thing to do is create a new OCB for the later phase, using `mkocb()`. This operation can fail if the node does not have enough memory. If `mkocb()` does fail, `split_object()` is careful to release the state copy made by `copystate()`. If `mkocb()` succeeds, `split_object()` copies the name of the object into the new OCB, and fills in other fields of the new OCB. `split_object()` must now check to see if the split is before or after the splitting phase's SVT. If it is after, then the SVT of the new OCB should be positive infinity and this phase should be put at the tail end of the scheduler queue. If it is before, then that OCB's SVT should be the SVT of the splitting phase, since the later resulting phase is the one that will run that next event for the object. In this case, the new OCB inherits a great deal of information from the splitting OCB, including its control and run status, its pointers into queues, its stack pointer, and so on. (The splitting object might be in the midst of performing an event when split, so all of these fields might contain valid, and important, values.) Then, the new OCB should be put in the scheduler queue in the place occupied by the splitting OCB. Also in this case, the earlier phase should wind up blocked at infinity, since it has done all of its currently available work, so various fields should be set to reflect that status. Since it is represented by the splitting OCB, that OCB should be pulled out of its current position in the scheduler queue and put at the end of that queue.

Local object location information is also updated at this point. No other object location information is changed, yet, since the object location entries on other nodes should still direct any messages for the splitting phase to this node. Only the local node need be concerned about what OCB on the local node will receive the message. (See Chapter 17 for more complete details.)

Now `split_object()` will divide up the various queues of the splitting phase, using `split_list()`. First the state queue is split, and the previously copied pre-interval state inserted into the front of the resulting state queue for the later phase. If no fresh copy of the pre-interval state could be made, then at this point `split_object()` steals the state from the earlier phase. There are complexities concerning the fact that a dynamic creation or destruction of this object could take place in the vicinity of the split time, so care is taken to make sure that both phases are of the appropriate object type. (See Chapter 14 for details of how dynamic creation and destruction affects objects' types.) Next the input queue is divided up with `split_list()`, and finally the output queue is divided. `split_object()` then returns a pointer to the new OCB.

The other major issue for temporal splitting is dealing with rollbacks and pre-interval states. Issues arise on both ends of the split. The earlier phase may

roll back and generate a new final state. TWOS must recognize this condition and make sure a copy of the new state is sent to the later phase. On the later phase's end, the arrival of a pre-interval state must cause the phase to discard its existing pre-interval state and roll back to accommodate its new pre-interval state.

At the earlier phase's end, eventually when `go_forward()` attempts to schedule another event for this phase, it will discover that there is no such event. The earlier phase has reached the end of its interval. This condition is marked by setting its run status to `BLKINF`, just as if the phase represented an entire object that had no more work to do. When `go_forward()` finds a phase with a `BLKINF` run status whose phase end is not POSINF, it must ship a copy of that phase's final state to the next phase.

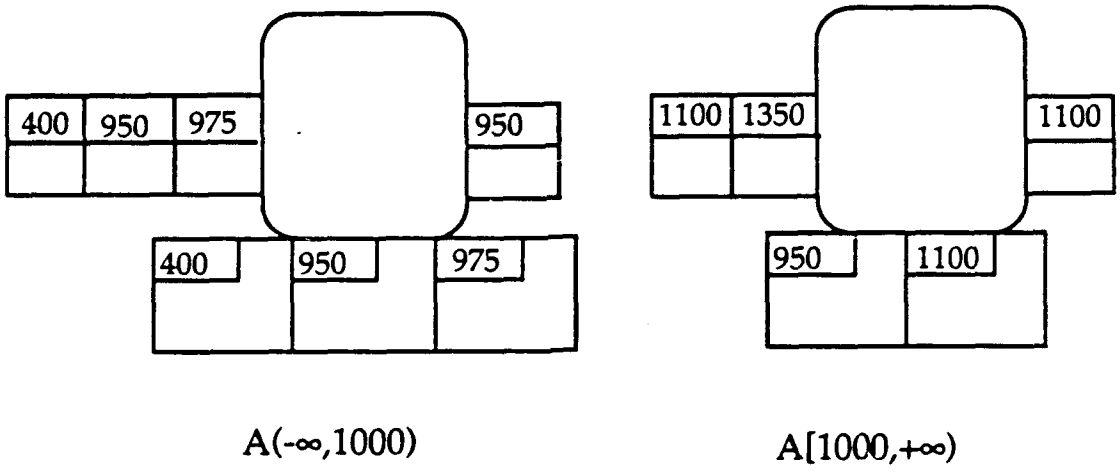


Figure 12-2: Sending a Pre-Interval State

Figure 12-2 shows an example. A(-∞, 1000) has rolled back and executed an event for time 975. The pre-interval state that A[1000, +∞) has for time 950 is no longer valid. Rather than using the time 950 state to run the event at time 1100, this phase should use the time 975 state. Therefore, A(-∞, 1000) must ship that state to A[1000, +∞).

`go_forward()` calls `send_state_copy()` to perform this task. `send_state_copy()` makes use of the limited jump forward optimization. This optimization recognizes that if the state about to be sent out is identical to the last state that was sent out, then there is no need to send this state out. In the case shown in Figure 12-2, such a situation is not too likely, but what if the 975 state were transmitted, the event for time 950 cancelled, the event for time 975 re-executed, and the resulting state exactly the same as the first time

the 975 event was executed? In this situation, resending the 975 state is a waste of time.

The limited jump forward optimization is done by a routine called `state_compare()`. `state_compare()` uses an OCB pointer to the last state a phase sent to compare to the state about to be sent. `state_compare()` is not a terribly bright routine, at this point. It can only handle simple states that have no dynamic memory segments; it cannot even handle empty dynamic memory segment address tables. For such simple states, `state_compare()` compares the two states, byte by byte. If they are identical, it returns 0 and the new state need not be sent. Otherwise, `send_state_copy()` goes on. There is a slight complexity at this point. The last state sent might no longer be a valid state for this phase, in which case it is no longer in the state queue. In this case, a special `out_of_sq` flag is set. Whether or not the state was sent, an out-of-state-queue state must be destroyed, at this point, using `state_destroy()`. In the future, a more complete version of `state_compare()` would have fully compared states of arbitrary complexity, but time did not permit completion of this routine.

If the state must actually be sent, `copystate()` is called to make a copy of it. This is the same routine called by `split_object()`, and it behaves in the same way. If memory permits, it makes a fresh copy of the state. If memory does not permit, it makes sure the existing copy has no deferred memory segments (see Chapter 9) and signals its calling routine that the state can be stolen. `state_compare()` either works with the copied state, steals the state from the earlier phase, or, if no sendable copy of the state can currently be made, sets the run status of the earlier phase to `STATESEND`. In the latter case, the earlier phase will keep trying to send the state until it succeeds, or until the system has determined that there will never be enough memory to send it, in which case the system halts. (The repeated attempts to resend the state are made from `dispatch()`, a routine discussed in Chapter 4. Whenever the sending phase's virtual time for the current event is the lowest virtual time of any phase on the node, `send_state_copy()` will be called again.)

Assuming that a sendable copy of the state is available, the run status of the phase might be either `BLKINF` or `STATESEND`. In the latter case, an earlier attempt to send the state failed, but this one will succeed, so the phase's run status should be set to `BLKINF`.

Now `send_state_copy()` is ready to get the state to its destination. One difficulty remains. The destination phase may only exist in the local migration queue. (See Chapter 13 for a complete description of this queue.) `send_state_copy()` contains code that has been ifdefed out that permits the pre-interval state to be inserted into the state queue of the OCB in the migration queue. This code probably works, but had been compiled out when it was uncertain if it was causing problems. The problem in question was

eventually found elsewhere, but this code had not yet been returned to the operating system. Time does not currently permit to put it back in and test it thoroughly. In its absence, a much more inefficient set of actions will occur, but they should be correct.

`send_state_copy()` now needs to find the destination phase for the pre-interval state. It calls `GetLocation()` to do so. `GetLocation()` is completely discussed in Chapter 17. For the moment, it suffices to know that it returns any locally stored information about the phase's location, or `NULL` if it does not have local information. If local information is available, `send_state_copy()` calls `finish_send_state_copy()`. If no local information is available, `FindObject()` is called, instead. (Chapter 17 also discusses `FindObject()`.) `finish_send_state_copy()` will eventually be called when local information becomes available.

`finish_send_state_copy()` first checks to see if the destination phase is local. If it is, `put_state_in_sq()` is called, then `rollback_state()` is called to roll back the destination phase. Both of these routines are rather general purpose, and are also used by phase migration, so they will be more completely covered in detail in Chapter 13. Briefly, `put_state_in_sq()` replaces the existing pre-interval state of the destination phase (if one exists) with the new one, and `rollback_state()` ensures that all necessary rollback related actions are taken.

A few further details on `rollback_state()` are worth mentioning here. The destination phase either has input messages enqueued or it does not. If it does, all that need be done by `rollback_state()` is to call `rollback()` with the receive time of the first of those messages. Any events performed by this phase will have to be rolled back and redone, since they were performed with an improper state. If there are no input messages, complexities arise if the phase is still migrating in. In this case, the time currently migrating in might have some input messages. If so, then SVT must be set to signal that this virtual time should be reperformed once its migration completes. (For further details, see Chapter 13.) If the phase has completed migration, however, and there are no input messages, then one further condition must be checked. Figure 12-3 shows the case of interest.

A has been split into three phases. The earliest phase generates a new pre-interval state and ships it to the middle phase. That phase has no input messages, at the moment, so it should ship the pre-interval state to the third phase. In this case, `rollback_state()` must forward the incoming pre-interval state to the third phase, using `send_state_copy()`.

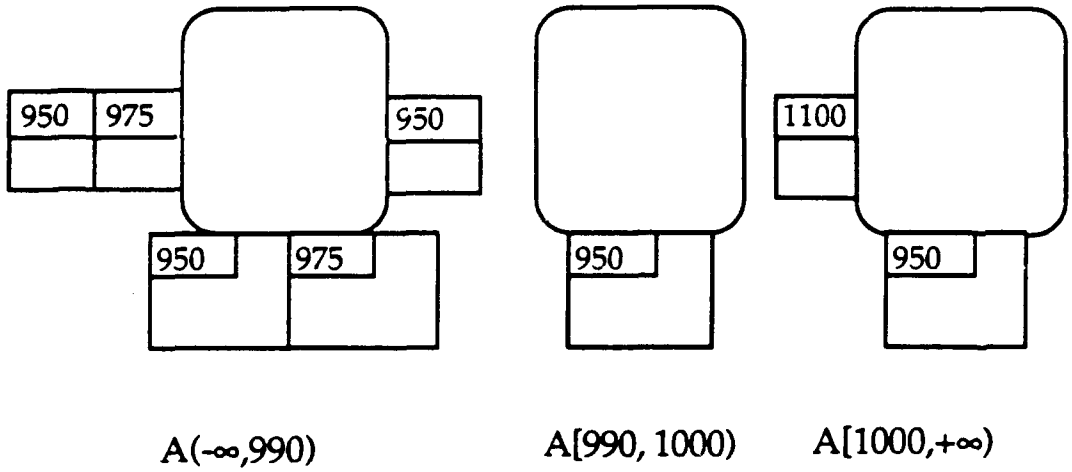


Figure 12-3: Pre-Interval State Problem For An Object Split In Three Phases

If the destination phase is not local, `finish_send_state_copy()` marks the state as a pre-interval state and checks to see if the phase is in the migration queue. If it is, then the state is marked as needing to wait for the migration to complete. `finish_send_state_copy()` then calls `send_state()`. `send_state()` is heavily used by the migration code, and will be completely covered in Chapter 13. In this situation, it causes the state to be stored in the queue of migrating states (a different queue than the queue of migrating OCB's). Eventually, by a process described in Chapter 13, this state reaches the head of that queue and is shipped to its destination node, by another process described in Chapter 13.

Once the complete state has been received at the destination node, `recv_state()`, a general purpose module for handling all kinds of migrating states (covered in detail in Chapter 13), calls `put_state_in_sq()` and `rollback_state()`, the same routines used for local delivery. Both perform substantially the same actions as if the destination phase had been local.

Chapter 13: Process Migration

TWOS migrates processes for purposes of dynamic load management. The process can be an entire object, or a partial object. Phase decomposition is described in Chapter 12, and dynamic load management is described in Chapter 11. This chapter is concerned exclusively with the mechanics of moving a process from one node to another.

Process migration is a rather complicated business. (The C module containing the bulk of the migration code, `migr.c`, is 3500 lines long.) This chapter will not exhaustively cover the totality of the migration code, but will concentrate on the more normal paths through it.

TWOS' default dynamic load management policy permits only one process per node to migrate at a time (see Chapter 11), but the migration mechanism is prepared to permit multiple processes to await migration simultaneously. However, only one process per node is actively migrating at a time. The others merely wait in the queue for their turn.

Process migration is initiated by calling `move_phase()`, which is given the OCB pointer of the process to migrate and the destination node number as parameters. `move_phase()` makes some error checks. (TWOS cannot move a process to the node it's currently on, cannot move a process before the simulation starts or after it terminates, cannot move a process that is currently migrating already, cannot move a process that is in the middle of sending a message, and cannot move a process containing an erroneous state.) If those checks pass, `move_phase()` removes the process' OCB from the scheduler queue. If the process was executing an event, the node's record of what process was executing is erased and the partial event is rolled back, which involves deallocating the state, the stack, and the message vector (see Chapter 4). The process is then put into `BLKPHASE` run status and put at the end of a local queue of migrating processs.

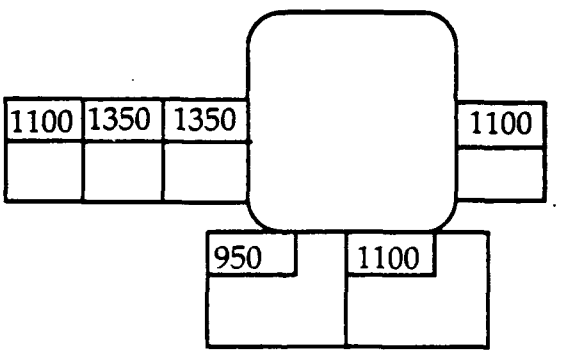
`move_phase()` counts the number of states and messages to be moved. It determines which node serves as the home node for the process' object (see Chapter 17 for a description of home nodes). If this node is the home node, it calls `ChangeHLEntry()` to locally change the object location data structures. If the home node is somewhere else, `move_phase()` calls `RemoteChangeHLEntry()` to send notification to that node, and `RemoveFromCache()` to clear out local record of the process' previous location. In either case, it calls `SendCacheInvalidate()` to invalidate object location information at the destination node. All of the cache and home node activities are meant to ensure that any request for the object's location will eventually be satisfied with the destination node, rather than the source node.

If this is the only process currently being migrated by this node, `move_phase()` then calls `send_ocb_from_q()`. This routine starts the process of moving the process, but will return long before the process completes. It will be called periodically, as acknowledgements of the migration are sent from the destination node. This first call primes the pump, as it were, getting the first part of the migration out so the rest of the protocol can be driven by the resulting acknowledgements. If there is already another process migrating, the completion of that migration will start up this migration. `move_phase()` then returns `PHASE_MOVED`, indicating success.

In the present implementation, once this point has been reached the system will successfully move the process to the destination node or die trying. (The latter is a definite possibility, in low memory situations, though the death is fairly graceful.) A future version of TWOS would have been able to back off migrations that could not be completed.

The TWOS migration protocol tries to move a process one virtual time at a time, from the earliest time to the latest. After setting up a new OCB at the destination node, the pre-interval state will be moved first, then any data items (input messages, state, output messages) for the next virtual time. Once all of those have arrived, the data items for the next virtual time will start to be sent.

Figure 13-1 shows a sample process to be moved. It consists of an OCB; a pre-interval state (for time 950); an input message, output message, and state for time 1100; and two input messages for time 1350. This process is to be moved from node 1 to node 2. This chapter will follow this sample migration throughout its course.



A[1000,+∞)

Figure 13-1: A Process To Be Migrated

send_ocb_from_q() is a rather simple function serving mostly as a driver for migration. It checks to see if the migration queue contains anything, and, if so, if the item at the front of the queue is ready to migrate something else or is still awaiting acknowledgement. If the item at the front of the queue is ready, its migration status is set to MIGRSTART and send_phase() is called.

send_phase() calls sysbuf() to allocate a buffer for a system message. It fills that buffer with basic information about the process to be migrated, including its identity, how many items are to be moved, and its type. Also, the statistics fields attached to an OCB (see Chapter 22) are copied into the message buffer. (The fact that all statistics must be sent in this buffer limits the size that the OCB statistics field can grow to, depending on message buffer size.) Once all of the information is ready, send_phase() calls sysmsg() to send a MOVEPHASE message, and sets the migration status of the OCB in the migration queue to WAITFORACK. send_phase() writes a line to the migration log and returns.

When the MOVEPHASE message is received by the destination node, it calls recv_phase(). recv_phase() must prepare the destination node to receive all of the various pieces of the migrating process. First, recv_phase() calls mkocb() to set up an OCB for the new process. It extracts the information about the phase from the message and stores it in the new OCB. It calls nqocb() to put the new OCB in the scheduler queue, where its run status is listed as BLKINF, meaning it won't yet be scheduled. Its migration status is set to MIGRSTART. RemoveFromCache() is called to clear out any stale object location information that may be here and GetLocation() is called to immediately set up a new cache entry (see Chapter 17 for more details). Also, if this is the process' home node, the local pending list is checked for location requests that slipped in between the invalidation of the previous entry and the arrival of the migration system message. (Again, Chapter 17 has more details.) Assuming that the process to be migrated did not consist solely of an OCB, recv_phase() then calls send_phase_ack() to tell the sender to start shipping the first virtual time. If the OCB itself was all there was to be moved, recv_phase() changes the local process' migration status to MIGRDONE and calls send_phase_done() to tell the sender that the migration is complete.

The migration module contains a number of routines like send_phase_ack() and send_phase_done(). These routines typically allocate a system buffer and send off the appropriate kind of message, usually with relatively little contents.

In our example, the migrating process now has an OCB on the destination node, as shown in Figure 13-2. As yet, no messages or states have migrated. The shading of the OCB on node 1 indicates that the OCB is in the migration queue on that node, and is not eligible for scheduling there. The OCB on

node 2 is in the scheduler queue on its node, and could run events, but it has no events to run, or states to use as input for them.

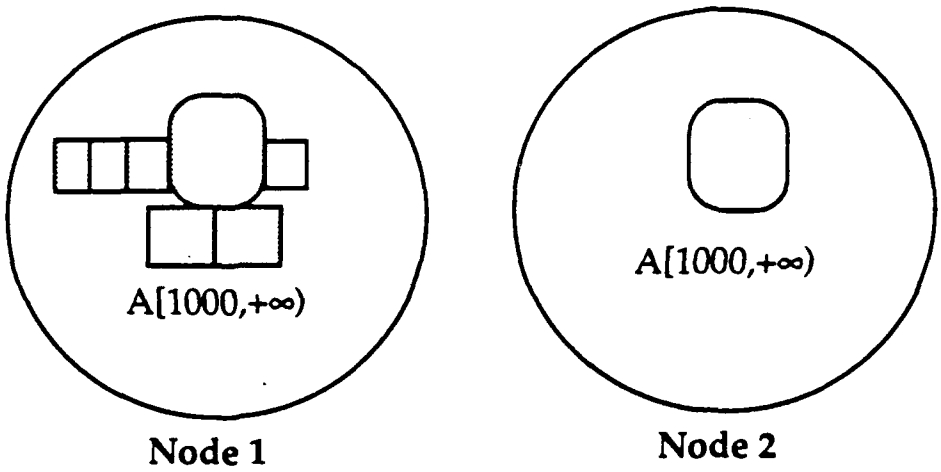


Figure 13-2: OCB Set Up On New Node

When the sending node receives the process acknowledgement, it calls `recv_phase_ack()`, which finds the migrating process in the migration queue, sets its migration status to `OCBSETUP`, and clears the `WAITFORACK` status. Then it calls `send_vtime()` to ship off the first virtual time. `send_vtime()` will be called once for each virtual time that is to be shipped.

`send_vtime()` allocates a system message buffer and fills it with all necessary information for shipping out this virtual time. It must figure out what virtual time is to be moved, and also what virtual time will be moved after this one. (If there is only one virtual time left to move, the next virtual time to be moved is set to the end of the phase.) `send_vtime()` must also count the number of states (zero or one), the number of input messages, and the number of output messages that are to be sent. Once all this information is gathered, it is copied into the message buffer and sent off as a `MOVEVTIME` message. The phase limit field of the local OCB for the migrating process is set to the virtual time being moved, the migration status is set to `SENDVTIME`, and the waiting status is set to `WAITFORACK`. A special check is made to determine if we are shipping out the minimum virtual time sent from the node (for GVT purposes, see Chapter 7). Then this node awaits an acknowledgement.

In the example of Figure 13-1, the first virtual time to send is 950, and the next is 1100. At 950, there is a single state to be sent.

When the receiver gets this `MOVEVTIME` message, it calls `recv_vtime()`. It finds the receiving OCB using `GetLocation()`, makes a few error tests, and copies the necessary information into fields in the OCB. It sets the OCB's phase limit field to the time that is moving (950, in the example), and sets the

OCB's next phase limit field to the next time (1100, in the example). The TWOS scheduler will not schedule any events for this process at times higher than its phase limit. Since its phase begin is 1100, and its phase limit is 950, there are currently no events this process can perform. The counts of messages and states are copied into the OCB, as well. The migration status of the OCB is set to `RECVVTIME`, and `send_vtime_ack()` is called.

When the sender gets this ack, it can start shipping out all data associated with this virtual time, since it knows the receiver is ready to get it. In `recv_vtime_ack()`, the sending node sets the migrating process' migration status to `SENDINGVTIME` and its waiting status to `WAITFORDONE`. Then it starts shipping out information as fast as possible, without waiting for further acknowledgements. It sends out the output messages first. Then it looks in the local `STDOUT` object's queue for messages from this object, and sends them. (See Chapter 20 for more details on the behavior of the `STDOUT` object.) The input messages are sent next. All of these messages are marked `MOVING`, indicating that they are part of a migration and receive special queueing and GVT treatment.

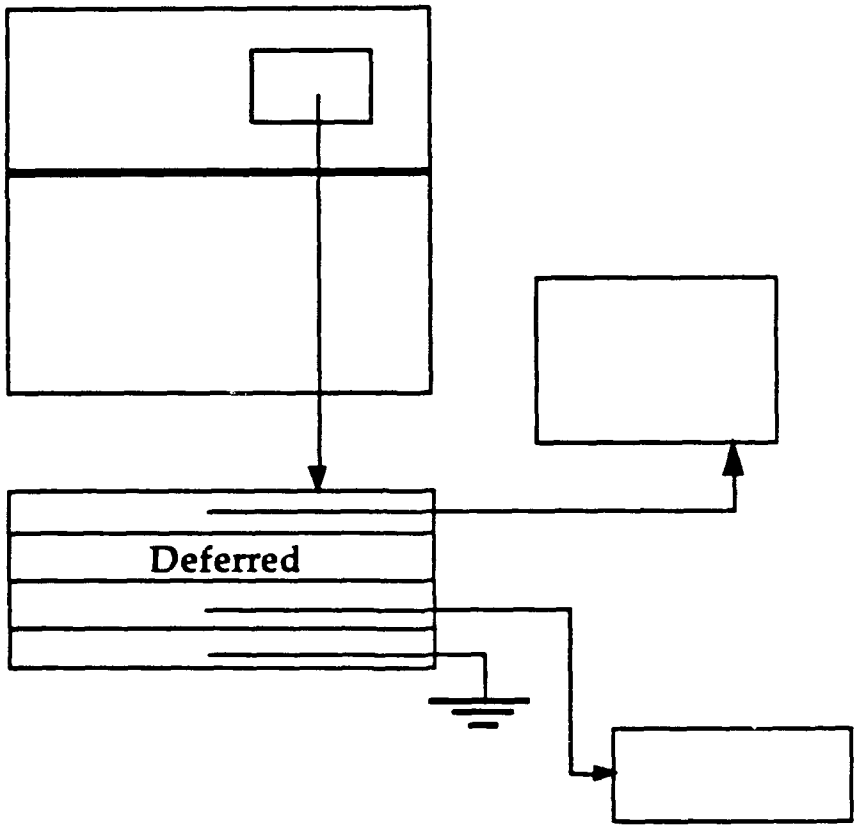


Figure 13-3: A State To Be Moved

Once the messages have been sent (but not necessarily received), if there is a state to be sent, `recv_vtime_ack()` calls `send_state()`. The state is marked

as migrating. If it is a pre-interval state, as it is in the example, it is also marked as such. Figure 13-3 shows the state that is to be moved. It consists of its header, the normal data part of a state, a dynamic memory address table, and three allocated dynamic memory segments, one of which is deferred. (This state would not actually be migrated if it were a pre-interval state, as such states never have deferred memory segments. However, this chapter will only cover the details of shipping a single state, so the example discussed will be as general as possible. See Chapter 9 for more details concerning deferred memory segments.)

`send_state()` allocates a state migration header and copies all relevant information about the state into it. `send_state()` must then determine how many message packets will be needed to ship this state to the receiver. Unlike messages, states can be bigger than a message buffer. In addition, they can have dynamic memory segments attached, making them, effectively, even larger. TWOS must ship the state out in pieces and reassemble it on the other end, leading to further complications.

States are sent to other nodes by putting them in a special queue. This queue can contain states from migrating processes and states sent from one phase to the next to cause rollback (see Chapter 12). `send_state()` must now put this migrating state into that queue.Rollback:Between Phases

`send_state()` is a general routine called both for migration purposes and for *purpose of shipping a new pre-interval state to a phase to force it to roll back.* There is no point in sending more than one pre-interval state to a given phase. Only the most recently shipped pre-interval state is needed; any others will simply be discarded, anyway. Therefore, at this point `send_state()` has an optimization that looks through the queue of moving states to see if there is another pre-interval state already being sent to this phase. If there is, that state is removed, if it can be safely removed. It cannot be safely removed if it is in the process of being transmitted, or if it is not a pre-interval state, or if it is migrating (as that would mess up the count of states to be migrated), or if it's not going to the same destination.

Then `send_state()` inserts the migrating state into the end of the moving states queue and calls `send_state_from_q()`. This routine looks at the state at the head of the state migration queue and determines if it is ready to be shipped. This state may already be in the process of moving, and may be waiting for acknowledgements, in which case `send_state_from_q()` returns immediately. If the state is ready to be moved, then `send_state_from_q()` must determine if it is attached to a migrating process that hasn't moved yet. (This situation could arise if the state at the head of the queue is a non-migrating pre-interval state.) If the destination phase isn't ready to receive it, this state must not be shipped. The state

migration queue is reordered to move something that can be shipped to the front, if any such state exists.

If the state at the head of the state queue can be moved, `send_state_from_q()` creates a system message. `send_state_from_q()` will be called more than once to ship a single state, so the following code is somewhat general. `send_state_from_q()` must check to see if this state has already started moving, by looking at the state migration header's `segno` field. State are shipped in segments. There is one segment for the state's header and body, plus one for each piece of dynamic memory attached to the state. A deferred piece of dynamic memory counts as a segment. There is no extra segment for the dynamic memory address table. In the example state shown in Figure 13-3, there are a total of four segments to be shipped, one for the state's header and body, and one for each of the three allocated dynamic memory segments.

In addition, each segment may be large enough to require multiple message packets for transport. Each packet can carry around 500 bytes of data, so only very modestly sized state segments can be transmitted in a single packet. Each of the segments may consist of multiple packets.

If the `segno` field is set to zero when `send_state_from_q()` reaches this point, then no segments have yet been sent. If, in addition, the `pktno` field is 0, the destination process does not yet have any information about the state it is to receive, other than possibly the fact that it is coming. (If this is not a migrating state, but a non-migrating pre-interval state, the destination process does not even expect the state.) In this case, the destination process must be informed of the incoming state and its contents before anything further can be done.

TWOS generally cannot count on ordered delivery of messages, unless a higher level protocol assures it, so `send_state_from_q()` must ensure that the receiver is ready to accept the pieces of the state in any order. Thus, when `send_state_from_q()` starts to ship off a state, it must be certain that the first packet of information sent contains everything that the destination process will need to know to reassemble the state. That information consists of the total number of segments to be sent and the total number of packets to be sent. Those pieces of information are stored in the state header, in fields called no_segs and no_pkts, so all that `send_state_from_q()` has to do to transmit that information is make sure that the first packet sent contains the state header. To make sure that nothing further happens until the destination process is ready, `send_state_from_q()` sets the migrating state's `waiting_for_ack` flag to indicate that no more packets can be sent until the receiver indicates that it is ready for them. The state's migration header is set to indicate that the state has started migrating.

send_state_from_q() decides what part of a state segment to send in each message by maintaining a count of the number of packets sent for the segment being transmitted. It uses this count to calculate an offset into the segment. Bytes are copied into the allocated message buffer from this offset. make_static_message() is called to set this message buffer up, then its fields are filled in to help the destination process identify what segment and packet it is carrying. It is sent out as a system message of type STATEMSG, using the sndmsg() call. Then send_state_from_q() determines if all packets of this segment have been sent. If they have, send_state_from_q() moves on to the next segment, checking at this point to see if all segments have been sent. If they have, the state's waiting_for_done flag is set, indicating that it expects a completion signal from the destination process. send_state_from_q() then exits without taking further action, having sent out one packet of the state.

send_state_from_q() will be called again when the destination process sends an acknowledgement of the receipt of the packet just sent. However, send_state_from_q() can be called from the main loop, as well, when TWOS determines that working on a migration is the most important thing for the node to do. (See Chapter 21 for discussion of the TWOS main loop.) The main loop may call send_state_from_q() before the destination process has fully handled the last packet, or even before that packet has progressed off node. Unless the outstanding packet is the first packet of the first segment, send_state_from_q() will not wait for an acknowledgement to send the next packet. Whenever called, it will just go ahead and send another packet. The only exception is for the first packet, since the destination process will not be prepared for the other packets until it has received the first one. When the entire state has been sent, send_state_from_q() will not move on to the next state until it receives word that all of the packets have been received.

One special case that send_state_from_q() must deal with is deferred memory segments. Such segments have no actual data associated with them. All that must be sent to the destination process is an indication that there is a deferred segment and its position in the dynamic memory table.

In the example shown in Figure 13-3, assume that the state header and body requires two packets, the first dynamic memory segment also requires two packets, and the third dynamic memory segment uses a single packet. In this case, send_state_from_q() would first send out the first packet of the state header and body. The migration header for this state would be marked as waiting for an acknowledgement. When the destination process sends the acknowledgement, send_state_from_q() will ship out the second packet of the first segment. Then, either when that packet is acknowledged or the system otherwise decides to deal with moving states, send_state_from_q()

will send out the first packet of the second segment, which is the first dynamic memory segment. Next, the second packet of that segment will be sent. Then, when `send_state_from_q()` is called again, a special packet containing the information necessary to set up the deferred segment will be sent. Finally, the single packet of the final segment will be sent. The sending node will then wait for a done signal from the destination.

Each time that an acknowledgement of one of these packets is received, `recv_state_ack()` is called on the sending node. `recv_state_ack()` makes sure that the right migration is being acknowledged, then simply calls `send_state_from_q()` again. Eventually, the receiver will send a done signal. Once that signal is received, `recv_state_done()` will remove this state from the state migration queue. A certain amount of care is necessary when doing so, as moving a state can have GVT consequences. Its arrival at the destination phase can cause a rollback to the beginning of that phase, and the only record that the rollback will occur is the shipment of this state. If the state is part of a migration in progress, other mechanisms ensure that GVT considers the consequences, but if the state is a pre-interval state sent due to rollback, `recv_state_done()` must compare its time to the minimum message receive time yet seen. If the state's time is lower, that time must be saved for the next GVT calculation. In any case, when the state is removed from the migration queue, it is destroyed and returned to the memory pool. Then `send_ocb_from_q()` is called, to determine if the completion of this state send will permit the sending node to ship a migrating phase's next virtual time.

We now turn to how a node receives a state. The basic routine is `recv_state()`. `recv_state()` is called for each packet of the state that comes in, so it must be able to set up the state data structure, fill in the empty parts as new packets arrive, and recognize when all of the state has arrived. `recv_state()` first calls `GetLocation()` to find the phase whose state is arriving. After error checks to deal with problems when the expected phase isn't there, the segment number of the incoming state packet is checked. If the segment number is zero, then the packet is part of the state header and body; otherwise, `recv_state()` finds the appropriate entry in the state's dynamic memory address table. The state header and body might not yet exist, if the segment number is zero, in which case a null pointer is assigned to the state pointer, instead of a pointer to the state. If the incoming packet does not belong to not segment zero, then at least the first packet of segment zero has already arrived. (The sender waits for an acknowledgement of that packet before sending any others.) The arrival of that very first packet should have set up both the state and its associated address table. Paranoid code checks to see if they have been set up, but that code is not normally compiled in.

Regardless of what segment is being dealt with, this might be the first of its packets to arrive. In that case, no memory has yet been allocated for that segment. All packets of a segment contain the overall length of the segment, so that information can now be used to allocate the appropriate amount of memory. If the segment is a deferred dynamic memory segment, then it can be so marked in the dynamic memory address table. Otherwise, `m_create()` is called to allocate memory. An experimental piece of code that is not compiled in tried to do some emergency operations if this allocation failed, but that code is not yet properly debugged. So, if the allocation fails, `recv_state()` will back off the entire receipt of the state by deallocating any memory already used for the state (using `destroy_state()`) and calling `send_state_nak()`.

If the sender gets a state nak, it will immediately restore the status of the migrating state to indicate that no work has been done to move it, since the receiver just undid any such work. `recv_state_nak()` now calls `send_state_from_q()` to send out the state at the front of the queue. The state at the front of the queue will usually be the one just rejected by the receiver. There is no necessary reason to believe that the receiver will be able to accept it, this time, but TWOS has no other choice but to try resending it, since TWOS cannot, as yet, back off of a process migration. Eventually, when GVT calculation and commitment rolls around (or when rollback at the destination node frees space), the receiver may have enough room for the state. If the receiver never frees enough space, TWOS eventually notices the deadlock situation and fails, complaining of insufficient memory. Note that, in this case, the lack of memory can be directly caused by the non-deterministic choices made by dynamic load management, so one run of the simulation may succeed, while another on the same number of nodes fails.

More frequently, and happily, there is enough room to allocate the required segment's space in the `m_create()` call in `recv_state()`. In this case, `recv_state()` checks to see if the allocation was for segment zero. If it was, then a dynamic memory address table may need to be allocated, as well. `recv_state()` checks to see if one is needed, and, if it is, calls `m_create()` again to allocate it. A failure here is just as bad as a failure to allocate the state header and body, and causes the header and body to be deallocated and a nak sent. If the allocation succeeded, the dynamic memory address table is cleared. Then the OCB's `rstate` field is filled with a pointer to the new state under construction. Until the state has been completely received, it will be accessed through this `rstate` pointer, only. It is not linked into the state queue until transmission is complete. If this is not segment zero being constructed, the newly allocated segment's pointer is copied into the appropriate place in the state's dynamic memory address table.

If the segment is not deferred, the data carried in the packet is copied into the segment. At this point, the packet under consideration may or may not have been the first packet received for this segment. If it was the first, `recv_state()` just allocated room for the segment. If not, `recv_state()` obtained a pointer to the earlier allocated memory for the segment. In either case, the segment's memory pointer plus an offset is used to determine where to start copying, and the packet length determines the extent of copying.

`recv_state()` must now determine whether it has reached any milestones in this state transmission. First, it checks to see if any more packets are expected. If so, it calls `send_state_ack()` to signal the sender that it is ready for the next one. It decrements the number of packets it expects and checks again to see if it has all of them. If it does, the OCB's `rstate` pointer is nulled, to indicate that there is no state under construction. `recv_state()` calls `send_state_done()` to signal complete reception of the state.

If the state has completely arrived, `recv_state()` must determine what to do with it. There are more complications here than might be expected. This state might be migrating, or it might be a pre-interval state sent due to rollback. In either case, other state receptions and incoming event messages may complicate handling of this state. `recv_state()` thus checks to see if this was a `MOVING` state, indicating that it was sent as part of a migration. If the phase has set its SVT to a time earlier than this state's, a rollback has occurred at the receiving end and this state is no longer needed, so it is discarded with `destroy_state()`. Otherwise, `recv_state()` calls `put_state_in_sq()` to deal with the problem.

`put_state_in_sq()` checks to see if this is a pre-interval state. If it is, and this phase already has a pre-interval state in its state queue, that state is removed and destroyed. The sending node will never have more than one pre-interval state in transmission at a time, and takes care to ensure that pre-interval states are shipped in the order of their generation, so the incoming pre-interval state is guaranteed to be a more up-to-date version of the pre-interval state than the queued one. Some care must be taken here to ensure that the receiving phase maintains its proper object type. (See Chapter 14 for details of object types, and their relations to states.) `put_state_in_sq()` then finds the proper position in the state queue for this state and puts it there.

`recv_state()` decrements the number of states it expects to receive as a result of migration. If it has received all the states, input messages, and output messages that are part of this virtual time, `recv_state()` calls `rollback_phase()`. `rollback_phase()` will usually call `rollback()`, at this point, setting the SVT of the receiving phase to the proper value. `rollback_phase()` then sets the phase limit for the receiving phase to be the next time to be transmitted. If all virtual times have been transmitted, it calls

`send_phase_done()`, to signal complete reception of the phase to the sending node. If there are still virtual times to be shipped, `rollback_phase()` calls `send_vtime_done()` to signal that the next virtual time can be shipped.

In the example of figure 13-1, when the pre-interval state at time 950 has been completely received, this is the path that will be taken. `recv_state()` calls `rollback_phase()`, which sets the phase limit at the destination phase to the next virtual time to be sent, which is 1100. Note that this phase's responsibility starts at virtual time 1000. During the course of migration, this phase may have received new event messages for times between 1000 and 1100. Figure 13-4 shows such a situation on the receiving end, with a message received for time 1030 and one for time 1070. Until the pre-interval state completely arrived, events at these times could not be run. As soon as that state arrives, however, the local node can schedule those events, if it wants. The next virtual time to be migrated, 1100, will not have any effect on those earlier events, so they can be safely run. If another message was received for time 1150, however, its event could not be run until all data for time 1100 was received.

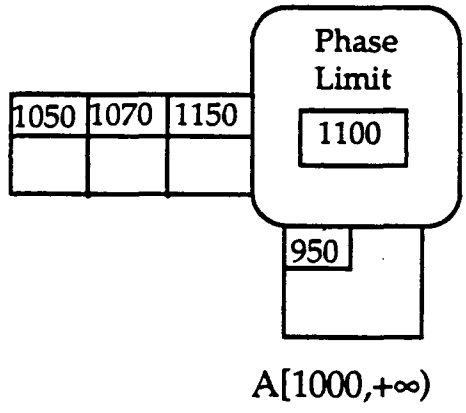


Figure 13-4: A Migrating Process Can Now Run Events

Of course, `recv_state()` must also be prepared to handle non-migrating pre-interval states. Once such a state has been totally received, `recv_state()` calls `put_state_in_sq()` to properly place it and then calls `rollback_state()`. `rollback_state()` is used to roll back a phase when it receives a new pre-interval state. It is not called for migrating pre-interval states, as such a state need not roll back its destination phase, since that phase knew about that state before migration started and had already taken care of any necessary rollback.

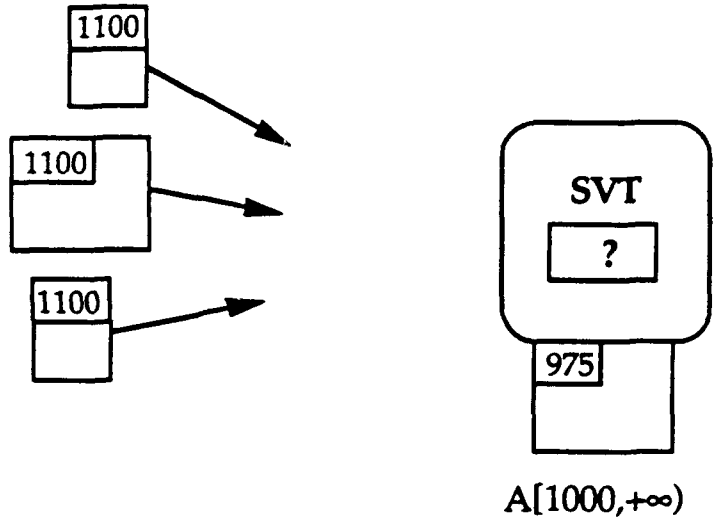


Figure 13-5: Pre-Interval State Rollback During Migration

`rollback_state()` was discussed in Chapter 12, but will also be covered here. If the receiving phase has any messages in its input queue, the arrival of the new pre-interval state should cause the phase to roll back to the receive time of the earliest such message. `rollback()` is used, for this case. If there are no such messages, then either a migration is in progress or it is not. If a migration is in progress, then migrating input messages may eventually arrive. The arrival of a migrating input message normally does not cause a phase to roll back, but it should, if a new pre-interval state has also arrived. Figure 13-5 shows the situation. A[1000, +∞) is migrating in, and has already received a pre-interval state for time 950. A new pre-interval state, for time 975, arrives. The input message, state, and output message for time 1100 are in transit, but have not yet arrived. The proper action for this phase is to roll back to time 1100, so that when the input message for that time arrives, its event will be re-executed with the proper input state (from time 975), rather than saving the results of the execution with the improper input state (from time 950).

To achieve this effect, `rollback_state()` calls `rollback()` with the phase limit as the rollback time. In Figure 13-5, for example, the phase limit is 1100, since that is the time migrating. A[1000,+∞)'s SVT will be set to time 1100. When the state for that time arrives, it will be discarded by `recv_state()`, as described earlier. The output message will be queued for lazy cancellation, and the input message will cause a new event for time 1100 to run, using the new pre-interval state.

The third possible option for `rollback_state()` occurs when the migration has completed and there are no local input messages. Chapter 12 covered that option thoroughly.

That completes the actions taken by `recv_state()`. Going back to the example from Figures 13-1 and 13-2, the pre-interval state for A[1000, +∞) has arrived and been queued, as shown in Figure 13-6. `recv_state()` called `rollback_phase()`, since the state was the only item at virtual time 950, and `rollback_phase()` called `send_vtime_done()` to inform the sending node that time 950 had been fully received.

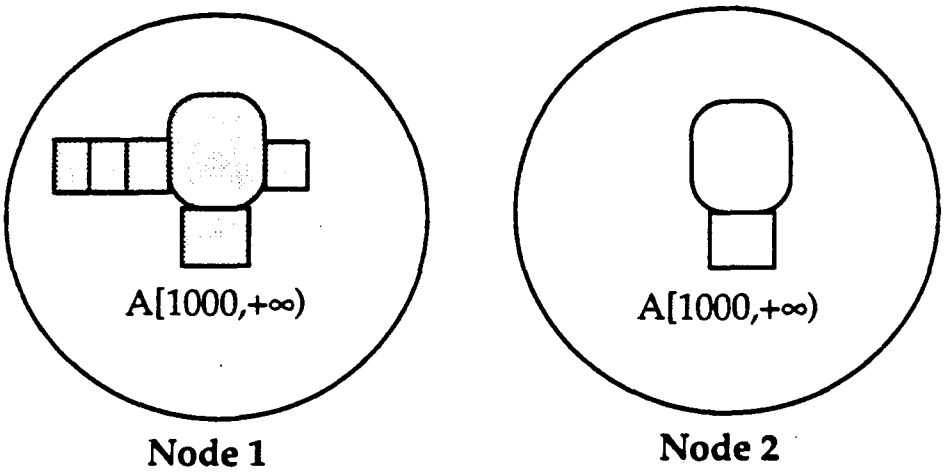


Figure 13-6: The Pre-Interval State Has Been Migrated

At this point in the example, node 1 has received a virtual-time-done message and called `recv_vtime_done()`. `recv_vtime_done()` finds the migrating process in the migration queue, sets its status to `SENDNEXTVTIME`, and clears its waiting status. Then it calls `send_vtime()` again. `send_vtime()` acts as before, making a system message to send to node 2 containing the inventory for the next virtual time to be shipped. Referring back to Figure 13-1, that virtual time is 1100, and there is a state, an input message, and an output message to be sent for that time. When the system message containing this inventory arrives, node 2 will again run `recv_vtime()`, storing the inventory and returning a virtual time acknowledgement. Upon receipt of the acknowledgement, node 1 calls `recv_vtime_ack()` and sends off the output message, the input message, and the state, in that order. Transmission of these items is as before, with the additional point that the messages are marked as migrating.

When a migrating messages arrives, `nq_input_message()` or `nq_output_message()` (depending on which queue it is destined for) will give it special treatment. Such messages never cause rollback, and, in the case of an output message, should be enqueued even if they are negative and reversed. (Normally negative reverse messages for the output come from extraordinary message sendback cases, as described in Chapter 10, and are returned to the sender in the expectation that they will annihilate.) If migrating messages annihilate, they are treated normally, causing rollback.

The enqueueing routines must check to see if all pieces of the virtual time inventory have arrived, just as `recv_state()` did, since the state may beat the messages to the destination, or, in many cases, there may be no state at all. Whatever piece arrives last, eventually `rollback_phase()` is called, resulting in a virtual time acknowledgement being sent to node 1 and the phase limit of the process on node 2 being raised to the next virtual time to be shipped (to 1350, in the example).

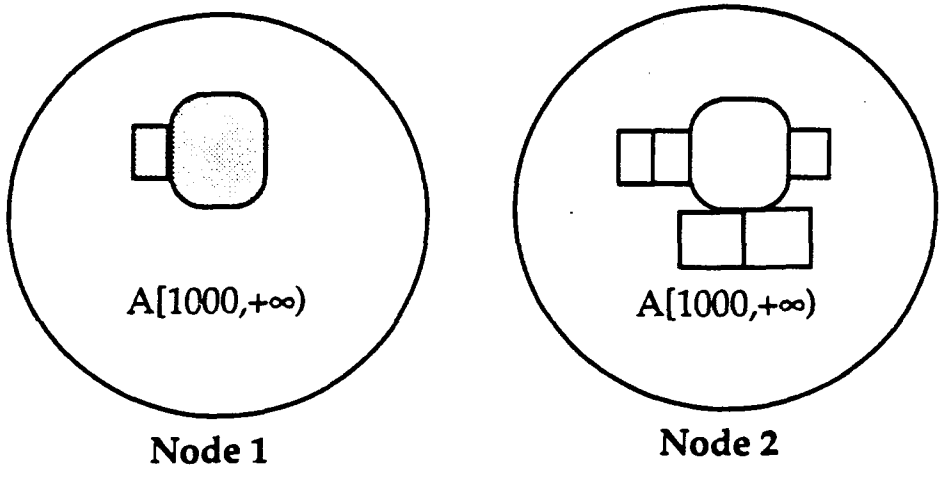


Figure 13-7: Time 1100 Has Been Migrated

Figure 13-7 shows the situation, now. Only one virtual time remains to be shipped, and it consists of two input messages at time 1350. The procedure for shipping them is as above, and the result is seen in Figure 13-8.

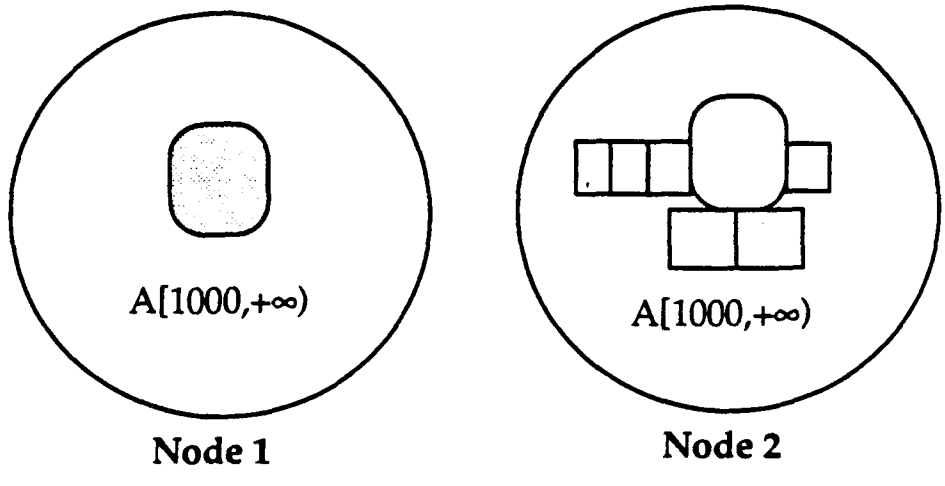


Figure 13-8: Time 1350 Has Been Migrated

Now, when `rollback_phase()` is called at node 2, instead of calling `send_vtime_done()`, it calls `send_phase_done()`. `rollback_phase()` can tell that all virtual times have been moved because the phase limit of the

migration has caught up to the phase end of the migrating process. When node 1 gets the phase done message, it removes the migrating process' OCB from its queue of migrating processes, calls `nukocb()` to dispose of it, and decrements its count of processes to migrate. If there are any more processes in its migration queue, it calls `send_ocb_from_q()` to start up the migration of the next process.

Chapter 14: Dynamic Creation and Destruction of Objects

TWOS is able to dynamically create and destroy objects, as discussed in Section 4.5 of the TWOS User's Manual. Dynamic object creation is accomplished in TWOS the `newObj()` system call. `newObj()` works by sending a special creation message to the object to be created. This message causes the object's data structures to be created, sets the object's type appropriately, and runs the object's initialization section. The dynamic create message contains a virtual receive time, which is the creation time for the object; a name for the new object; and an object type, which must be one of those defined by the user in the simulation source code. (TWOS does not allow dynamic creation of object types.) The user need not, and, indeed, cannot specify the node on which the object will be created. Instead, TWOS chooses a node based on load information available to the system.

This method of dynamic creation has an obvious complexity. The object is created by delivering a message, but there would not seem to be anywhere to deliver the message until the object is created. The problem is solved by the way TWOS handles messages sent to nonexistent objects. The need to handle messages for nonexistent objects arises not only from this problem, but also from assumptions TWOS must make about the order of message generation and delivery.

There are two time-related issues concerning message delivery and dynamic object creation. First, the virtual send and receive times of the dynamic creation message and some other message sent to the created object may cause complexity. At virtual time 100, process A can send an event message to object C to be received at virtual time 200, even though object C does not exist at virtual time 100. If process B creates object C at virtual time 150, process A's message will be normally delivered. If object C is not created by virtual time 200, then process A's event message is in error. In either case, TWOS must be able to save this event message until the system can determine whether object C will be created in time to handle it.

The second issue relates to real time ordering of events. Consider another example. Process X creates object Y at virtual time 3000. Process W sends an event message to object Y with a send time of 3500 and a receive time of 4000. In a normal sequential simulation, process X's event will definitely be processed before process W's, so object Y will be present to receive the event message sent at time 3500. In TWOS, however, optimistic execution on different nodes may cause process W to send the message to Y earlier in real time than process X creates Y. TWOS must be able to save this event message and eventually deliver it to Y. An even simpler version of the problem can arise, when, in a single event, one process creates a new object and sends it an event message for some virtual time after the virtual creation time. TWOS

does not guarantee ordered message delivery at the low level, so the event message may be delivered before the creation message.

TWOS deals with the problem of delivering messages to nonexistent objects by taking the philosophical position that there are no nonexistent objects. All nameable objects exist, in an abstract sense. Some of them have internal machine representations (as one or more processes), some are only abstract. Any object for which the user requested creation must, of course, have a physical representation. Additionally, any object that the user sent a message to must have a physical representation. Whenever a message must be delivered to an object that has no process representing it, TWOS creates such a process.

A process created for this reason is called a *null process*. The major activity of null processes is queueing incoming messages. Null processes do schedule events, largely to make their internal handling similar to normal processes', but these events do no work, and are called *null events*. Null processes essentially loiter around waiting for a dynamic creation message to arrive. When one does, the null process is converted into an object of the type specified in the creation message. Its initialization section is run and the object becomes ready to handle normal event messages. If any are already queued, they may have run null events before the create message arrived, but the creation will roll the null events back and re-execute them after the object has been properly created.

In some cases, a null process may be created by another process running optimistically down a path that will be later rolled back. Due to premature use of data, a process can create a garbage process name as the receiver of a message. TWOS has no way of distinguishing such a message from a legitimate message to an object not yet created, so the message must be delivered. A null object with the garbage name is created and the message is stored in its input queue. Eventually, the sending object will roll back and the message will be cancelled. TWOS could then free the memory used by the null object.

Bugs in a simulation can cause committed messages to be sent to null objects. For instance, the user may incorrectly copy the name of a destination object for a message into the send request. In such cases, the misnamed object will be created and the message delivered. The commitment protocol will eventually notice that a committed event message has been sent to a null object. TWOS will then flag the error and halt the simulation so that the user can try to find the bug.

The most normal case, however, is that null objects are fleetingly created by the arrival of dynamic creation messages. On arrival, the dynamic creation message is put in the input queue of the new null process. The null object will be scheduled once the virtual time of the creation message becomes the

earliest unprocessed time for any process on the local node. Once scheduled, the null process will change its type to the type specified in the creation message and run its initialization section. Then it may go on to run normal events, like any other TWOS process.

Dynamic destruction of objects has some of the same complexities. Destruction is performed by sending an object a dynamic destruction message. Like any other user-generated TWOS message, this message is subject to cancellation until it is committed. Thus, TWOS must not take any irreversible action based on this destruction message until it is sure that the message will not be cancelled. Before the commit point is reached, the effect of a dynamic destruction message is to change the type of the process receiving it back to null. Any event messages received for times later than the destruction generate null events, just as if the object had never been created. If the destruction is committed, and there are no later input messages queued for the destroyed object, much of the storage it occupies could be freed. Only enough storage to hold statistics for the destroyed object need be kept.

TWOS permits an object to be dynamically created at a given virtual time, to run an event at that virtual time, and to be dynamically destroyed at that virtual time. To support this feature, TWOS must always ensure that dynamic creation messages for a given object at a given virtual time are queued before either event messages or dynamic destruction messages for the same time and object. In addition, the event handling code must be aware that running a dynamic creation at a given virtual time does not give the initialization section of the code access to event messages for that time. Instead, those messages are presented to the event section of the object, as a new event, once initialization is complete. Similar statements apply to the relationship between event messages and dynamic destruction messages.

TWOS permits multiple dynamic creations of an object, provided they are all of the same type. (Additionally, it is legal to create an object as one type, dynamically destroy it, then create it as another type.) Only the first creation of a given type will cause that type's initialization section to be run for the object.

The remainder of this chapter will discuss details of how dynamic creation and destruction of objects works in TWOS.

`newObj()` behaves much like `schedule()`, which was discussed in Chapter 3. `newObj()` makes simple error checks, performs some timing routines, then uses `switch_back()` to call `sv_create()`. `sv_create()` allocates and clears a message buffer, then sets it up as a `DYNCRMSG`. It calls `dispatch()` to deliver the message. `dispatch()` and subsequent routines treat the `DYNCRMSG` message just like an event message, giving it similar priority and handling. Only when the scheduler is about to start an event for it does the path

significantly diverge from that of an event message, so the rest of that path will not be discussed here.

`objhead()` is where the path of a dynamic creation message diverges from that of an event message. `objhead()` checks the type of message it is handling early on. If the message is a `DYNCRMSG`, `objhead()` calls `ChangeType()`, which extracts the character string type provided by the user from the message and calls `find_object_type()` to get the local pointer into the type table.

The TWOS type table consists of one entry for each user defined type, plus a special entry for the null type and a special entry for the `STDOUT` type. Each entry has a pointer to its string name, pointers to the various entry points for the type, the size of the type's state, and some pointers used for type initialization and user library support. `find_object_type()` merely does a simple comparison of the string the user provided to the string pointed to by each type table entry and returns a pointer to the matching entry. `ChangeType()` compares that pointer to the existing type pointer and returns -1 if they match. Otherwise, the pointer to the type table entry is returned.

`objhead()` copies the new type pointer into the OCB's type pointer. If -1 is returned, it copies in the type from the previous state, instead, and saves a record that -1 was returned. `objhead()` goes on with its normal activities, but, instead of running the event section for this object, it runs the initialization section, unless -1 was returned by `ChangeType()`. In that case, the object has just received a create message of a duplicate type, which causes a dummy initialization section to be run. Running the dummy initialization section allows the duplicate create to follow the normal path of a creation without running the initialization section twice.

If an object receives multiple dynamic creation messages of the same type at the same virtual time, TWOS will not flag an error, and will run the initialization section of the object only for the first such creation message. However, TWOS will pass through `objhead()` once for each dynamic creation message.

TWOS takes no further special action for a dynamic creation message until commitment time. (For complete details on how commitment works, see Chapter 8.) When committing a message in a process' input queue, `objpast()` checks to see if the message is a create message. Both dynamic creation messages (`DYNCRMSG`) and static creation messages sent from the configuration file (`CMSG`) are treated the same, here. An OCB variable called `crcount` is incremented when a create message is seen by commitment. If this variable is above 0 when a creation is found, before incrementing it `objpast()` calls `find_object_type()` using the creation message's type. If the type matches the type in the OCB, then this is simply a duplicate create

message that should be ignored. `crcount` is not incremented for such messages, and they are fossil collected.

Commitment time is also when TWOS detects messages improperly sent by the user to objects that were never created. Whenever an event message is to be committed, if the `crcount` field for the process committing it is less than 1 then it was performed by a null object. TWOS prints an error message and traps to tester.

`crcount` was originally envisioned as a counter, but it has become more of a flag, in practice. It should always be either zero (if the object hasn't been created or was destroyed) or one (if the object has been created and not yet destroyed.) In the case of multiple legal creations for a single object, `crcount` will not be incremented above 1.

As mentioned earlier, TWOS must ensure that a representation of an object is already in place when any message, including a dynamic creation message, needs to be delivered. Ensuring the existence of a null object to receive a message, if no representation yet exists, is the responsibility of the phase location code, which is fully described in Chapter 17. To cover only the directly relevant parts of the phase location code, when a process' home node receives a request for the process' location, but the home node does not know anything about the object that the phase is part of, the home node creates the object with type null. This action is performed in a routine called `ServiceHLRequest()`. This routine calls `MakeObject()` to create a null object of the proper name. Ideally, `MakeObject()` would use load information (as described in Chapter 11) to determine where to put the object. In the existing implementation, `MakeObject()` always puts the object on its home node. Thus, any dynamically created object will always be located on its home node. This choice can lead to heavy overloading of nodes if dynamic creation is used frequently and the hashing algorithm for matching object names to home nodes gives a poor selection of nodes. This problem has never been detected, in practice.

Dynamic destruction in TWOS is a simpler process than dynamic creation. `delObj()` is the system call to delete an object. It follows a path similar to that of `newObj()`, calling `sv_destroy()` through `switch_back()`, which in turn formats a `DYNDSMSG`. In `objhead()`, a `DYNDSMSG` is handled by changing the object type to null and running a routine called `dummy_destroy()`, whose main purpose is compatibility with normal event handling. In `objpast()`, during fossil collection, a check is made to ensure that the destroy message was sent to an object that had been created, by looking at the process' `crcount` field. If that value is not zero or less, then `crcount` is decremented.

In principle, a committed destroyed process could be totally removed from the system. However, we need to save statistics associated with that process (see Chapter 22) to permit proper balancing of statistics after the run.

Attempts have been made to reclaim some of the memory used by such processes. None of these attempts have produced completely correct results, yet, so they are not included in the compiled code.

Chapter 15: Throttling Code

TWOS relies on optimistic execution for synchronization and speedup. There is evidence to suggest that over-optimism in Time Warp simulations can lead to poor performance. In response to this evidence, we have experimented with code to limit the optimism of TWOS, in the hopes of improving performance. This chapter does not discuss the results of those experiments, but only the code implemented for them. Throttling is not used by default, in TWOS, but can be turned on by the user from the configuration file, if desired. At the moment, none of the experimental throttling methods described here are regarded as sufficiently satisfactory to use by default.

TWOS can throttle optimism in six different ways, but only one method can be used during a single run. The method can be set from the configuration file. The six methods are

1). Output message throttling - Do not permit processes to execute if they produce too many output messages.

2). Event Time Throttling - Do not permit processes to execute if they have too much outstanding uncommitted event time, compared to the event time they have recently committed.

3). Dynamic Window Throttling - Do not permit processes to execute if the virtual times of their events are more than a varying amount of virtual time beyond the last computed GVT, where the varying amount depends on the difference between the last two values of GVT.

4). Event Count Throttling - Do not permit processes to execute if they have more outstanding uncommitted events than some constant factor.

5). Rollback Throttling - Do not permit processes on a particular node to execute if that node has rolled back too much work compared to the amount of work it committed during the last load management interval. (Amount of work is measured in real time, not count of events, for this option.)

6). Static Window Throttling - Do not permit processes to execute if the virtual times of their events are more than a single set amount of virtual time beyond the last computed GVT.

In all cases, the decision of whether to throttle is taken in one place in the operating system, while code supporting the decision making process is somewhere else. This chapter will cover all code associated with throttling.

The decision of whether or not an event should be run is made in `load_obj()`. (See Chapter 4 for other details of `load_obj()`.) If throttling has been enabled for this run, the type of throttling chosen is checked. For each type, some simple comparison is made, either looking at variables already set up, or calling a routine to perform some calculations and checking the return value. If the option in force determines that an event is not to be run, `blockObject` is set to 1. If the blocked event is set to run at GVT, then this decision is ignored, as events running at GVT are always correct and should always be run. Otherwise, the process is marked as blocked and `load_obj()` returns to `dispatch()`, which checks the next process in the scheduler queue to see if it can run its event.

For output message throttling, `load_obj()` calls `countOutputMsgs()`, which simply counts how many messages are stored in the process' output queue. If the number is lower than a value set from the configuration file, then the event can execute. Otherwise, it cannot. This method has no real chance of working well unless aggressive cancellation is used, instead of lazy, as a process that has run far ahead and generated many output messages before rolling back will rarely get to run, even if it is behaving properly now. If aggressive cancellation is used, however, then information about how much this process has annoyed its neighbors is lost and cannot be used for throttling purposes.

For event time throttling, the process' OCB is checked to see if the amount of time it is permitted to spend executing events is less than or equal to zero. If it is, then that process cannot run an event. This field is reset in `objpast()`, after all fossil collection has been done. `objpast()` keeps track of how much event time was committed during this fossil collection cycle for this process. It then counts up how much time all uncommitted events for that process take up. The amount of time committed is multiplied by a factor set from the configuration file, and another settable factor is added. Then the amount of uncommitted time is subtracted. The result (which could be positive, negative, or zero) is how much time the process is permitted to spend running events before the next fossil collection cycle starts.

This value is adjusted during the cycle. Whenever an event finishes executing, the amount of time it spent is subtracted from the allocation. On the other hand, whenever an event rolls back, its amount of time is added to the allocation.

Dynamic window throttling compares the simulation time field of the virtual time of the event to be run to a dynamic window simulation time. If the event's time is less than the window time, the event is not throttled. The dynamic window time is readjusted during each GVT cycle, in `gvtupdate()`. `gvtupdate()`, once it has determined the new GVT, subtracts the simulation time of the previous GVT from the simulation time of the new GVT. If the

difference is not zero, the difference is multiplied by a factor set from the configuration file and added into the new GVT simulation time. The result is the latest time at which an event can execute before the next GVT computation completes. If no GVT progress has been made, the window is left at the same value as it had in the last cycle.

Only the simulation time field of virtual times is used by the dynamic windowing method. Applying the method to all three fields of virtual time is not really practical, as not all applications use the extra fields, and not all of those using the fields use them in the same way. Thus, the utility of this method is somewhat limited.

Event throttling counts the number of uncommitted events for a process and compares the count to a constant factor. If the count is higher than the factor, the request[ed] event cannot be run. `objpast()` calls `eventsBeyondGVT()` when this thr[ot]tling method is in use to count the number of events for the process. A more sophisticated dynamic version of this method was tried earlier, similar to the event time throttling method, but with counts of events instead of totals of event times. Some of that code still exists in the system.

Rollback throttling always blocks all or no events on a node during a given load management cycle. (Except, of course, that events at GVT always run.) After calculating the node's utilization in `loadStart()` (see Chapter 11 for details on this routine), that utilization is compared to the proportion of time spent on work that was rolled back. If more time was spent on work rolled back than on work committed, the blocking flag is set true. This option, as currently implemented, is rather stupid, as it will block events far too often. There should be some factor multiplied by the good utilization before comparison. Adding this factor would be simple, but has not yet been done because this method showed little promise and inspired little confidence.

Static window throttling is similar to dynamic window throttling, but instead of recalculating a window every GVT cycle, a constant window value is added to GVT on each comparison and compared to the time of the event. (An obvious optimization would be to make the addition once at GVT computation time.) Like dynamic window throttling, static window throttling only uses the simulation time field of virtual time, limiting its utility for certain applications. It would be easier to extend static windowing to use all three fields than it would be to extend dynamic windowing. This method can only work if users are willing and able to determine a good static window values themselves for each of their applications.

Chapter 16: Critical Path Computation

TWOS is able to compute the critical path of a simulation on the fly, as the simulation is running. The critical path computation cannot be completed until the simulation ends, so no critical path information is printed out until then. Once the simulation ends, TWOS computes the critical path and writes it into a file called CRITPATH. The TWOS User's Manual, section 7.5, describes how to request the critical path and use the resulting data.

Critical path computation has a rather high overhead, especially in memory usage, so TWOS does not compute the critical path by default. Even when the critical path computation facility is not in use, however, TWOS will print out an estimate of the length of the critical path. This estimate will only be correct, though, if dynamic load management was turned off and aggressive cancellation was used instead of lazy cancellation, due to inability of the critical path mechanism to function properly in the face of these features. Any other form of phase decomposition (see Chapter 12) can also adversely affect the correctness of this estimate.

When requested, the critical path will be produced, generally at the cost of a substantially slower run. (The slowdown occurs only in overhead, not in user code, so the length of the critical path itself is not affected by this slowdown.) In the existing system, TWOS does not have enough memory to compute the critical path for many simulations, in which case TWOS will run out of memory and halt. As mentioned above, the critical path cannot yet be correctly calculated if lazy cancellation or dynamic load management is used, so TWOS automatically turns off those features when the critical path computation feature is turned on.

The critical path computation facility relies on the concept of *EPT* (event processing time), as developed by Tapas Som [Wieland 92]. This chapter will not attempt to explain that concept, beyond stating that the EPT of an event is the earliest possible time at which that event could have been correctly processed, ignoring supercritical effects [Reiher 91]. TWOS can calculate EPTs for all events and tag all messages with EPTs. At the end of the run, the event with the largest EPT is the last event in the critical path. By saving information about what events caused other events, TWOS can then move backwards along the critical path, identifying each critical path event along the way. Certain properties of EPTs permit some pruning of the event set before the computation completes.

The internals support for critical path computation falls roughly into three sections. First, code is scattered throughout the kernel to support the computation of each event's EPT. Second, there is code in the commitment mechanism for determining which information need not be saved for critical path computation. Third, there is code run at the end of the simulation to

generate the critical path. This final section of the critical path code is by far the largest portion of the code.

The code scattered throughout the kernel is largely intended to compute EPT. In order to properly compute an event's EPT, some timings must be taken. In `loadstatebuffer()`, the EPT for an event is initialized to the EPT of the last event for the process. The one exception is for creation messages, for which there is no previous state. In this case, the EPT is initialized to the EPT of the creation message. In `message_vector()`, the EPT contributions of the message or messages causing the event are compared to the state EPT, and the largest one is selected as the initial EPT for the event.

Most of the code for calculating EPT consists of a few lines duplicated in many places. Any place in which control passes from the user event to the operating system has a number of lines of timing code, some of which are for EPT computation purposes. `schedule()` is one such place, and as good an example as any. After making a system call to read the clock (which is hardware dependent), a variable called `object_end_time` is set to the current clock value. `object_start_time` was set to the time when the event last resumed execution, so subtracting `object_start_time` from `object_end_time` gives the length of time this event has been executing since it started or last returned from an interruption. That time can be added into the current estimate of the event's EPT, which is stored in its state. (`object_start_time` is reset every time the event is about to restart execution; the resetting is done in the main loop of TWOS.)

The second part of the critical path code is associated with commitment. (See Chapter 8 for a full discussion of commitment.) Not all states and messages can be fossil collected when GVT passes their characteristic virtual time, if critical path computation is turned on. TWOS uses those states and messages to trace the critical path backwards at the end of the run. However, some states and messages can be pruned before the end of the run, for reasons described in [Wieland 92]. In addition, the critical path computation facility does not really require the full state and message in order to perform its work, so some of the memory used up by items saved solely for critical path computation can be freed.

In `objpast()`, where normally a state would be destroyed when committed, the critical path code instead calls `truncateState()`. `truncateState()` releases all dynamic memory segments associated with the state, and its address table. Then `truncateState()` creates a special truncated state data structure and copies all information needed for critical path computation into that data structure. `nqTruncateState()` is called to put the special truncated state data structure into the process' queue of such truncated states, then the state itself is removed from the state queue and destroyed.

Generally, only one of the events that caused the event whose state was truncated could possibly have caused this event to be on the critical path (if it is). Only the event with the highest input EPT can possibly participate with this event on the critical path. (For an explanation of why, see [Wieland 92].) Therefore, `truncateState()` next calls `informNonCritPredecessors()` to tell all the other predecessor events that they did not cause this event to be on the critical path.

`informNonCritPredecessors()` checks the EPT of the previous state and the EPT of all input messages that caused this event. The previous state will have been truncated already, so `informNonCritPredecessors()` looks in the truncated state queue for it. Its EPT is then checked against the EPT of all input messages in the bundle for this event. (A bundle of messages is a set of event messages for the same process at the same virtual time that will collectively cause a single event at that process.) If any of these messages has a larger EPT than the previous truncated state, then that message is chosen, rather than the state.

After all messages have been examined, `informNonCritPredecessors()` sends out system messages to the processes owning all the causing events except the chosen item. If the previous state does not make the maximum EPT contribution, its state field called `resultingEvents` is decremented. This field is set to one for all states, and incremented once for each message sent by the event creating the state. If this decrementation brings this field down to zero, then this state is definitely not on the critical path, and `nonCritEvent()` is called for it.

`nonCritEvent()` finds its own previous state and decrements its `resultingEvents` field, as well as informing the events caused by any input messages from this event that this event is non-critical. Those receiving events will decrement their states' `resultingEvents` fields, as well. The state is not yet fossil collected, though it need not be saved any longer. The fossil collection will take place in `objpast()`, in order to centralize all forms of fossil collection.

`informNonCritPredecessors()` then must inform any events that sent it input messages that did not cause this event to be on the critical path. It calls `nonCritPathMsg()` for each such message. `nonCritPathMsg()` sets up a system message of type `CRITRM`. `nonCritPathMsg()` then calls `GetLocation()` to find the location of the sender of the message that did not cause this event to be on the critical path. If the location information is available locally, and the process is local, `successorNotOnCP()` is called, to avoid sending a system message to the same node. If the information is available locally, but the process is not local, then `sysmsg()` is used to send it. If the information is not available locally, then `FindObject()` is called, with `finishNonCritPathMsg()` as one of its parameters. (See Chapter 17 for

complete descriptions of the behavior of `GetLocation()` and `FindObject()`.) When `finishNonCritPathMsg()` is eventually called, it will in turn send the system message.

`successorNotOnCP()` will eventually be called for the process that sent this message, whether directly from `nonCritPathMsg()` or as a result of the receipt of the system message. This routine finds the state for the event that sent the message (which may or may not have been truncated, by this point) and decrements its `resultingEvents` field. If that field goes to zero, then `nonCritEvent()` will be called for this state, as well, and will behave just as described above.

Returning to the fossil collection procedures of `objpast()`, after all states have been examined for truncation, all truncated states are examined to see if they have been fully removed from critical path consideration. Any state whose `resultingEvents` field is zero is not on the critical path, and has so informed all of its predecessors. (Its successors already know – they were the ones informing it that it wasn't on the critical path.) Such a truncated state is removed from the queue and destroyed.

Input messages are also preserved by the critical path facility, to serve as backward pointers to the cause of events. Like states, committed input messages being saved for critical path computation are truncated, chopping off their data section. Unlike states, however, the entire message header is preserved, to facilitate handling of the truncated message. With the entire message header in place, the message can be left in the normal input queue, rather than requiring a truncated input message queue, thus simplifying much of the handling code. Truncated messages can cause fragmentation, so a method to limit that fragmentation would be valuable. Time did not permit adding such a feature, which would work along the same lines as the special pool of contiguous memory for allocating truncated states.

When `objpast()` examines an input message, it checks to see if the message's flag indicates that it is a `NONCRITMSG`. All messages start off with this flag value turned off. It is turned on when it is determined that this message did not cause the resulting event to be on the critical path. (That determination is made when either this message proves not to have the highest EPT contribution for the event, or when the event is determined not to be on the critical path.) If `NONCRITMSG` is set, then `delimsg()` is called to fossil collect the message. If `NONCRITMSG` is not set, then `truncateMessage()` is called to chop off the message's data part.

The third major section of code dealing with critical path computation actually calculates the critical path. It is called from `gvtupdate()` when the simulation ends, only if the critical path calculation has been requested by the user. The primal function for this phase of critical path calculation is `calculateCritPath()`.

`calculateCritPath()` is called on each node at the end of the run. One of those nodes is the critical path master, and takes slightly different actions than every other node. Other nodes simply look through their scheduler queue for the OCB with the final event with the largest EPT. This event is the local event that logically had to be run latest of all those on this node. Globally, such an event over all processors must be the final event on the critical path, as no other event necessarily had to finish as late as that one. Once the local event is found, a system message is sent to the critical path master with the local contribution.

The critical path master calculates its own contribution and waits for all other nodes' contributions to arrive. As each arrives, `checkCritPath()` is called on the critical path master node. `checkCritPath()` compares the incoming contribution to all contributions seen so far. If this node's event has the highest EPT seen so far, the EPT, object name, and its node are saved. Once all contributions arrive, `startCritPath()` is called.

`startCritPath()` checks to see which node has the last event on the critical path, that event being the one chosen by `checkCritPath()` from all nodes' local EPT maxima. If the event is on the critical path master node, a system message is dummied up and `takeCritStep()` is called.

`takeCritStep()` takes one step backwards along the critical path. It finds the required process in the scheduler queue, and the state within that process. (There is a bug in this code if the object with the event being sought has two phases on this node, but that situation shouldn't arise when the critical path is being computed, as all non-user requested phase splitting is suppressed when the critical path is being computed, and users typically do not request phase decomposition on their own.) Once that state is found, `makeCrit()` is called. `makeCrit()` sets a bit in the state's flag to indicate that it is on the critical path and searches for the input item that caused it to become critical. If an input message caused this event to be on the critical path, a system message of type `CRITSTEP` is set up for the process that sent the input message. `GetLocation()` (and possibly `FindObject()` and `finishCrit()`) are used to locate the process. If the process is local, `takeCritStep()` is called recursively. If the process is non-local, the system message is sent. Upon receipt, that message will cause `takeCritStep()` to be called.

If the critical path event is on the critical path due to the previous state, then `makeCrit()` is called recursively for that event.

Termination of this process occurs when the sending process for a message proves to be "TW", indicating that the message was sent from the configuration file. At this point, `outputCritPath()` is called. `outputCritPath()` runs through all the processes on the local node, printing out a critical path record for each state they have marked as a

CRITSTATE. Then this node calls endCritComputation() to inform all other nodes that the critical path has been completely found. endCritComputation() sends a system message to all other nodes of type CRITEND. Receipt of this message causes each of those nodes to call outputCritPath() to print out their own local portion of the critical path.

The critical path log produced as a result is not ordered, so a postprocessing tool must be used to order it. (Use of this tool is covered in the TWOS User's Manual, Section 7.5.)

The critical path code could be changed to write out the critical path events as each state is marked as a CRITSTATE, rather than waiting until the path has been traversed. Time did not permit this change. Note that the node that detects the first event on the critical path would still have to send system messages to all other nodes, at this point, to ensure clean termination of the system. Otherwise, those nodes would hang waiting further critical path computation activities.

The critical path computation code could use some cleaning up. It works correctly for most cases, unless it runs out of memory. Further work could lessen its memory usage. Also, the code contains a number of paths that should never be taken, left over from earlier implementations. These should be removed. Time did not permit us to either improve or clean up the critical path facility any further.

Chapter 17: Phase Location

Object names in TWOS are character strings assigned by the user. Whenever the user creates an object, either statically from the configuration file or dynamically in an event, the user or his code must specify a character string name to assign to that object. All messages sent to that object are addressed with that name. Each object must have a unique name, a restriction that is enforced by TWOS at the commitment of virtual creation time for the process.

When a process sends a message to another process, addressed only with this character string name, TWOS must find the node hosting the process and the address of the object control block on that node. Any process may send a message to any other process at any time, without setting up any channel to that process. The destination process may be on the same node or a different node. Moreover, since TWOS attempts to conceal from its users how many nodes are being used for a run, and even that the run is being done in parallel, rather than sequentially, the user's application code has no responsibility for addressing messages, other than specifying the name of the destination process.

A further complexity is that TWOS processes can move from node to node. This migration is transparent to user code, so the operating system must handle all problems of routing messages to moving processes.

Another difficulty is purely internal. As described in Chapter 12, TWOS processes are not objects, but phases. Generally speaking, the decomposition of objects into phases is transparent to the user, so, again, the operating system must take complete responsibility for routing a message not just to the correct object, but to the process handling the correct phase of that object.

TWOS is intended to be a scalable system, so solutions to this problem must work well for large numbers of nodes, large numbers of processes, and large numbers of migrations.

The basic method used to locate processes in TWOS relies on a per-node cache to respond to most requests. This cache keeps the location of the processes recently requested by the node, with a least recently used replacement algorithm. Each cache entry contains the identity of a process, its location, and cache replacement information. A process' identity consists of the character string name of an object and a virtual time indicating which phase of that object is represented by the process. Since any given virtual time is the virtual phase begin time of only one phase of an object (see Chapter 12), the virtual time portion of the process identity is the phase's begin time. The process location also consists of two fields. The first is the node number of the process. If that node number is not the same as the local node number,

the second field is not used. If, however, the cache entry indicates that the process in question is local, the second field of the location part of the cache entry contains a pointer to the process' phase control block.

However, sooner or later a node will receive a process location request that cannot be satisfied from its cache. In such cases, the node must have some other means of locating the process. The method used by TWOS to reliably find processes is to assign each process a *home node*. That node has permanent responsibility for keeping track of the location of the process. Given the character string name of the process, any node can immediately determine its home node by applying a hashing function to the name. The hashing function has been chosen to spread out home node responsibilities in a reasonable way.

All phases of a single object have the same home node. Whenever an object is split into phases, the home node must be notified of the split so that it knows that the object is composed of multiple processes. Whenever a process moves, the home node must be notified of the new location of the object.

`GetLocation()` is the basic operating system routine for finding processes. It first calls `FindInCache()` to look in the local node's cache. The cache is accessed via hashing the object's name to one of the hash buckets that make up the cache. That bucket contains a linked list of entries with the same hash value. The list is reordered on accesses to bring frequently accessed items near the front of the list. If `FindInCache()` finds the entry, `GetLocation()` returns a pointer to the cache entry.

Should the process' entry not be in the cache, the next method used is to look through the local scheduler queue for it. `FindInSchedQueue()` implements this search. `FindInSchedQueue()` runs through all scheduler queue entries comparing the requested process' name to the name in the OCB; if a match is found, the time of the request is compared to the phase begin and end of the scheduler queue entry. If `FindInSchedQueue()` succeeds, an entry is put into the cache (using `CacheReplace()`) and a pointer to that entry is returned.

If the process does not have a cache entry and is not in the local scheduler queue, there is one more set of local information to examine before `GetLocation()` gives up. `GetLocation()` hashes the object name to its home node. (The hashing function here does not necessarily produce the same results as the hashing to cache buckets described earlier.) If the home node is local, the local home list will contain correct location information for the process. `FindInHomeList()` looks for it.

`FindInHomeList()` uses a hashing structure similar to that of the cache, with each hash bucket containing a linked list of entries. Searching for the entry involves applying the hash function and traversing the linked list until the proper entry is found, or all entries have been examined.

If this node is the requested object's home node, `GetLocation()` expects `FindInHomeList()` to find the appropriate entry. If `FindInHomeList()` does not, then this is the first time any request for the object (not just the phase, but the entire object) has reached the home node. This suggests that the object does not yet exist, and must be dynamically created. (See Chapter 14.) `GetLocation()` calls `ChooseNode()` to decide where to put the new object (currently, always on the local node), and then `MakeObject()` to set up an OCB and queues for it. Then `AddToHomeList()` is called to create the home list entry for the object. Whether or not the object had to be created, `GetLocation()` now has sufficient information to set up a cache entry for it, so it calls `CacheReplace()` to do so, and returns the resulting cache pointer.

But if the cache had no entry, the phase wasn't stored locally, and this wasn't the object's home node, then `GetLocation()` can do no more. It returns a null pointer.

The code making the request now must decide how badly it wants to find the phase, knowing that doing so will require sending an off-node message. A further complexity is that the code making the request must be set up in such a way that it can be suspended and resumed later when the message arrives, as TWOS cannot afford to hang waiting for the message. In almost all cases, the information must be obtained, no matter what, in which case `FindObject()` is called.

`FindObject()` takes all the parameters of `GetLocation()`, plus a pointer to a message, a pointer to a routine to be called when the process' location is found, and a flag indicating whether the item stored in the pointer to the message really is a message or not. (`FindObject()` can be called from a number of places, and what needs to be saved to resume execution may not always be a message, though it most typically is.) `FindObject()` always assumes that `GetLocation()` has already been tried, so it makes no attempt to find the process locally. It sets up an entry in the *pending list* for the request, instead. The pending list is a list of unfinished work awaiting the arrival of object location information. All the information passed in to `FindObject()` as parameters is stored in the pending list entry.

Generally speaking, if `FindObject()` is called, a system message is sent to the home node requesting the location of the process. The local node then goes on to do other work. However, in certain cases there may be many requests for exactly the same piece of process location information. The first request is queued in the pending list and a message will be sent off. However, when the response to that message arrives, it can be used to satisfy any number of identical requests, so the second request for a particular piece of location information merely causes an entry to be put into the pending list, using `AddToPendingList()`.

If the request is not for the same object at exactly the same virtual time, then a new message must be sent to the home node asking for the necessary location information. This message is called a "home ask" message, and it will be responded to with a "home answer". After using `AddToPendingList()` to save the necessary information for dealing with the home answer, `FindObject()` hashes the name to the home node number and sends off a message to that node, of type `HOMEASK`. `FindObject()` then return.

Routines that call `FindObject()` cannot expect to get their information immediately. Since they typically need that information before proceeding, when `FindObject()` returns they usually return themselves. While their work has not been finished, they are assured that it will be eventually. `FindObject()` must be used with care in cases where literally no system activity can go on until the location information is available. As of yet, no such situation has arisen in the implementation of TWOS.

When the home ask message arrives at the home node, `ServiceHLRequest()` is called to handle it. `ServiceHLRequest()` calls `FindInHomeList()` to search its existing home list for the information. If the information isn't found, then the object is dynamically created, just as described in the case where `GetLocation()` is called for a local home node. In either case, the resulting location information is put in the home list and made available to `ServiceHLRequest()`, which packages the information up in a system message of type `HOMEANS` to the node that needed the information. No entry is made in the local cache, as this node has no reason to believe its resident processes are especially interested in the process in question. Home ask messages do not look in the home node's cache first, as they know the correct information will be in the home list, but cannot be sure it will be in the cache. Looking up information in the home list is no more expensive than looking it up in the cache, so it is sensible to look where the information is sure to be, first.

When the node that made the request for the location information receives the `HOMEANS` message, it calls `ObjectFound()`. `ObjectFound()` first calls `FindInPendingList()` to look for an entry matching the information in the `HOMEANS` message. In some cases, the pending list entry might have already been removed due to an earlier `HOMEANS` message, but most often it will still be there.

Once the entry is found, `ObjectFound()` checks to see if the `HOMEANS` message says that the requested process is local. Remembering that `GetLocation()` already checked for this case, one might think that the `HOMEANS` message could not possibly say that the process is local, but various timing conditions arising from process migration (see Chapter 13) could lead to this situation. In fact, the local node might still not have the process in its

scheduler queue, despite the home node indicating that it should be there. In this case, the process is still on its way, but hasn't arrived. In this situation, `ObjectFound()` removes the old entry from the pending list and calls `FindObject()` again, more or less to delay until the process actually does arrive. For especially slow migrations, a `HOMEASK/HOMEANS` pair might bounce back and forth several times before the request can actually be passed up to the code that needed it.

More often, however, there is no problem with the home node's response, so a new cache entry is made for it and `RemoveFromPendingList()` is called. That routine not only pulls the pending list entry out of the pending list, but also calls the routine that was supplied to `FindObject()` as a restart point. When `RemoveFromPendingList()` returns, `ObjectFound()` looks through the pending list for any more entries this request satisfies and calls `RemoveFromPendingList()` on them, too. A `HOMEANS` contains sufficient information to identify all requests that involve the process in question, so a single `HOMEANS` message may clear out entries generated for a large number of `HOMEASK` messages. One implication of this ability to remove multiple pending list entries with a single `HOMEANS` message is that `ObjectFound()` will often receive `HOMEANS` messages that no longer have pending list entries. Therefore, `ObjectFound()` does not regard failure to find a matching pending list entry for a `HOMEANS` message as an error. Note that requests that the sender cannot identify as identical may be able to be satisfied with a single `HOMEANS` message.

A typical calling sequence for phase location is that of `deliver()`. `deliver()`, covered in Chapter 3, handles user message delivery in a very general way. It calls `GetLocation()`, and, if that routine returns the needed information, uses it to ship the user message wherever it needs to go. If `GetLocation()` returns null, `deliver()` might call `SendCacheInvalidate()` (if this message came here from another node, to make sure that node does not send any more messages for processs no longer here), and then calls `FindObject()` with the message and a routine called `FinishDeliver()` as two of its parameters. `deliver()` then exits. When `RemoveFromPendingList()` handles the pending list request for this message, it calls `FinishDeliver()` with the message and the location. `FinishDeliver()` is then able to enqueue the message locally (if the desired process migrated in to the local node since the original request was made) or ship it to the node that does have the process, using `sndmsg()`.

This method of process location may store messages temporarily in a pending list. Therefore, when the GVT algorithm is run, each node must also look for the virtual times of messages in its pending list, as the lowest-time event in the system could be represented by a message in that pending list. (See Chapter 7 for further details.)

A certain amount of care is necessary to handle the home node of a migrating process. There are points in the migration protocol in which neither the sending node nor the receiving node are capable of accepting messages for that process. In such cases, the home node informs nodes trying to locate the migrating process that it is at the destination node. If messages are delivered to that node before the process is ready to receive them there, the destination node will simply disclaim information about the process, resulting in a reconsultation of the home node. The home node will again indicate that the destination node holds the process, which the destination node may again deny. Sooner or later, however, the destination node has set up the incoming process sufficiently to accept the incoming message.

Care must also be taken to clear the cache entries of the source node in a migration. Before the migration, the entry not only shows the process as being stored locally, but specifies a pointer to its process control block. If a message were delivered to the source node without clearing this cache entry after the process is migrated, the process control block pointer from the cache entry would point to freed memory. If TWOS attempts to use that pointer, the system could crash. Therefore, the local cache entry is always cleared early in a migration.

In certain cases, some of which were described above, nodes can determine that other nodes have incorrect cache entries. When this happens, `sendCacheInvalidate()` is used to force removal of a bad cache entry. This routine simply format a system message of type `CACHEINVAL` and sends it to the node with the bad cache entry. The arrival of this message causes the node to run `RemoveFromCache()`, a routine also used for locally generated cache invalidations.

Temporal decomposition and process migration does not cause immediate clearing of incorrect cache entries for all nodes. Some nodes may preserve a stale cache entry for a migrated process for quite a long time, even through the end of the simulation. However, the stale cache entry will only persist until information is sent for that process to its old location. Then, when that location forwards the message to the new location, a cache invalidation message is also sent to the node with the stale entry, forcing it to reconsult the home node before sending any further messages to the process.

The phase location cache is of limited size, so sometimes entries must be removed even when there is no evidence that they are incorrect. `CacheReplace()` is used, in such cases. It calls `ChoosePosition()` to select an entry to remove. `ChoosePosition()` currently uses an LRU algorithm to select an entry for removal. Whenever a cache entry is used, its replacement field is set to a monotonically increasing, per-node counter's incremented value. Therefore, the *n* most recently used entries have the *n* highest replacement field values. In particular, the least recently used entry has the

lowest value. `ChoosePosition()` looks through all cache entries for this lowest replacement value.

`ChoosePosition()`'s search can be somewhat expensive, but it is rarely necessary. A typical TWOS simulation might consist of 500 or more objects spread across 32 or more nodes. For such cases, typical cache hit ratios on process location requests exceed 99%, and sometimes exceed 99.9%. Most simulations run under TWOS use objects that have substantial locality of reference in their communications patterns. Moreover, many of the misses are from cache initialization, before it has filled up with its maximum entries. No searches for least recently used entries are made until all unused entries are exhausted. Optimizations to this cache method are no doubt possible, but the high hit ratios and modest cost of the existing algorithm have made improvements to it a low priority.

Chapter 18: The Tester

The TWOS tester is a component of the operating system that serves two purposes. First, it is a parser for reading configuration files. Second, it is a debugger for finding operating system problems. This chapter will discuss the tester's functions and describe how to make some basic changes to the facility. It will not describe all of the tester commands and what each command does. For a list of interesting tester commands, see Appendix H.

The heart of the tester code is in a routine called `command()`. This routine, and several routines it calls, contain exceptionally dense, difficult code. Fortunately, they almost never needs to be changed. (Only if a new data type is to be added to the parser need `command()` be changed, or if the system developer insists on fixing a couple of small, slightly annoying bugs in the tester.) This manual will not cover `command()`, in detail.

`comdtab.c` contains the code that is more frequently revised. The tester is essentially a command interpreter, and the operation that most frequently must be performed is to add a new command. The next most frequent operation is to slightly modify an existing command. Both of these operations can be performed mostly within `comdtab.c` .`comdtab.c` consists almost exclusively of data definitions used by `command()` as parser tables. Adding a command to tester largely consists of altering these tables, and changing a command largely consists of altering table entries.

The list of commands for tester is kept in an array called `func_defs[]`. Each entry in this array consists of the command name, a short help message, a flag indicating whether the command should be broadcast to all nodes or just performed on the node tester is running on, the name of the TWOS routine that executes the command, and an optional list of parameters. The first two parameters are character strings. The command name is written in caps, but the command can be issued in any combination of capital and lower case letters. The help message (which shouldn't exceed four or five words) is also a string in caps, but will appear in lower case when printed. The valid values for the third parameter are 0 or `BCAST`. The next parameter is the name of the function that implements the command. This function must be written when a command is to be added, of course, but in most cases it is very straightforward and can be modelled on an existing command. The parameter list should consist of the addresses of variables that will hold the command's parameter values.

The function that is to be used must be declared before it can be used in the `func_defs[]` array. Since `comdtab.c` does not itself contain these functions, the function should be declared as externals before the declaration of `func_defs[]`. Any of the existing functions can be used as a model for this external declaration.

The arguments of the command should be separately declared and then entries should be made for each in the `arg_defs[]` array. If the entry is not made here, the parameter will almost certainly not be passed correctly to the function by `command()`. The format is rather simple. The first part of each entry is a short help description, typically one or two words. The next part is the tester data type of the argument. The third part is the address at which the argument is to be stored, which must match the address used in the `func_defs[]` definition.

Tester supports a limited number of data types, called `INTEGER`, `REAL`, `HEX`, `STRING`, `STIME`, `SYMBOL`, and `NAME`. `NAME` and `STRING` are the same, except that a `NAME` can be no more than 29 characters, while `STRING` can be of arbitrary length. `INTEGER` is a decimal integer, `HEX` a hex integer. `REAL` is a real number in xxx.yyy format, not in floating point format. `STIME` is a simulation time, which is essentially the same as a `REAL`. `SYMBOL` refers to a limited set of defined symbols. The only ones currently defined are "`CMSG`", "`EMSG`", and "`GVTSYS`". Minor changes to `command()` can be made to define new tester data types, but the existing ones are sufficient for most purposes.

The steps in adding a command to tester, then, are

1. Add an entry to `func_defs[]`.
2. Declare the related function earlier in tester.
3. Declare the arguments (if any) in `arg_defs[]`.
4. Declare the arguments as static entries of their types earlier in tester.
5. Write the function that implements the command.

The tester is called in one of three ways. First, it is invoked at the beginning of the run to read the configuration file. Second, when an error condition is detected by the operating system, tester is typically called to help diagnose that condition. Third, the user can stop any run, once the configuration file has been read and the run is underway, trapping to tester.

The first use of tester occurs towards the end of system initialization. The tester is called by `init_command()`, which in turn is called from a non-looping portion of `Main_Node_Execution_Loop()`. This latter routine contains TWOS' main loop (see Chapter 21), but also contains much of the initialization code for TWOS. Only node 0 calls `init_command()`, so node 0 will do the actual reading of the configuration file, parcelling out the commands to the nodes that need them.

`init_command()` opens the configuration file and then repetitively calls `command()` to read a single command. As mentioned earlier, `command()` is the heart of the tester. It calls `get_word()` to read one word of a command, which will initially call `get_line()` to read in a full line to extract the word from. `init_command()` has set up a file pointer for the configuration file, which signals `get_line()` to read the configuration file, rather than wait for input from the console. `command()` will eventually get an entire command and execute it. If the command is a broadcast command, it is sent to every node. If it is not a broadcast command, execution starts on node zero. Certain commands cause messages to be sent to other nodes, which may indirectly cause other actions. For example, an object creation from the configuration file is handled by node zero sending a CMSG message to the node that is to host the new object. Once all commands from the configuration file have been read and handled, `init_command()` returns and the main loop goes on to further system initialization tasks.

The second major use of the tester is to trap to a debugger when TWOS detects an internal error. The tester is not called immediately for user errors that might be rolled back later. Instead, the executing process' state is marked as in error and, if committed, an error is generated at that point. (See Chapter 8 for further details.) But when the operating system itself detects an erroneous condition, the most common pattern in TWOS is to call `twerror()`, which prints an error message, and then to call `tester()`. `tester()` repeatedly calls `command()`, exiting when a variable called `host_input_waiting` is set to zero. (One of the tester commands, "go", does just that, permitting the programmer to restart a TWOS run after a call to `tester()`.) Any node can call `tester()`, and one result of the call is that all nodes will halt and await tester commands.

The third major use of the tester is to permit a user to halt a TWOS run at any point and examine the state of the system. This capability would only be of use to those debugging the operating system, generally, but can be very helpful for them. One of the actions of the main loop is to periodically check for input from the console. If a carriage return comes from the console, then `tester()` is called. Just as for calls to `tester()` from inside TWOS, all nodes will be halted when `tester()` is called in this way. Each node periodically checks, in the main loop, to see if some other node wants it to trap to the tester.

In certain cases, a TWOS node might erroneously enter an infinite loop, or otherwise get into a non-halting situation where it will never again execute the main loop. In such cases, a request to trap to the tester would be ignored by such a node, as the main loop test for the request would never be examined. TWOS has one way out of such situations. Sending a control-C signal from the console will always cause all nodes to halt and trap to the

tester. (This trap is set up in `BF_MACH_init_args()`, since the signal only works on the Mach GP-1000 implementation of TWOS.) As currently implemented, a given run can only be trapped in this way once. If a run is trapped with this interrupt signal, then restarted, the next interrupt signal will not properly trap to tester.

TWOS behaves differently during system initialization than at other times, so it is almost certainly a mistake to issue tester commands normally used in the configuration file in the middle of the run, or to assume that methods used for such commands will work well once the run has started. For example, sending a CMSG to create an object after initialization has ended, instead of using the dynamic creation system outlined in Chapter 14, is likely to cause problems. Changing various run time parameters, such as whether lazy or aggressive cancellation is in use, after initialization is also likely to cause errors. An obvious exception is the `schedule` command in tester, which simply calls `schedule()` to send a message.

Tester commands can be issued to be executed at a particular simulation time. Such commands can be issued from the configuration file, or at any time that the system has trapped to tester, for whatever reason. This feature is intended to help in debugging TWOS capabilities, not for user purposes. Only tester commands preceded by a node number or an asterisk (indicating that the command is to be broadcast) can have a simulation time attached. The simulation time is a second number before the command itself. Any command preceded by a valid simulation time will be viewed as a command for delayed execution. The command will be stored in a special command queue, along with its simulation time, and will be executed once GVT passes that time. Chapter 21 discusses details of how and when the commands are removed from the command queue.

Chapter 19: Event Logging

TWOS is able to maintain a log of all events performed during its run. This log is primarily used for debugging purposes. The most common use is to compare an event log for a correct run of a simulation to an event log for an incorrect run of the same simulation. By finding the point of first divergence, the programmer looking for the bug can get a pointer towards what might have caused the error. The event log is used extensively by programmers developing TWOS simulations to locate problems in determinism caused by inadvertently breaking some rules of user behavior, or to detect inconsistent results between the simulation running on two different hardware platforms, usually caused by incompatibilities between the two platforms' underlying hardware or software.

Event logging can run in one of two modes. Either it can simply produce an entry for every committed event made during the run, or it can read in an existing event log and check each committed event against that existing log when the events are committed. In the former mode, a file called "evtlog.*x*", where *x* is a node number, will be created for each node in the run. These files must be merged to create a single event log; this merge can be easily accomplished with standard UNIX tools. In the latter mode, the first discrepancy will cause the system to trap to tester. (See Chapter 18 for details of tester.)

If the checking mode is in use, then each object needs to have a special data area attached. This data area is allocated and its address stored in the object's OCB by code in `createproc()`, the routine used to set up a new object. The entire log file is scanned for any entries related to this particular object, and those entries are copied into a special temporary data area. This temporary area must be very large, as it must have enough space for all events. Once the temporary area has gotten all entries for this object, its true size is known. A new chunk of memory of exactly that size is allocated, the event log entries are copied from the temporary area to that correctly sized chunk, and the object's `evtlog` pointer is set to the address of that chunk.

The major body of the event logging code is in the commitment mechanism, in `objpast()`. (See Chapter 8 for full details of commitment.) `objpast()` first checks to see if the event log has yet been opened, and opens it, if it has not. If the event log is being produced, rather than checked, as each output message bundle is committed, `HOST_fprintf()` is called to write an event log entry for it. (An output message bundle is all messages sent by a particular object from a single send time.)

If event checking is being performed, the number of messages in each output bundle is checked against the reference log in the object's event log area. The reference log is ordered by send time, and a pointer is maintained into it that

points at the next unmatched bundle. If the bundle about to be committed exactly matches the record for the next unmatched bundle in the reference log, then the checking procedure moves on to the next bundle. If it does not, either there is an extra bundle in the reference log, or this bundle being considered isn't in the reference log, or the bundle being considered is in the reference log, but has an improper count of output messages. An appropriate error message is generated and tester is called.

Event logging has a rather high overhead, especially if event checking is being performed, so event logging is not done, by default. In fact, the code is ifdefed out of TWOS by default, requiring recompilation when it is to be used. For many simulations, event checking will not work at all, as nodes do not have sufficient memory to hold the entire event log. A truncated version of the event log can be read in, instead, if the error being chased occurs relatively early in the simulation, but event checking will not provide much help if the error occurs near the end of a lengthy simulation. A robust method of allowing event checking on partial event logs with arbitrary start and end times, and for arbitrary objects, has been proposed, but time has not permitted its implementation.

Chapter 20: I/O

TWOS is unable to use standard, pre-existing I/O capabilities of the machines it runs on because output might need to be rolled back. TWOS handles this problem by holding the writing of all output to its device until the output is committed. In addition, limitations of the existing platforms make I/O rather difficult and expensive, which would artificially slow down simulations run in parallel. TWOS tries to overcome some of these limitations by implementing its own I/O.

TWOS permits users to open both read files and write files. Read/write files are not currently permitted. Each file opened uses one slot in a table of static size, for the entire course of the run. The parameter `MAX_TW_FILES` controls the number of slots in this table, thus also defining the number of files that TWOS may open at any time during the run. Additionally, each process has a limit on the number of streams it can have open simultaneously. This limit is defined by the parameter `MAX_TW_STREAMS`. Both of these parameters can only be changed by recompilation.

Each file has its own unique TWOS name, which must be bound to the name of the file in the underlying operating system's file system. The configuration file command `getfile` performs this binding for input files, while the configuration file command `putfile` performs this binding for output files, as described in the TWOS User's Manual, Section 2.2.3. At most, a combination of `MAX_TW_FILES` `getfile` and `putfile` commands are permitted during a single run. `getfile` causes the entire file to be read into TWOS memory on all nodes, so reading very large files can have bad effects on TWOS' performance. Having a local copy on all nodes is necessary because the I/O channel to the disk copy is very slow and narrow, and maintaining a copy of the file in the memory of only a few nodes might put too heavy a burden on the hosting nodes.

`putfile` behaves somewhat differently. It, too, uses up a slot in the table for the name translation, but it does not have to allocate or use memory to hold the data of the file. Instead, it creates an object of type `STDOUT` on node 0, with the name of the file.

In addition, the user has access to the standard output through the `tw_printf()` call, described in the TWOS User's Manual, Section 4.4.3.1. Any data printed using this call will appear in a file called "`STDOUT`". Note that objects of type `STDOUT` are not quite the same as objects that handle standard out output. The remainder of this chapter will make the difference clearer.

Files are opened in TWOS using the `tw_fopen()` system call (see TWOS User's Manual, Section 4.4.1 for details). Each object must perform its own

tw_fopen() call to access a file, so if every object needs to access the file, every object must perform a tw_fopen(). The various phases of a single object inherit the file pointer from the tw_fopen() from the previous phase, so only one tw_fopen() need be done per object, not one per phase. Each process maintains its own pointer to the current position within the file, so each object reads the file independently of all other objects.

The tw_fopen() call searches through the table of all files for which getfile or putfile commands have been executed, looking for the named file. If it is found, and the requesting object has a stream slot open the file in question is available and the user is given a pointer into the object's table of open streams. This pointer can then be used in tw_fprintf() or tw_fscanf() calls, whichever is appropriate.

tw_fscanf() finds the correct entry in the requesting object's stream table, then tries to read data from the last position read in the file. The method of reading is similar to that for normal UNIX scanf() calls, but only one item can be read with tw_fscanf(). If the attempt was successful, 1 is returned. 0 is returned if the requested formatting could not be done properly, and EOF is returned if the end of the file was reached. The other TWOS input calls work analogously.

tw_fprintf() works very differently. After performing some formatting work on its arguments to put them into a single string of text, tw_fprintf() calls tw_fputs(). tw_fputs() calls schedule() with a message consisting of the string just produced, to be sent to the object with the name of this file. The selector is assigned a unique number that is incremented each time tw_fputs() is called for a given file. Each object writing to the file has its own such sequence number.

Thus, a request to tw_fprintf() will result in a message being sent to a special object for the requested file. That object was created by the putfile command, and will be on node 0. It is an object of type STDOUT, and objects of that type receive special treatment. They are not considered for GVT purposes, and they are never migrated by dynamic load management. More directly to the point, a STDOUT object starts life with its SVT at positive infinity, and never rolls back, no matter how many messages arrive for earlier times. As a result, such objects never schedule events. Instead, their input messages just sit in the input queue waiting for fossil collection at commitment time.

At commitment time, in objpast() (see Chapter 8 for full details), the first thing done for STDOUT objects is to call commit(). commit() runs through the object's input queue, finding all messages that can be committed and printing them out to the requested file in the underlying operating system's file system. These messages are then removed from the queue and destroyed.

The standard out object on each node works somewhat similarly. `tw_printf()` is used to write to the standard out file. Unlike `tw_fprintf()`, the standard output messages are not immediately shipped off to an object on node zero. Instead, they are sent to the local standard output object stored on every node. Such objects are similar to file objects, in that they never run, never roll back, never migrate, and do not contribute to the GVT computation. When commit time is reached, `commit()` is called for the local standard out object, once its turn comes. However, generally the individual nodes cannot physically write to the output file. (And we wouldn't want them to, if they could, as the result would be a badly scrambled file, as multiple nodes wrote to it in an undisciplined fashion.) Instead, any committed messages are sent to the IH node.

The IH node is a holdover from an early architecture that TWOS ran on. It is a node that is able to perform physical output from the parallel machine to a hardware storage device holding the standard out file. On the BBN GP1000, it is a special node that is used by TWOS but does not participate in the Time Warp computation. `sndmsg()` is used to get the message to the IH node. Upon its arrival, it is written to the standard out file.

Chapter 21: The Main Loop of TWOS

The main loop of TWOS is the driving engine of the operating system. This loop repetitively executes, alternating between running processes, handling messages, migrating states, and performing other operating system work. Experience has shown that the implementation of this loop can have significant effects on the performance of the operating system.

After system initialization has been completed, TWOS enters the main loop. All user and system code executed from this point until the termination code is called from within the loop, directly or indirectly. TWOS' main loop is contained in a function called `Main_Node_Execution_Loop()` on the GP-1000, and called `main()` on all other implementations. (Having the same looping code in different functions for different machines is rather sloppy and confusing, but time does not currently permit fixing this problem.) The remainder of this chapter will assume that `Main_Node_Execution_Loop()` is the function containing the main loop, as this manual is primarily to be used with a GP-1000 implementation of TWOS.

The main loop of TWOS is an exceptionally difficult piece of code to read, not so much because of complexity of its code as because of the multiple conditional compilation pre-processor statements. It's often taxing to determine whether a particular line will or will not be compiled for any particular platform. The description below will apply exclusively to the version of the loop compiled for the GP-1000.

`Main_Node_Execution_Loop()` also contains some initialization code. First, in some GP-1000 specific code, `Main_Node_Execution_Loop()` starts off by setting up the node number, configuration file name, and statistics file name, then determines how much memory to allocate for TWOS use. Once this memory has been allocated, it loops to touch all pages that have been allocated, plus all pages of TWOS' heap and text area, thus ensuring that the Mach virtual memory system will bring all of these pages into physical memory. The main loop routine then calls `butterfly_node_init()` to initialize all nodes. This initialization largely concerns timers and low level message sending data structures. `BF_MACH_init_args()` is then called to do further, similar machine specific initialization.

Next, machine-independent initialization is done. This initialization includes setting up the type table, initializing phase location data structures, initializing process migration data structures, initializing I/O data structures, setting up the GVT graph (described in Chapter 7), setting up some message sending and receiving data structures, and preparing to read the configuration file. Then `init_command()` is called to read the configuration file, as discussed in Chapter 18. When `init_command()` returns, the actual loop begins.

The TWOS main loop is an infinite loop. It will execute until the end of the run is reached, at which point an exit statement will cause it to terminate. (This exit statement is contained in a routine called by the main loop, not the loop itself.) Much of the code in this loop is machine specific. This chapter will attempt to discuss the loop at the higher, machine-independent level, but, when necessary, will cover the GP-1000 version of the loop's code.

If `host_input_waiting` is not true, the main loop first calls `check_alarm()` and `check_for_events()` to see if some low level event has occurred, such as the GVT interval timer going off, or the dynamic load management timer going off, or if the system termination signal has been received. If one of the timers went off, the requested protocol is not necessarily started. Rather, a counter is simply incremented and, if necessary, another timer event is scheduled. Another check is made in `check_for_events()` to see whether the user has sent an interrupt signal to call the tester (see Chapter 18). If the tester has been called, `host_input_waiting` is set to true.

`host_input_waiting` is a flag set when the user has just interrupted the system to trap to tester, or when the system is already in tester. When `host_input_waiting` is set, GVT and dynamic load management should not be run, so the main loop will not test for their timers expiring when this flag is set.

After possibly calling `check_alarm()` and `check_for_events()`, the main loop tests `host_input_waiting` again. If it is true, the loop calls `command()` to read a tester command, and control goes back to the top of the loop, to check again. Once tester is entered, the only actions taken by the main loop will be iterative calls to `command()`, until one of the commands exits from tester by setting `host_input_waiting` to false.

Once `Main_Node_Execution_Loop()` has gotten past the tester and timer related work, it next checks a variable called `rm_msg`. If this variable is non-null, a message has been received by this node. It could be a message from the host computer, giving some form of control instruction, or it could be a message to perform a tester command, or it could be a normal message. This normal message might be either a user-generated message or a system message. `ih_msgproc()` is used to handle message from the host computer, `command()` is used to handle tester command messages, and `msgproc()` is called for normal messages.

The messages handled by `ih_msgproc()` include messages directed to standard output objects (see Chapter 20), error messages, statistics messages (see Chapter 22), messages relating to various logs, a simulation end message, or an acknowledgement of the creation of an object ordered from the configuration file. There is only a single node that receives such messages,

typically node 0. This node is sometimes referred to as the IH node, and sometimes as the CP node.

If the message is a regular message, `msgproc()` must determine which type of message it is, and properly handle it. `msgproc()` consists largely of a case statement. There is one case for each type of system message. If the incoming message is a system message, its type causes the appropriate case to be chosen and the appropriate handling routine to be called. If the incoming message is a user message, the case statement is bypassed in favor of simply calling `deliver()` to handle it. (See Chapter 3 for further details on what happens to user messages at this point.) Whatever type of message this was, `msgproc()` next calls `dispatch()`, a routine discussed in Chapter 4. `dispatch()` might cause an event to be run, or might handle a rollback, or perform any of several other possible actions involving a single process. When `dispatch()` completes this work, however, it will return to `msgproc()`, which returns to `Main_Node_Execution_Loop()`.

If `Main_Node_Execution_Loop()` found an message in `rm_msg`, then after the message is handled, control returns to the top of the loop. Otherwise, the command queue is checked. As discussed in Chapter 18, the tester has the ability to issue commands that will be performed at some particular simulation time, and not before. These commands are saved in the command queue, ordered by the time of their execution. At this point, `Main_Node_Execution_Loop()` checks to see if the current GVT has exceeded the time of the first command in the queue. If it has, that command, and possibly others, should be executed, so `exec_commands_in_queue()` is called. This routine will execute all commands in this queue whose times are before GVT, in simulation time order, by calling `command()` on each of them. Once the queue has been emptied of all eligible commands, `exec_commands_in_queue()` returns to the main loop, which returns control to the top of the loop.

If the loop continues to execute, rather than returning to its top, the next action is a call to `timer_interrupt()` to determine if the GVT protocol needs to be started. If it is started, control returns to the top of the loop. Otherwise, `dlmTimer_interrupt()` is called to check if the dynamic load management protocol needs to be started. If it is started, control returns to the top of the loop.

Assuming the loop goes on, rather than going back to the top, the next action is to check if there are states to be sent from the state sending queue. (See Chapter 13 for details of this queue.) If there are states to be sent, `send_state_from_q()` is called. Whether or not that routine is called, the loop next checks to see if there are messages to be sent from the queue of messages not yet transported off nodes. If there are, and the loop is not planning on executing an event, then `send_from_q()` will be called to send

messages. `send_from_q()` will try to send out as many queued messages as it can, stopping the first time it fails to send one. A flag called `execObj` is used to determine if the loop intends to execute an event or not. When it is false, a message will be sent from this low level queue. When it is true, the queue will not be examined. If `send_from_q()` is to be called, `execObj` will be set to true, so that the next iteration of the main loop will try to execute a process, rather than send a message from the queue.

If the queue is not examined, the loop checks to see if there is some event ready to execute. Event execution is set up as described in Chapter 4, but only at this point does TWOS actually consider switching contexts to run the event's code. If there is an event to be executed, `execObj` is set so that the next iteration of the loop will send a message from the queue, instead of scheduling an event again. On the GP-1000, the main loop will then call `read_the_mail()` to see if any other node is trying to send a message to this node. `read_the_mail()` will try to fill `rm_msg`, if some other node wants to send a message to this node. This call to `read_the_mail()` will only bring a message into `rm_msg` if the message is a system message, or if its virtual time is lower than the event about to be run. If `rm_msg` is filled by this call to `read_the_mail()`, then control will be transferred to the top of the loop, where, as discussed earlier, the message pointed to by `rm_msg` will be handled. In essence, this call to `read_the_mail()` and the subsequent test of `rm_msg` determine if any incoming work has a higher priority than the event about to be run.

If `read_the_mail()` did not find a more important message to handle, the next action is to perform a lot of timings, some of which are conditionally compiled, some of which always occur. TWOS is about to switch into user code, so the timing functions must account for the change in control. A flag called `objectCode` is set to true to indicate that user code is about to execute, and `switch_over()` is called to switch to the user's stack and program counter.

Whether the queue was examined or an event run, the next thing done when control returns to the main loop is to check if `rm_msg` has been set, indicating that an incoming message must be handled. If `rm_msg` has been set, control returns to the top of the loop, where, as discussed earlier, `msgproc()` will be called to handle the message.

Assuming that control has not jumped back to the top of the loop, the next action is to call `read_the_mail()`. This call to `read_the_mail()`, unlike the one made before executing an event, will accept any incoming message into `rm_msg`. This call to `read_the_mail()` is the last action in the loop, so when `read_the_mail()` returns, control simply goes back to the top of the loop, which starts all over again.

A short recapitulation of the actions within the loop itself will be worthwhile, to help clarify the actions of the loop.

1. Check for any alarms that may have gone off. Go to 2.

2. If a tester command needs to be performed from the console, do it and go to 1. Otherwise, go to 3.

3. If there is an incoming message, handle it and go to 1. Otherwise, go to 4.

4. If there are queued tester commands, execute them and go to 1. Otherwise, go to 5.

5. Check the timers for GVT and dynamic load management, in that order. If either went off, start its protocol and go to 1. (Do not start both protocols without first returning to 1.) Otherwise, go to 6.

6. If there are states to send to other nodes, send one of them. Go to 7.

7. If there are messages to send at a low level and we aren't cleared to execute an event, send as many messages as possible and clear the way for event execution on the next iteration. If the sends led to the arrival of a message, go to 1. If the send did not cause a message to arrive, go to 9. If we didn't try to send messages, go to 8.

8. If there is an event to execute, block execution of another event during the next iteration and look for a message indicating that some higher priority work is coming in from another node. If so, go to 1. If not, perform some timings and execute the event. If, as a result, there is an incoming message, go to 1. Otherwise, go to 9.

9. Get the next incoming message, if any. Go to 1.

Chapter 22: Statistics

TWOS gathers many statistics during the execution of a simulation. These statistics permit verification that TWOS executed the simulation correctly, and also can be used to analyze the performance of the system. This chapter covers how those statistics are gathered and printed.

The statistics gathered by TWOS are varied. Some related to the behavior of processes, some to the behavior of entire nodes. Some relate to committed behavior, some count actions that may be rolled back. Some are counts of user visible actions, some count actions that occur below the level the user can see.

Per-node statistics are not kept in any unified place. A large number of variables are scattered throughout the system holding this sort of data. Typically, they are updated in the routines dealing with the actions they count. For instance, several variables keep track of the behavior of the phase location mechanism (see Chapter 17). One such variable is incremented every time a local node's cache is examined, and another is incremented every time the desired entry is found there. These statistics can be used to calculate the hit ratio of the cache. Other per-node statistics keep track of the number of migrations performed, load management data, message buffer pool usage, the largest Ept for any local process (see Chapter 16), and page fault statistics.

Per-process statistics are kept in each process' OCB, in a special statistics data structure. This data structure is called the `stats_s` structure, and contains around 40 statistics. Most of them are integer values. They keep track of the number of messages sent and received, the number of events performed, the number of negative messages sent and received for cancellation purposes, the number of messages returned by message sendback, the amount of time spent in the process, the length of queues, migration counts, the number of states forwarded due to temporal decomposition, the largest size of the dynamic memory address table, the number of dynamic creation and destruction messages sent, and how many events were started and how many completed. In most cases, the number of messages of a certain type that were sent and received are both stored. When the totals for all processes of the simulation are added up, these sent and received counts should match. (See Appendix E's discussion of the `check/measure` program for details on balancing statistics.)

These statistics are also gathered in a variety of places in the kernel, but a significant number of them are dealt with during commitment. `objpast()` calls `stats_garbtime()` for each committed input message and `stats_garbouttime()` for each committed output message. Depending on the type of the message (event, creation, or destruction), `stats_garbtime()`

will increment the correct statistics field to indicate another committed message of that type. In the case of an event message, `stats_garbtime()` might also increment the count of committed event bundles. (An event bundle is a set of event messages with the same receiver and the same receive virtual time.) `stats_garbouttime()` only calculates statistics for event messages, but does count output event bundles, which will not necessarily match up with the simulation's total count of input bundles.

Two other places where many statistics might be incremented are during the sending of a message and during the receipt of a message. `smsg_stats()` is called when a user message is to be sent, in several places in the kernel. These include:

- places in `nq_output_message()` and `nq_input_message()` that might send the message back, instead of enqueueing it (see Chapter 10);
- `cancel_omsgs()`, when messages are cancelled due to rollback, under lazy cancellation (see Chapter 5);
- `cancel_all_output()`, when messages are cancelled due to rollback, under aggressive cancellation (see Chapter 5);
- `get_message_to_send_back()`, when a message is about to be sent back to relieve memory shortages (see Chapter 10);
- and `sv_doit()`, when normal message delivery has been requested (see Chapter 3).

Statistics are written to a file called `XL_STATS` at the end of the run. (A `-s` switch followed by a file name on the TWOS command line forces the statistics to be written to that file, instead. See the User's Manual, Section 2.3.2.1.) The statistics file is created at the start of the run, but no data is written to it until the run completes. Once TWOS detects termination of the run, `dump_stats()` is called. Termination is detected in `gvtupdate()`, when the new GVT estimate reaches positive infinity plus one. (See Chapter 7 for further details.) `dump_stats()` is called before writing out the critical path, if the critical path is to be written from this run. (See Chapter 16.)

`dump_stats()` calls `Excel_head()` to write out a header, then runs through the scheduler queue calling `Excel_body1()` on each entry in that queue. Finally, `dump_stats()` calls `Excel_tail()`. `dump_stats()` works independently on each node.

`Excel_head()` only writes data from node zero, though it is called for all nodes. It formats a number of messages containing header lines that consist mostly of identification information that should appear in the statistics file.

Excel_head() calls send_to_IH() for each of these header lines to get them printed out.

send_to_IH() simply causes a message of type XL_STATS to be sent to the CP node, a special node that does not run the normal Time Warp system, but acts as a utility node for the run. When the message arrives, the CP node will run a routine called ih_msgproc() to handle this special message. (Other messages that use send_to_IH() to get to the CP include error messages, log messages, simulation end messages, and acknowledgements of creations done from the configuration file.) xl_stats_msg() is called to handled a message of type XL_STATS.

xl_stats_msg() accumulates data contained in XL_STATS messages in an internal memory area until that memory area fills up. Then it calls HOST_fputs() to write out the entire data area to the actual file in the host computer's file system. This method guarantees that any single line to be written to the statistics file will appear in a single piece, with no intervening data, but it does not guarantee that all lines produced by a single node will appear contiguously in the statistics file. The arrival of messages from other nodes between two messages for a single node may scramble the statistics file, somewhat. For this reason, all messages sent to the statistics file contain the node number of the sending node in the message's data area. (Typically, most statistics files do contain all data for a single node in a contiguous block, but it is not guaranteed.)

Excel_body1() prints out all per-process statistics. It reads the statistics from the process' stats_s structure and formats a message containing them as text. Excel_body1() then calls send_to_IH() to get the data into the file, and this call to send_to_IH() leads to the same actions as Excel_head()'s did.

Excel_tail() is the most complex of the statistics output routines. This routine writes out all per-node statistics. The basic method is the same as that used by Excel_head() and Excel_body1() – a message is created and send_to_IH() is called to ship it to the CP node, for later printing. However, Excel_tail() sends many more kinds of XL_STATS messages than the other two routines. If it is running on node 0, it sends a message containing the count of event messages sent by the configuration file reader. For all nodes, it calculates the phase location cache hit ratio and sends a message with both the raw data for the cache and the calculated hit ratio. It sends a message with data concerning the number of migrations into and out of the node, as well as a count of how many times messages were misdelivered and had to be forwarded and the number of states sent out and coming in, and, finally, the number of times the pre-interval state optimization saved a state send (see Chapters 12 and 13). Excel_tail() then sends a message concerning dynamic load management, with the number of load management cycles, the

number of times this node was considered overloaded, and the number of times that this node was overloaded but could not find a process to migrate out (see Chapter 11).

On the GP-1000, `Excel_tail()` sends a message from each node with page fault statistics. TWOS tries to avoid page faults, so this message's data is used to determine how well TWOS succeeded. The Mach operating system on the GP-1000 causes each node to have several potentially different node numbers for different software layers of the system, so another message prints out these numbers for each node. Sometimes the low level message sending facility on the GP-1000 is delayed because a receiving node cannot accept its incoming messages, being busy doing something else. Another message is sent to the statistics file containing data about how often each node was blocked because other nodes were not reading its messages.

Another message is sent containing information about process migration and state migration naks. (See Chapter 13.) The next message contains data about the relative success of the limited jump forward optimization on this node (see Chapter 12). TWOS is able to keep information about which processes used up the single largest slice of time without trapping to the system. If this information is being kept, another message is sent containing this node's highest "hog". If internal TWOS timings are being made, a number of messages will be sent containing timing data. Another message contains information concerning the message buffer pool. If the critical path is being calculated, data concerning the performance of the critical path algorithm is sent in a pair of messages. A message containing the node's local contribution to critical path length is sent whether or not the critical path is being calculated. (See Chapter 16 for more details on critical path computation.) A message describing the number of times a single home answer message was able to deal with multiple home ask requests is also sent. (See Chapter 17 for more details on home ask and home answers.) The next message sent contains the number of times a process was blocked due to throttling. (See Chapter 15 for more details on throttling.) The last two messages sent contain the number of home asks and home answers sent and received. (See Chapter 17.)

After all node have completed sending their statistics to the CP, and have also completed any other termination cleanup they must do, each node sends a signal to the CP indicating that it is done. Once all of those signals have arrived, the CP must clear out any data remaining in the statistics data area it allocated to deal with incoming `XL_STATS` messages. Once that data is dumped, the CP closes the statistics file.

The statistics file is a large file, and is not easy to read. The `check/measure` program, discussed in Appendix E, is able to compress some of the data into a more convenient format. This format is typically enough for checking

normal runs, but the entire statistics file is sometimes needed to obtain details on the behavior of particular objects or particular features of TWOS.

Currently, TWOS does not calculate the complete set of message statistics. For instance, the number of dynamic creation messages sent is not counted, only the number received. Also, some statistics that are gathered are not printed. For instance, the number of committed output messages for each process is not printed in the statistics file, even though `stats_garbouttime()` gathers it. TW 2.7 was originally planned to have all of these irregularities corrected, but time does not permit correcting them, under the circumstances.

Chapter 23: Queue Handling

TWOS contains many queues. Each node has a scheduler queue, a process migration queue, a state migration queue, a pending list queue, and several low level message sending and receiving queues. Each process has an input queue, an output queue, and a state queue. (And, if the critical path computation is being performed, a truncated state queue.) Rather than handling each of these queues with special, queue-specific code, TWOS contains general queue handling routines. These routines are located in two modules called `list.c` and `turboq2.c`.

Because of this layer of software, TWOS queues can generally be implemented as any type of data structure that is quick or convenient. In practice, the only data structure supported by the routines in `list.c` and `turboq2.c` is a doubly linked list, so TWOS queues are all implemented as doubly linked lists.

list.c contains very low level queue handling routines. The lists handled here consist of a header element and an arbitrary number of elements in a single order. The routines in `list.c` handle tasks like:

- creating a list element
- destroying a list element (once it has been removed from the list)
- creating a new list
- destroying a list (once all of its elements have been destroyed)
- inserting an element into a list
- determining if a pointer points to a list header
- finding the next element in a list
- finding the previous element in a list
- removing an element from a list

For the most part, these routines are very straightforward, given that the list implementation supported is a simple doubly linked list. Each list element consists of a forward pointer, a backward pointer, a size, a padding integer, and the actual data of the list element. The purpose of the padding integer is to force the data to start on an eight byte boundary, which is of importance to some machines. In the future, the padding integer could be used to contain a tag indicating what kind of list element this is, but time has not permitted

this change to be made. The list header consists only of the pointers, the size, and the padding element. It has no associated data allocated for it.

In practice, certain of these routines are almost never used. Instead, macro versions of them are used, instead. For instance, `l_next()` is rarely used, with `l_next_macro()` being used, instead. Since many linked list operations are so basic, undergoing the overhead of a function call is hardly worthwhile for these operations. If a more complex data structure was put in place, for which a macro version of the function was not practical, the macro would be redefined as simply a call to the function.

`turboq2.c` contains higher level queue management routines that know certain facts about the structures of particular queues. These routines mostly work on the input queues and output queues of processes. Most of the routines in `turboq2.c` are devoted to handling bundles in the input and output queue - finding the next or previous bundle, finding the next message in a bundle, and so forth. A call to `nxtibq()`, for instance, will return the first message in the next bundle in a process' input queue, given a pointer into that queue. These routines are used by the event scheduling code and the critical path code.

There is one extremely general purpose routine in `turboq2.c` called `find()`. `find()` finds a particular thing in a particular queue. Its parameters are a pointer to the queue's header, a starting point for the search, a pointer to some comparison element not yet in the queue, a pointer to a function to be used to compare queue elements to the comparison element, and a variable that will be filled with an integer describing whether the located element is equal to the comparison element or not. `find()` returns a pointer to the element it found.

While it is very general purpose, in practice `find()` is only called in two places by TWOS. Both are in enqueueing routines, `nq_input_message()` and `nq_output_message()`. (These routines are discussed in Chapters 3 and 10.) `find()` is used by these routines to find the appropriate place for an incoming message in either the input or the output queue. A different comparison routine is used for the two, with `nq_input_message()`'s being more complex. (The routine used by `nq_output_message()` is somewhat out of date, but still works properly, though for the wrong reasons. Time does not permit fixing it, at this point.)

Very likely, replacing the linked list data structures with a structure more like a splay tree would improve the performance of many TWOS applications. A linked list is not an especially efficient structure for searching, and every time that a new message is to be sent the output queue must be searched. Similarly, every time a new message arrives, the input queue must be searched. Recent experience has shown that, under certain circumstances, these queues can grow very long, and the resulting search time can become a

very real cost to the run time of the simulation. Changing to a binary tree or a splay tree is conceptually simple, but would require a great deal of fairly mechanical work. Also, if the new data structure could not use simple macros for operations like finding the next or previous element, the cost of using a subroutine, instead, might be more than the benefit of the more efficient data structure, for many simulations.

Chapter 24: Debugging Facilities Internals

TWOS has a number of built-in debugging facilities. One of these, the tester, was covered in Chapter 18. This chapter covers a number of other debugging facilities in TWOS, including paranoid code, the monitor, the flow log that is used as a driver for the fplot graphical utility, the message log that is used as a driver for the mplot graphical utility, and the migration log.

24.1 Paranoid Code

Paranoid code is code that is normally not compiled into TWOS, but can be, for debugging purposes. This code performs much more substantial checking of possible error conditions, especially checking to see that assumptions made before operations are performed are indeed true. Always performing these checks would slow TWOS down unacceptably, but being able to request them when a problem is known to exist has proven very helpful. To compile the paranoid code into TWOS, the makefile should be altered to define `PARANOID` in the `DFLAGS` line, all object files should be deleted, and TWOS should be remade. Once the problem is found and fixed, remember to remove the definition of `PARANOID` from the makefile, delete all object files, and make TWOS again.

24.2 The Monitor

The TWOS monitor is a special subsystem that can be used to watch the operations of the system during a run. It can selectively print out a message each time one of many internal TWOS functions is called, along with the parameters passed to that function. If desired, the tester can be called whenever a particular function is entered, and tester commands can be used to control the monitor. The monitor code is normally compiled out of the system, as it imposes a rather high overhead on almost all standard operations, so, like paranoid code, a separate compilation must be done whenever the monitor is needed. The `DFLAGS` line in the TWOS makefile should be changed to define `MONITOR`, all object code removed, and a make done. As with paranoid code, it is important to completely undo this process after debugging is complete, since the monitor overhead is high.

The monitor works off of four data files that must be present in the directory from which the simulation is being run. These files are called `names`, `levels`, `str`, and `datatypes`. The `str` and `level` files are created by performing a "`make str`" in the Time Warp directory containing the object code for the Time Warp library. The `name` file is made by redirecting the output of the UNIX utility `nm` run on the Time Warp library to a file called `name`. The `datatypes` file does not typically change from version to version of TWOS, so any existing `datatypes` file can be used. One is stored in the `BF_MACH` directory of the release tape.

The `levels` file is the only one requiring any modification by the programmer doing the debugging. Initially, this file is set up so that the monitor will not print a message for any function that is called. There is a line in this file for each function that can be monitored, consisting of the function name and a numerical monitoring level. Initially, all monitoring levels are zero. Changing one of those levels to 1 will cause a message to be printed out whenever that function is called. Changing a level to 20 will cause tester to be called when that function is called. A tester call can also set up the monitor so that only a single process is so monitored. This method is imprecise, as internal TWOS functions do not always contain enough information to identify what process they are called for, and some are not associated with a process at all, but it does a reasonable job of filtering out unwanted information.

The other data files are simply read in and used by the monitor. They need not be altered from run to run. (`str` and `names` must be recreated every time TWOS is recompiled. `datatypes` need never be changed.) They are read in during system initialization by a function called `moninit()`. The details of this function are tedious, yet fairly obvious, so they will not be discussed too closely here. First, `load_funcs()` is called to read the `str` file to get a list of function names and parameters, which is stored in an array called `mon_func[]`. The `datatypes` file is used to translate the string descriptions of data types in the functions' parameter lists into an internal coding.

Next, `load_nt()` is called to read the `names` file. This file contains beginning and ending addresses for all functions in Time Warp. `load_nt()` looks up each function named in this file in the `mon_func[]` array set up earlier and stores the start and end addresses of the function in that array, in the proper entry. An array called `mon_array[]` is then set up to have pointers into the `mon_func[]` array, and the `mon_array[]` is sorted by the beginning addresses.

Next, `load_levels()` is called to read the `levels` file. The monitor level for each function is copied into proper entry in the `mon_func[]` array. Handling the `levels` file completes monitor initialization.

In order to make a function visible to the monitor, the first statement after data declarations in the function must be "`Debug`". When the system is compiled without the monitor, this statement is defined to be a null string, doing nothing. If the system is compiled with the monitor, this statement is defined as a call to `mad_monitor()`.

`mad_monitor()` finds the address of the statement that called it by illegally looking back through the stack. Because this statement is in the function that monitor should print information about, `mad_monitor()` can use the statement's address to find the name of the calling function, by searching the `mon_array[]`. The `mon_array[]` gives a pointer into the `mon_func[]` array,

which contains the function name and information about its parameters. Various other nasty business with the stack extracts the parameters, and the information stored in `mon_func[]` is used to properly interpret the extracted bits. (This interpretation is done in a function called `format()`, which is where the check for monitoring a particular process is made. The check is made by examining the parameters and seeing if any of them refer to the process in question.) Once the parameters are properly interpreted, a line of information about the monitored function is printed and, if the level value is set to 20, tester is called. In any case, once `mad_monitor()` exits, it simply returns to the function that originally called it.

24.3 Flow Logging and `fplot`

One of TWOS' associated graphical tools is the `fplot` program. This program takes a data file produced by TWOS and displays a graphical plot showing all events run during the simulation. The `fplot` program is described in section 7.3 of the User's Manual, so this chapter will only discuss how its data, called the flow log, is produced.

The flow log is produced at the request of the user, from a tester command that should be put in the configuration file. This tester command calls `flowlog()` to set up a data area on each node to hold that node's part of the flow log. Using the flow log takes substantial space, so it is not done by default.

`fplot` draws one line for every event performed during the simulation, so TWOS must produce a record for every event. The flow log entries are made by `flowlog_entry()`. This routine is called by `tobjend()`, a routine in turn called whenever the low level context switching code returns from the end of an event. (Appendix G contains some details of that code for the GP-1000; it tends to be assembly code written for a specific machine.) `tobjend()` will eventually call `objend()`, but first it will take some timings and call `flowlog_entry()`. `flowlog_entry()` uses the timings just taken to determine the length of the event. Then it tries to write a record into the log set up by `flowlog()`. The first time the space set aside for the log is full, if it ever is, `flowlog_entry()` prints a message saying that the area is full. Once the flow log area for a node is full, the run continues, but no further entries are made on that node. Other nodes continue to make their entries until they fill their logs or the run ends.

At the end of the simulation, when the CP has collected simulation end messages from all nodes, it calls `dumplog()` to write each node's flow log to the data file. Each node puts the data in the same data file, so it may be somewhat scrambled, but each entry is written atomically. `dumplog()` figures out how many flow log entries the local node generated and sends a `FLOW_DATA` message for each to the CP. When the CP receives this message, it calls `cp_flog_msg()`. This routine uses complex trickery very specific to the

GP-1000 to read the message, whose contents are stored until all flow log messages have arrived. Eventually, `cp_flog_msg()` will print out all of these messages into the flow log.

24.4 Message Logging and `Mplot`

The `mplot` program is similar to the `fplot` program, but it plots messages, instead of events. One line is drawn for each message sent, on a virtual time versus real time graph. The line starts at the virtual time, real time position at which the message was sent, and ends at the virtual time, real time position at which it was received. Color coding is used to distinguish between messages of various types, and the same menu and mouse interface used for the `fplot` program is used for `mplot`.

Instead of logging events, the message plot data gathering feature in TWOS logs messages. Like the event logging feature, messages are not logged by default. A tester command issued from the configuration file sets aside space for logging these messages. The basic logging function, `msglog_entry()`, is called in several places. It is called in `msgproc()`, to log system messages. It is called in `nq_input_message()` and `nq_output_message()` to log regular messages. And it is called in `gcpast()` to log a special dummy message used to make GVT entries in the message log. The mechanism for transporting the log messages sent by `msglog_entry()` is similar to that used by `flowlog_entry()` to log events, and all subsequent code is a mirror image to that used by the flow log.

The code used for the event log and the message log, while very similar, is completely separate. In principle, both logs could be used for a single run. In practice, doing so would use up far too much space to be of much practical value.

24.5 The Migration Log

The migration log is used as a debugging and performance monitoring tool to examine dynamic load management. Since it produces a relatively modest amount of data, it is written for all TWOS runs. A message is produced by each node during each load management cycle. This message contains the node's identity, a unique identifier of this load management cycle (to ensure that all load messages from all nodes for a single cycle can be easily grouped for examination), and the effective utilization of the node during that cycle. (See Chapter 11 for an explanation of effective utilization.)

Additionally, a message is sent by any node initiating the migration of a process. This message contains the node number, the value of the local real time clock at the start of the migration, the load cycle identifier, the name of the process, the process' phase begin time, the node it is being sent to, the current SVT of the process (which is usually the same as its phase begin,

under the current temporal splitting policy), the number of states, input messages, and output messages being sent, and the current GVT at the start of the send. A matching message is sent to the migration log by the sending node once the migration completes. This message contains the sending node's identity, the local real time clock, the name of the process moving, its phase begin time, the receiving node, and the current GVT when the send completes. These two entries can be used to determine how long migrations take, what sort of processes are migrating, which nodes are involved, and how much data is moving.

The migration log can be used to produce data that drives a graphical dynamic load management playback tool, instead of its regular output. This graphical tool has limits on the number of nodes it can support, and it has at least one bug that causes a crash, but it can sometimes provide insight into the workings of dynamic load management. It sends data to the migration log at the same times as the normal logging mechanism, but the messages contain different data, including a tag to make identification of the type of message easier. Also, when the graphics program is being driven, TWOS must send data to the log whenever an object is created, and whenever a process is split. Other than the format of the data, its handling is the same as for normal migration logging.

Whenever a migration log entry of any type must be made, `send_to_IH()` is called. This function sends a message to the CP. That node calls `migr_log_msg()` upon receipt of the message, which causes the CP to open the log file (if it hasn't already been opened), perform `HOST_fputs()` to write it, and `HOST_fflush()` to make sure the data is flushed immediately to disk. Unlike some other logging functions, the migration log writes its data to the actual disk file as soon as it can, permitting it to be examined during the run.

Appendix A: The Sequential Simulator's Internals

By John Wedel

The sequential simulator (the simulator) is a fully sequential program which uses the same application modules as are used by Time Warp itself but operates sequentially without roll back. It runs entirely on one node. It is supposed to produce the same statistics as Time Warp insofar as the statistics are pertinant to a sequential run, that is, the number of events, the number of committed messages, and some other statistics. One of the purposes of the simulator is to debug programs which may not be running properly or may cause a crash when run with Time Warp itself. For this reason the simulator contains a number of options specified on the command line and a number of internal facilities for debugging. These are described in the Users Manual.

The simulator is contained in a library which is linked with the application modules to form a program. All Time Warp entrypoints are included in this library and are identical in name and arguments (if any) to the entrypoints in Time Warp itself. The simulator is compatible with the Time Warp Users Library. There are two simulator libraries called twsim.a and twsiml.a respectively. The latter contains all of the former plus the users library.

A simulator application is a single process running on one node and is fully sequential with only a single message queue. No operations are performed in parallel as they would be in Time Warp itself. Most of the data structures in the simulator are implemented as arrays which are sized at compile time. This is not, however, true for the message queue which is a linked list. The queue is discussed later in this document.

If the simulator is started with the -q option it will display its banner on the console and await commands. A list of these commands which are designed for debugging aids can be obtained by entering an H (for help). This initial stop is dependent on the -q option which sets the step flag. The step flag is read before each event starts and causes a stop and request for input at the beginning of every event until the flag is turned off.. The step flag is normally turned on by ST and off by SF.

The simulator next reads a configuration file which is specified on the command line or defaults to a file named `cfg'. The configuration file reader is a yacc program using a lex program to define its tokens. This is different from Time Warp which uses a c-program to read the file. Note that yacc and lex are processors which produce C-files that are compiled by the C-compiler. The simulator file reader will check the syntax of all commands input including checking the types of the arguments. If it finds a configuration command it does not recognize it will issue a warning and ignore the command. If it reads a known command its behavior depends on the specific

command. If it is a Time Warp command which is not applicable to the simulator it will check the syntax and issue a warning if it is wrong. If, however, the command is a simulator command and the syntax is incorrect an error flag will be set and the simulator will exit after it finishes processing the file. Continuing to the end of the file may show more errors in the file.

The configuration file reader sets up the data structures for the defined object types and objects (object body). the object body array entry is the equivalent of the OCB in Time Warp. It contains a pointer into the object type array. This array in turn has entries pointing to the code for the init, event, and term sections of the object type. The object type array is accessed frequently and its search routine uses a binary search.

The configuration file should contain a command to schedule the first event. This command will create the only input queue and place the initial event message in it. The order of configuration commands in the file is important for both the simulator and Time Warp. For example, the getfile command is processed and reads in the file and this data might be used by the application programmer in a message scheduled for the `init' section of an object in the application. Thus the getfile must be processed before the object is created by an obcreate configuration command. The obcreate command creates the object and also runs its init section. When the simulator finishes the configuration file it will initialize some timing variables and attempt to read the input queue thus starting the simulation. Thus the implementation is such that timing statistics do not include the configuration operations nor the time for init sections of objects.

When the simulator has exhausted the event queue it processes the term sections of all of the objects, does some clean up work, displays some statistics and exits. If the -q option was used on the command line, additional statistics are displayed. The simulator measures the time talen by the various events and keeps timing statistics. These statistics are of little value on the Sun machine version because of the very low resolution of the system clock. The overall time if it is several minutes or greater is of reasonable accuracy. On the BBN Butterfly machine, the clock resolution is much better and the data on individual objects is useful. The overall timing statistic is also derived at the time that the program finishes but before the statistics file is written. After every event the simulator checks an error flag and will terminate if this flag has been set.

The simulator has a number of options which can be specified on the command line. They are listed in the users manual. Some of these options may slow down the simulator. Options which write to files have this effect except that special attention is given to the option(default on) which makes the `XL_STATS` file and the option for the `SIMDIR` file so that they do not actually write the files until the clean up at the end of the program. This

procedure is used so gathering the data only uses memory and has a minimal effect on the timing of the run. The options which produce the TRACE file and the message files output the data for each event to the file at the time the event is processed and thereby affect performance.

The simulator responds to control-C and stops for instructions before starting the next event. The implementation uses a Unix signal to stop the program. The signal handler simply sets the step flag to true and continues the program. Therefore if an event is in an infinite loop control-C will not stop because the step flag will not be read until the next event is about to start. The Unix quit command (control-backslash) will always stop the program but will dump core unless this has been disabled (in the operating system). A list of the commands to the simulator when it has been stopped by the step flag during execution can be obtained by typing `H' as previously noted. This list describes the breakpoint insertion and removal commands which also are acted upon just before starting events.

Because of the single process nature of the simulator the Time Warp commands for creating and processing files set up the appropriate environment and then call the appropriate UNIX library functions. The action of the simulator for reading files is the same as Time Warp. The entire file is read into memory during the initial `getfile' configuration command so that any object can use the file with its own local character pointer.

During operation of the simulator a number of error conditions may be detected which will result in warning messages. A number of other more serious error conditions will result in error messages and the simulator will exit immediately. Because the simulator is a single process it can be run under control of a debugger such as dbvtool or gdb. See also the next paragraph about internal debugging commands.

One of the help commands which can be used is the `debug on' command(DBT) This will cause most of the Time Warp entry points to print out data regarding the arguments with which they are called and then to pause with the name of the entry point itself. Use of this switch will evidently cause the simulator timing to be useless. The operator must also continue the simulator run after each debugging pause by typing a carriage return. Use of breakpoints to stop the simulator before an event with a known error and then using the debug switch to run through the Time Warp entry points in the event is advised.

The implementation of the memory allocation commands `newBlockPtr` and `newBockWithPtrs` has included a feature useful for debugging. Recall that `newBlockPtr` in Time Warp results in an index number which must be passed through `pointerPtr` to obtain a genuine C-pointer. This is also the case in the simulator although the `newBlockPtr` could return a genuine C-

pointer itself. Therefore failure to call `pointerPtr` will likely result in a crash in the simulator as it will in Time Warp.

The linked list message queue uses message blocks obtained from the operating system as needed. On the Sun these are obtained from the malloc system call. On the Butterfly, the page map command is used to map an amount of memory determined at compile time and then a routine which is similar to malloc is used to set up and maintain a free list.

The simulator does not release message blocks but maintains its own free list and reuses them. Message blocks are added to the front of the free list when released and taken from the end of the list when needed. A debugging command is available(DM DEL) which reads the free list. This command will read all the blocks which have been released and not yet used again thus making it more useful.

The simulator uses a single queue for the application. Events are inserted into this queue by the schedule entry point and removed when the event is processed. The queue is a splay tree. Early implementations of the simulator used an implementation with a linked list or a heap for the queue. Tests indicated that about a 10 percent speedup could be obtained with the splay tree which was therefore adopted as standard. Some applications with very short maximum queue lengths would, of course, run more rapidly with a linked list.

The splay tree brings the last accessed item to the root of the tree. This is a questionable procedure for this type of queue because all removals are at the leftmost node whereas insertions (messages sent) are never there unless the message is to be received at time `now'. Different queue accessing strategies have been proposed but were not implemented in this project.

Messages sent to time `now' are not legal in Time Warp but can be sent by special options for backward compatibility. Time Warp will handle such messages properly unless they cause an infinite loop but the simulator does not have rollback and may give different statistics for programs using this technique. This occurs because insertion of a message for a given object by other objects running at the same time may cause the object to run twice if its event has already been processed for this specific time. Messages at time `now' are, however, legal for messages sent to the stdout object. The Time Warp routine tw_printf sends messages to the stdout object at time `now'.

If the flag for allowing messages at time `now' is set by using the configuration file command 'allownow', the simulator implementation may give different statistics from Time Warp if objects are created or destroyed. The simulator does not send messages at all for such objects if the action is to take place at time `now' but creates or destroys the object immediately. Therefore a different statistic for number of messages will result.

The sorting algorithm for inserting messages which have common subfields is the same in the simulator as it is in Time Warp. Therefore messages are processed in the same order. This procedure is used to make the statistics agree. Note however the there is an anomaly in the behavior. The global event message count given in the `SIMDIR` file includes the initial messages sent in the configuration file. The `XL_STATS` file does not include these messsages.

The simulator contains a special object for the standard output (stdout). It sends the output directly to the console (or file if redirected) immediately. Stdout is implemented as a C-module in the simulator library. If desired the user can write his own C-program to replace this module and link it in ahead of the library so it replaces the library module. The simulator always creates stdout even if it is not created by an obcreate in the configuration file. This action is implemented in the configuration file reader if it comes to the end of the file and has not encountered an obcreate.

The internal functions of the simulator are distributed among several C source files. The contents of these files are described below. To speed up the program it also uses some routines written in assembly language. These routines are identical to the same named routine in timewarp itself.

`twspl.c`: This files contains the "main" routine and generally includes routines that are directly used by the simulator internally. Many routines specific to certain major parts of the simulator are in other files with more descriptive names. The initial routine `main()` initializes certain variables and arrays and then calls `main_process()`. The simulator loops in this routine as long as messages are found in the queue and there are no fatal errors. When the queue is exhausted the simulator calls `term_process()` to perform the actions scheduled by the term section, It then prints some data depending on the values of some command line options and exits. `dis_times()` and `dis_states()` accomplish the printing. The routine `find_hog()` calculates the event taking the most time. As stated, time values are not useful on the Sun because of the low resolution of the system clock. The routine `pr_setup_pathtime()` is used ONLY if the critical path calculation option is turned on. It finds the 'clock' time the event terminated by taking the maximum of the termination time for the object and the time each message which started the event occurred. The other routines whose names begin with `pr_` are self explanatory. Note that the lines in the file which follow the comment 'run the event section' are where the event section is called by calling a routine through a pointer to its code. This pointer is stored in the array `process[]` which is indexed by type of object. This array is setup by the configuration file.

`twsp2.c`: This file contains the routines that constitute the interface with the application and a few accessory routines. All application calls to timewarp have a routine defined in this file.

`twsp3.c`: The routines that comprise the IO-interface and the memory allocation routines used by the application are in this file.

The routines with names starting with "`io_`" are used to process the array `fileary[]` which contains an entry for each file that is accessed (read, written, or created) by the configuration file commands `getfile` etc. This array is also accessed by the timewarp file routines such as `tw_fopen()` for processing the file itself. The routines which read data from the file or write data to it do so through a streams structure in each object so that individual pointers to characters in the file can be maintained.;

The memory allocation functions (such as `newBlockPtr()` and `disposeBlockPtr()`) are also in this file as well as the bookkeeping routines for them. These routines use indexes into an array for the pointer values returned rather than returning C-pointers directly. This is in agreement with Time Warp and makes it likely that an attempt to the use the index directly rather than call `pointerPtr()` to obtain a valid C-pointer will crash the simulator as it would Time Warp itself. The index array is created when the first `newBlockPtr()` is issued and expands dynamically if necessary. The expansion occurs by getting new memory from the operating system, copying the current array values to the expanded array, and then freeing the memory for the old array. The array is never contracted if memory is released.

`twqueues.c`: This file contains the routines for handling the message queue. It also contains the routines used by the simulator to call the queue specific functions. These are `find_next_event()`, and `cm_queue_event_message()` which are self-explanatory. The simulator calls `Pr_setup_messages()` and `pr_delete messages()` just before the beginning of an event and at the end of the event to read the messages queued for the event into an array the event can process and to delete the messages from the queue respectively.

`twhelp.c`: This files contains various error handling routines and processing routines for user input to the simulator at runtime. It also includes the `help()` routine which prints out a message on the screen when the user types 'help' at a simulator prompt.

`vtime.c`: This file is common to timewarp and the simulator and contains routines for processing the virtual time structures.

`simlex.l`: This is a source file to be processed by the lex program. It defines the tokens used in the configuration files. Lex creates the file "`simlex.c`" which is used by the simulator.

`simpar.y`: This is a source file processed by yacc which contains the grammar for the configuration file commands. Yacc creates another file called `simpar.h` which contains symbols used by lex and also creates `simpar.c` which is used by the simulator.. The yacc parser is not recursive and only one can normally be used in a program. The CTLS prgoram also uses yacc. Thus part of the `Makefile` for making the simulator renames all the global variables used by yacc to avoid conflicts. Rename is accomplished by the Unix sed program.

`simmem.c`: This file simply calls `malloc()` for memory allocation when used on the Sun. On the Butterfly, it maps a block of memory, touches every page to get it into the memory and then acts to emulate the malloc routine with the memory obtained. If it runs out of memory from the original memory map, more memory will be obtained and a warning issued. The routine `sim_free()` will free memory managed by the simulator but will not release it to the operating system. To avoid conflict with system library routines the simulator programmer should use `sim_malloc()` and `sim_free()`. Note that malloc and free can NOT be used at all in Time Warp.

`faults.c`: This file is only used on the Butterfly and contains a routine which calls the operating system to find out how many page faults have occurred during the run.

`itimer.c`: This file contains the low level timing routines. They are specific to the platform on which the simulator is running. The `itimer()` routine accesses the operating system's clock

`newconf.c`: This file contains the routines called by the configuration file parser to actually do the work of setting up the necessary structures in the program.

The configuration file routines are defined in such a way that a self-contained program can be made which only reads configuration files and prints out the data. This is accomplished by defining the symbol `STANDALONE` in newconf.c before compiling. This action changes the name of `configure_simulation()`, a simulator routine, to `main()` to create a standalone program. The parser and token recognizer for the configuration file are in `simpar.y` and `simlex.l`.

The names of the routines in this file correspond to the names of configuration file routines. The routine `cr_typetable()` is called by `main()` and sets up the process array containing the entry point pointers for the

event, term, and init sections of the code for each type of object. The routine `cs_send_message()` actually creates and queues messages sent by the user. Such messages are tagged with the sender object `twscon`. This routine is called by `config_msg()` which results from a `schedule()` (formerly `tell()`) in the configuration file. The routine `configure_simulation()` is called by `main()` to start processing the configuration file. When that file is exhausted, the routine `sim_start()` is entered to create a `stdout` object and start the simulation.

`stdout.c`: This file contains a routine which acts as stdout in the simulator and sends messages to the console. As stated, it can be replaced by another stdout module written to output things to the users specification.

`cubeio.c`: This file is intended to be the same as in Time Warp and contains routiness which produce the `XL_STATS` file. Not all of these routines are used in the simulator. The routine `record_obj_stats()` is used in the simulator and serves to copy statistics from the simulator into the same structures that are used by Time Warp itself and are found in the Time Warp ocb structure.

`twsd.c`: This file contains the global variables used by the simulator. Recent modifications to the simulator have tended to use variables defined in the files containing the routines rather than global variables. These symbols can be accessed by extern declarations in other files that may need them.

`HOST-fileio.c` This file contains low level IO-routines which differ among the operating systems on which Time Warp runs.

`oldtw.c`: This files contains some routines used in previous versions of Time Warp, It is expected to become obsolete.

`tlib.c`: This file contains some library functions for the user which are not defined in all the platforms. The file is not needed on the Sun.

Appendix B: Benchmarking TWOS

This appendix describes the procedure of running a standard TWOS benchmark. (An example of the results of such a benchmark is contained in Appendix C.)

A standard TWOS benchmark is typically run for each new released version of TWOS. (No benchmark was run for TW 2.7 because it contained relatively few new features, and time was short.) The benchmark consists of running three simulations on varying numbers of nodes. The three simulations are warpnet, STB88, and pucks. Each simulation is run on 72, 68, 64, 60, 56, 52, 48, 40, 36, 32, 28, 24, 20, 16, 12, 10, 8, 6, 4, and 3 nodes Also, Warpnet is run on 2 nodes. (The other two applications typically cannot execute on 2 Butterfly nodes.) For each benchmark, for each number of nodes, four separate runs must be made, to investigate variation in the new system's performance and to ensure that the reported numbers are representative of the true performance. (The four run per point figure is actually only a guess of the proper number of runs to obtain statistical accuracy.)

The first step in running a benchmark is to make a correct version of both TWOS and the sequential simulator. These should be stored in a directory structure set up for this particular version of Time Warp. Next, the three applications should be linked with both TWOS and the sequential simulator, producing a total of six load modules. These load modules are usually titled `warpnet`, `warpnettw`, `stb88`, `stb88tw`, `pucks`, and `puckstw`. The load modules whose names end in tw are the TWOS versions of the applications, while the others are the sequential simulator versions. The makefiles stored in the BF_MACH subdirectories of directories for the applications will create these load modules. Care must be taken to first edit the makefiles to ensure that they are loading the most recent version of TWOS and the sequential simulator, however.

Typically, Unix shell command files are used to run the benchmark suite. One is created for each application, containing one command line for each one of the runs that must be made. These lines should use the -S switch to ensure that every individual XL_STATS file is saved (see the TWOS User's Manual, section 2.3.2.1). In most cases, these command files can be left running overnight, or even invoked in a batch mode by the Unix `at` command. (Note that, if the at command is used, the user should not give the command file name to at, directly, but, rather, the name of a second command file that in turn calls the command file containing all of the TWOS commands. Otherwise, the Unix `cron` demon will object to the length of the line submitted to `at` and will fail, more or less silently.) Care should be taken that no other activity is going on while the benchmark suite is running. Past experience on the GP1000 has shown that other activities on the machine can corrupt TWOS timings, even if TWOS is given its own cluster to work in.

Other platforms may permit some of their nodes to run TWOS independently while other nodes do other work, without impacting the TWOS timings, but testers should verify that this is the case, not just assume it.

If the benchmarker is not watching all of the runs at all times, he might want to use the configuration file batch command to allow TWOS to abandon any runs that trap to tester and proceed on with the command file. Otherwise, a single failed run early in the command file may prevent that file from producing much useful data. See section 2.2.3 of the TWOS User's Manual for details on the batch command. Be careful to remove the batch command from configuration files before making runs for debugging purposes, as any runs made with the batch command will bypass tester and merely exit upon detecting any problem.

When all runs have completed, the user must check them to ensure that all of them were correct. This check is performed with the check/measure program, described in the TWOS User's Manual, section 7.1. The measure version of this command should be used to produce measurements files containing one summary line for each run made. A separate measurements file should be produced for each of the three simulations. Once the measurements files have been produced, the benchmarker should examine each of them to ensure that all runs produced proper results. The correct number of committed events and committed messages can be found in the TW 2.6 benchmark included in Appendix C of this manual. Any runs producing incorrect results must be done again, until they produce correct results. If more than a handful of all runs in a benchmark produce incorrect results, then the version of TWOS being benchmarked should be regarded as being insufficiently bug-free to benchmark.

The measurements files contain 4 lines for each application for each number of nodes. The graphs to be plotted need these four lines to be combined into a single line. The collapse program, described in Section 7.1 of the TWOS User's Manual, will automatically do the averaging and combination of these lines. This program must be run once for each of the three measurements files.

Once the TWOS runs are complete, the user should make one sequential simulator run for each application to obtain a timing to use for speedup comparisons. For complete safety, the sequential simulator runs should be made when the parallel machine has no other activity, even though other activities should not impact the sequential run times. The benchmarker should use the -s switch with these runs to produce an `XL_STATS` file from each simulator run. When the run completes, the benchmarker should take care to rename these `XL_STATS` files to something that will uniquely identify them as part of this benchmark.

The benchmark curves are typically plotted using the Excel spreadsheet program on a Macintosh, but the benchmarker should feel free to use whatever plotting software is most convenient. The collapsed measurements files are set up with a tab between each field, and with a line of header information first, which meshes well with Excel.

For each application, four curves must be produced. The first is the timing curve, which plots run time versus number of nodes used. This information is directly available from the measurements file. The second curve plots speedup, which is obtained by dividing the run time for the sequential simulator by the run times in the measurements files. The third curve is the ERBO curve (events rolled back over), which is calculated by subtracting the number of events committed from the number of events completed. Both of these values are available from the measurements file. The final curve plots the number of process migrations performed for each set of runs. These numbers are also directly available from the measurements file. In addition, the spreadsheets containing the collapsed versions of the measurement files should be printed and included in the resulting benchmark document.

Appendix C contains examples of what each curve has looked like in past benchmark memos. The benchmarker should feel free to use his own favorite format, but should strive to ensure that all information in the format previously used is also available from his preferred format. Appendix C also contains a sample of the outline of a benchmark memo. Again, future benchmarkers should feel free to adapt this format to their own needs.

All data from the benchmark should be moved onto secondary storage. This data includes all statistics files produced by the individual runs, the regular and collapsed measurements files, and the statistics files produced by the sequential simulator runs. In the past, these data files have been saved on Macintosh floppy disks, but magnetic tape might be a more appropriate medium. The benchmarker should also ensure that a tape version of the entire release directory of the benchmarked version of the software is preserved somewhere safe.

new version of TW 2.6 taking 6-9 seconds longer, between a 5% and 10% increase in run time.

The maximum speedup obtained for Warpnet under TW 2.6 was 27.00, on 72 nodes.

The timing chart and ERBO chart for Warpnet under TW 2.6 show no effects worthy of note. The migration chart does not show as strongly the odd shape of the chart for TW 2.5, in which migrations fell off as more nodes were added. But there is still a flattening of the number of migrations at higher numbers of nodes, and a dropoff at the largest number of nodes. Further tuning of dynamic load management might be in order. It's also possible that the slight decrease in performance of Warpnet under TW2.6 is related to improper migration decisions having a greater effect as more of them are performed. Work is underway to improve dynamic load management for these cases.

C4. STB88 Results

STB88 fairly normal performance, with one exception. At eight nodes, the run time for STB88 shoots up dramatically. (The difference is more visible on the timing chart than the speedup chart.) The ERBO chart shows that the 8 node runs had almost four times as many events rolled back as the 6 node or 10 node runs. There were also more migrations for the 8 node run than its neighbors, though the total number of migrations is so small (twenty, at 8 nodes) that putting the blame entirely on poor migration decisions seems questionable. Examining the four runs that comprise the 8 node point, three of them were actually longer than the average run time, with a single, somewhat shorter run lowering the average by 15 seconds or so. Therefore, the poor eight node performance cannot be blamed on a single run that performed very, very poorly.

As might be expected, most of the statistics for the 8 node point are higher than those for its neighbors. For instance, nearly 700,000 positive messages were sent, instead of a bit over 600,000. One statistic particularly stands out, however – the number of events started jumped from 435,000 (for 6 nodes) to 645,000. At ten nodes, it dropped back to 470,000. This statistic is indirectly shown in the ERBO chart.

Another dramatic statistical difference is in the number of tells. (Actually, nowadays, the number of schedules.) This number jumped from 655,000 to 948,000, a difference of nearly 300,000. However, fewer than 100,000 more positive forward messages were actually sent. Apparently, lazy cancellation saved the resending of almost 200,000 messages, in this run. It is unclear whether this was a good thing or a bad thing. If the messages in question were eventually committed, it was a good thing. If they were not, it was a very bad thing, and could explain the performance discrepancies.

Further investigation of this anomaly is warranted. Tests with dynamic load management turned off, and tests with aggressive cancellation turned on, would be good starting points. The former would indicate whether the difficulty has to do with a poor configuration that dynamic load management has trouble correcting. The latter would indicate whether lazy cancellation is causing a problem, here. It is interesting to note that the same configuration did not cause this problem in TW 2.5, but it is hard to say what changes in TW 2.6 could have caused the change in performance.

Apart from this anomaly, the performance of STB88 under TW 2.6 is pretty steady. The speedup curve trends upwards almost linearly. There is another interesting phenomenon observable in the migrations chart. At even multiples of 16 nodes, the number of migrations jumps. These jumps aside, the migration chart is fairly smooth. This data suggests that there is something about round robin assignments on multiples of 16 nodes for STB88 that causes greater observed imbalances in load. Perhaps the order of grid definition in the configuration file leads to poor clustering of busy vs. idle grid objects when 16 node multiples are used. However, we see no corresponding jump in the run times of configurations for these cases, so apparently dynamic load management is successfully handling any imbalances.

Compared to the performance of STB88 under TW 2.5, TW 2.6 is mostly an improvement. The eight node point is much worse, and a few other configurations run a little slower, but most configurations run a little faster. The major exception is that, between 56 and 68 nodes, TW 2.5 provided somewhat faster run times. However, the 72 node configuration runs faster under TW 2.6, giving a maximum speedup of 27.44, versus 25.56 under TW 2.5. Part of the improvement is due to a longer sequential simulator time under TW 2.6, but the Time Warp run time also improved 5%.

C5. Pucks Results

Pucks has a very regular set of performance curves under TW 2.6. The only curve that is not regular is the migration curve. TW 2.6 performs few or no migrations for small numbers of nodes. At large numbers of nodes, however, many migrations are performed, far more than were performed under TW 2.5. At large numbers of nodes, Pucks is inherently unbalanced, as it has too little parallelism to make good use of many nodes. Many migrations are performed in a fairly futile attempt to make use of nodes for which there is simply no work available.

While these migrations seem to do little good, they also do little harm. Pucks gives similar performance under TW 2.6 as under TW 2.5. Most run times under TW 2.6 are one to five seconds slower than under TW 2.5. Because of an increase in the run time of the sequential simulator, the maximum

speedup of Pucks under TW 2.6 is actually better than under TW 2.5, going up from 12.8 to 13.1.

One exception is the three node point. Under TW 2.5, a three node Pucks run took 1977 seconds. Under TW 2.6, it takes 1472 seconds. An examination of the number of message sendbacks shows the reason for the decrease. Under TW 2.5, nearly 39,000 messages were sent back for three node runs. Under TW 2.6, only 5 messages were sent back. The other two applications performed relatively few sendbacks at low numbers of nodes, even under TW 2.5, so there was no significant improvement in their performance for those configurations. This result indicates that our attempts to improve the memory situation have worked, to some extent.

C6. Comparisons With Earlier Benchmarks

Generally, but not universally, TW 2.6 runs a bit slower than TW 2.5. The speedup figures conceal this slowdown, since the sequential simulator seems to have slowed even more, but direct timing comparisons reveal it. In a few cases, TW 2.6 runs faster.

The most probable culprit for the loss of speed is changes in dynamic load management parameters that permit more migrations. These changes will be examined before TW 2.7 is released, and possibly further tuning will be performed.

C7. Conclusions

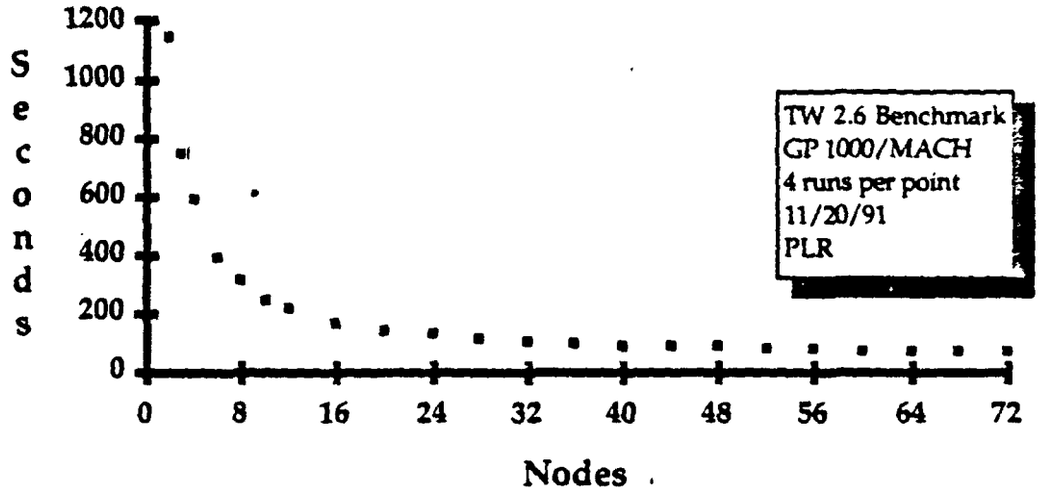
Despite minor differences in performance, TW 2.6 is characteristically similar to TW 2.5. In the single case where a memory shortage was causing a problem under TW 2.5, TW 2.6 seems to have solved it.

C8 Charts and Raw Data

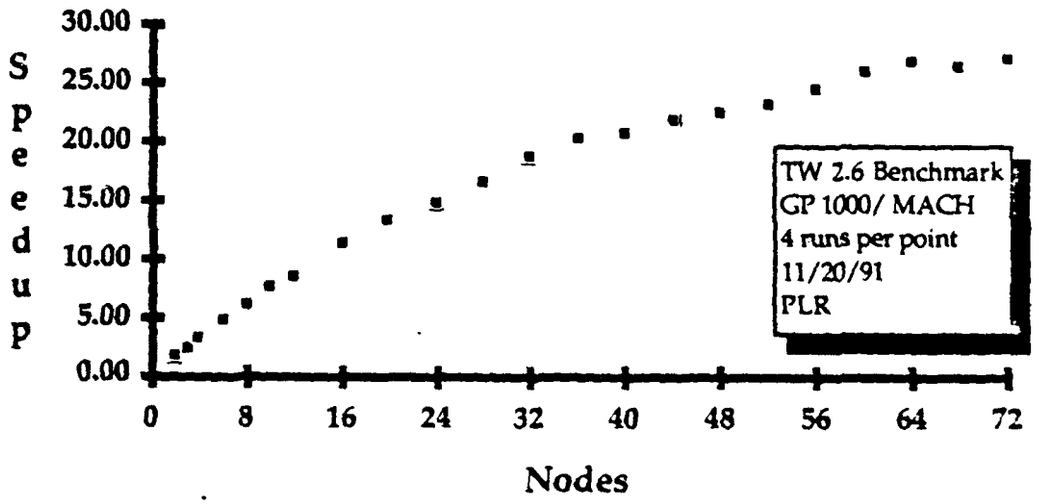
The following pages contain the charts mentioned in earlier sections, as well as worksheets containing the data used to make the charts, along with much other data. The worksheets are the collapsed versions, with the four runs per point averaged into a single line of data.

Warpnet Benchmark Data

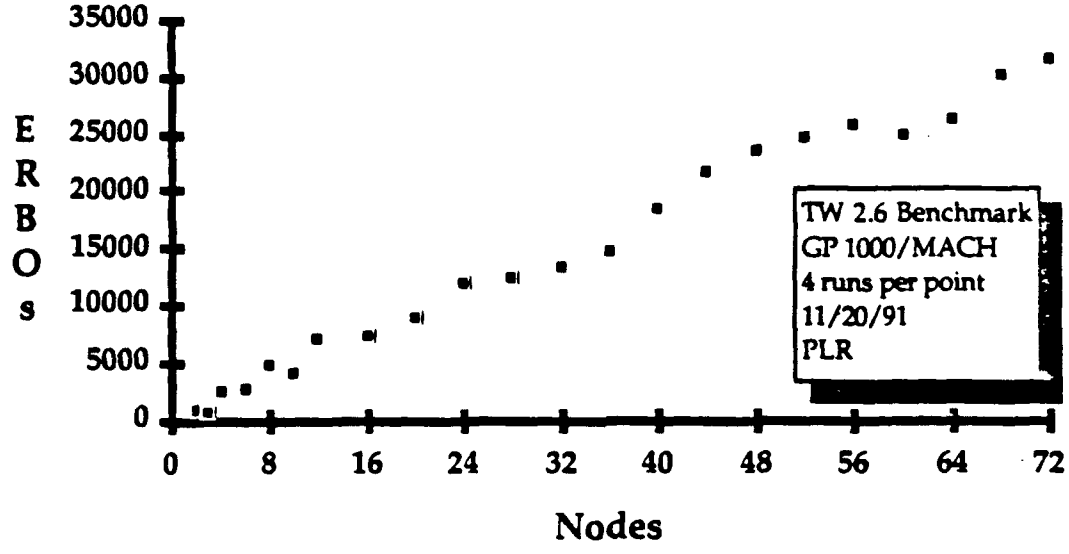
Warpnet Timing Chart



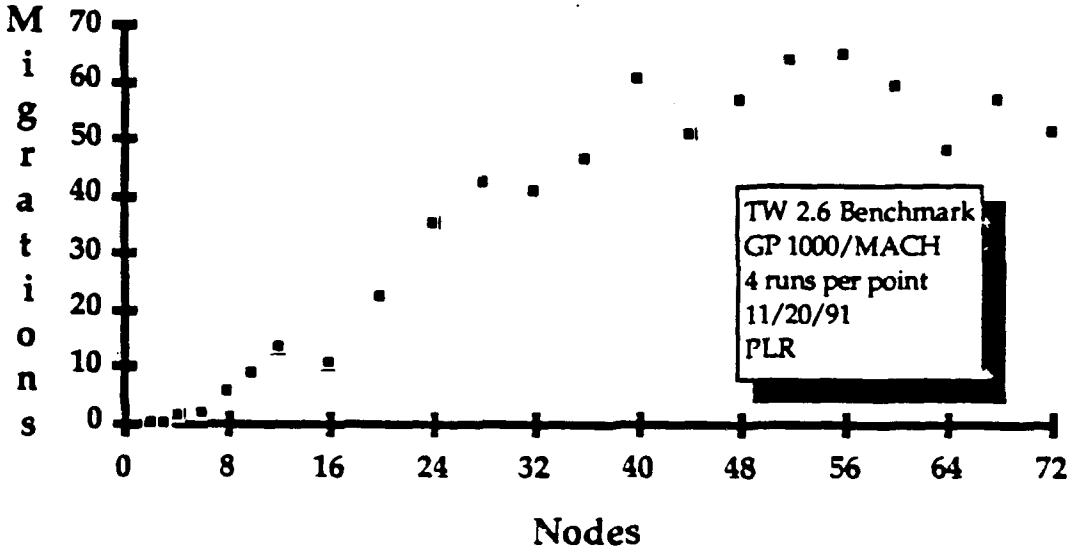
Warpnet Speedup Chart



Warpnet ERBOs Chart

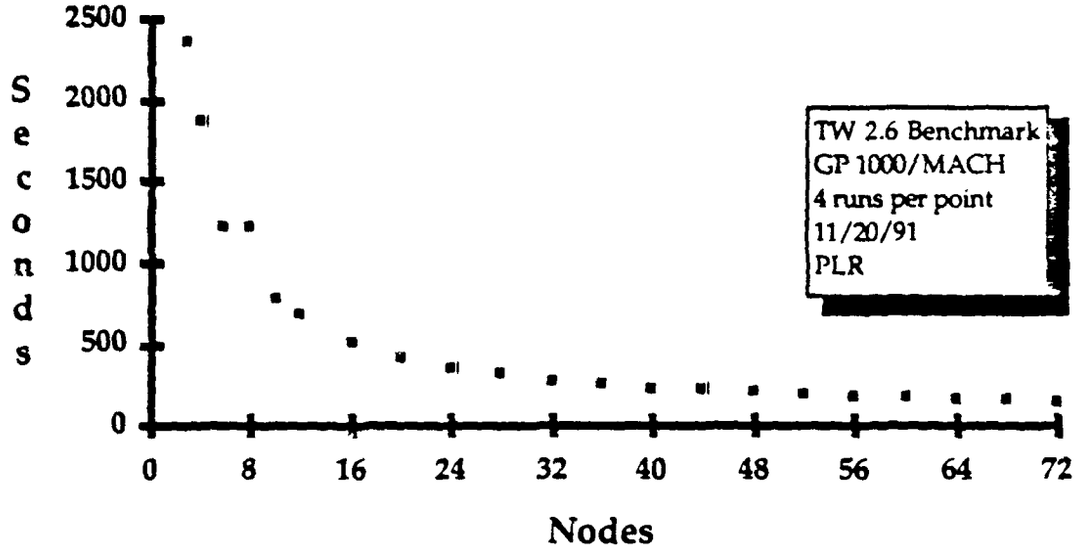


Warpnet Migration Chart

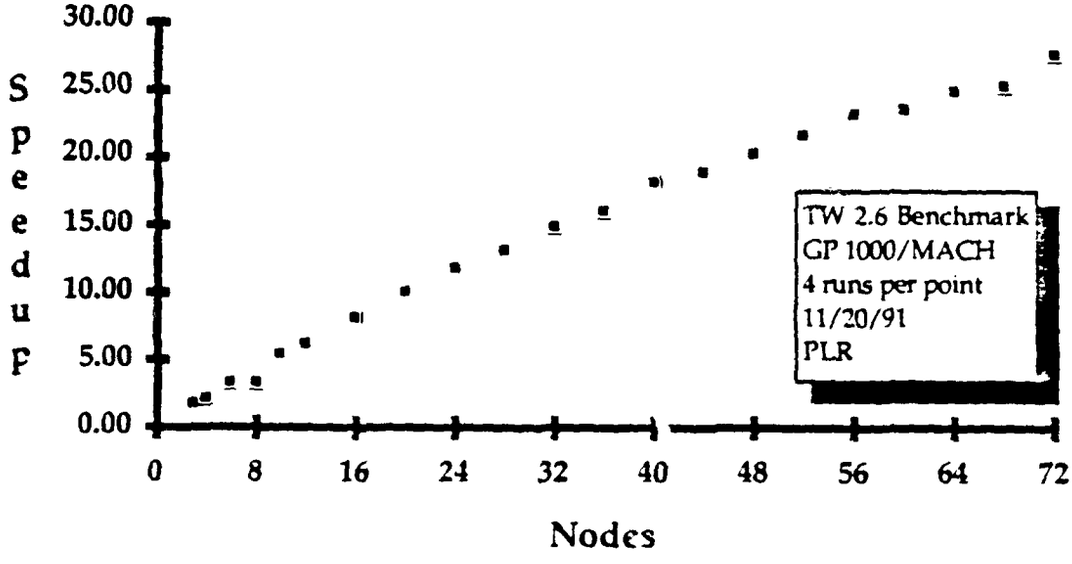


STB88 Benchmark Data

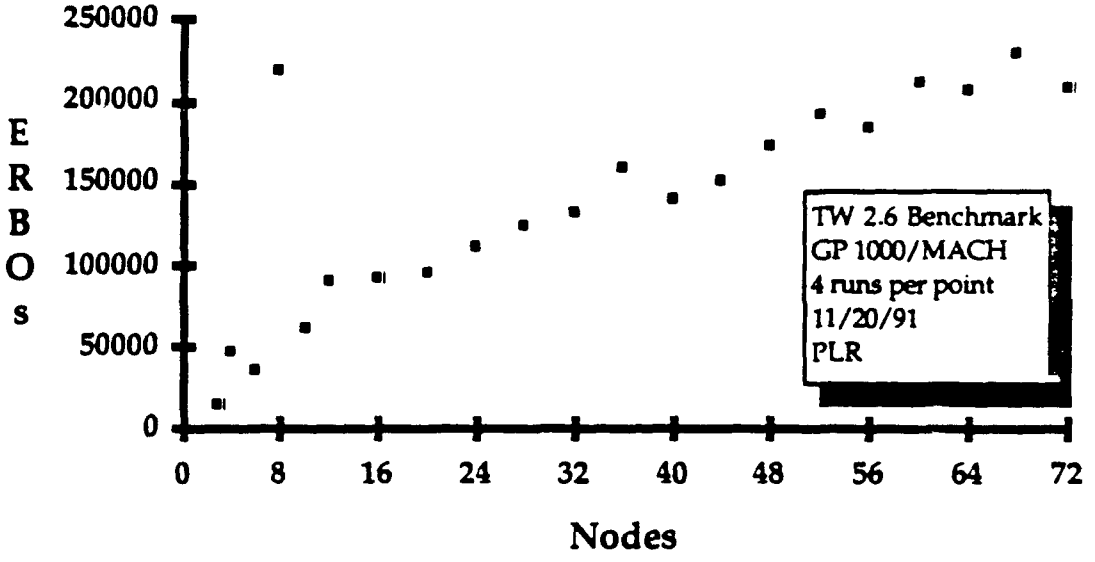
STB88 Timing Chart



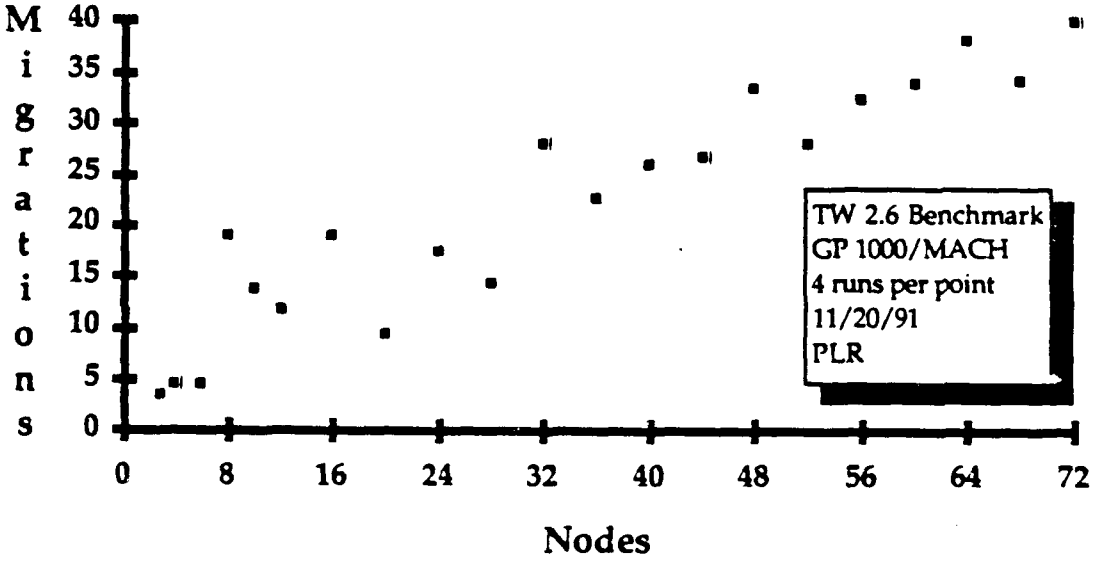
STB88 Speedup Chart



STB88 ERBO Chart

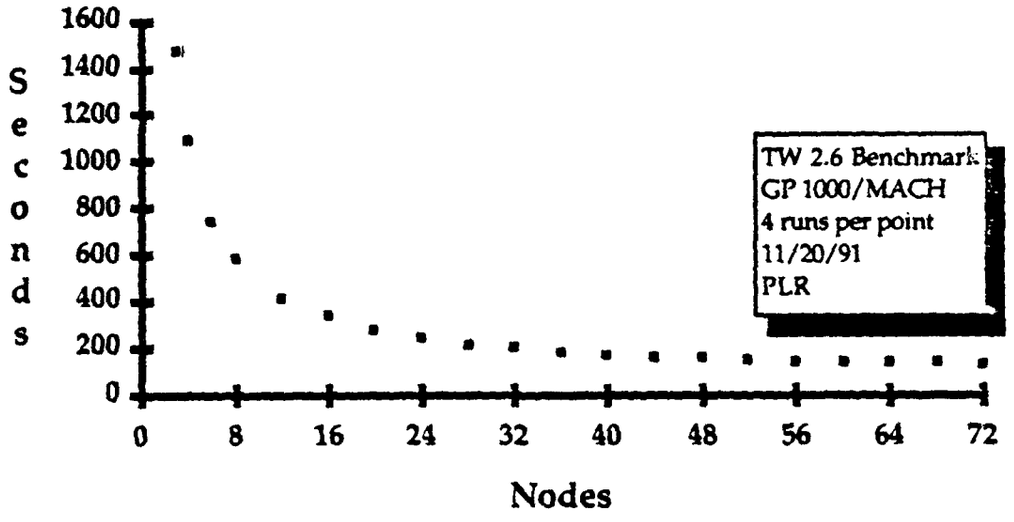


STB88 Migrations Chart

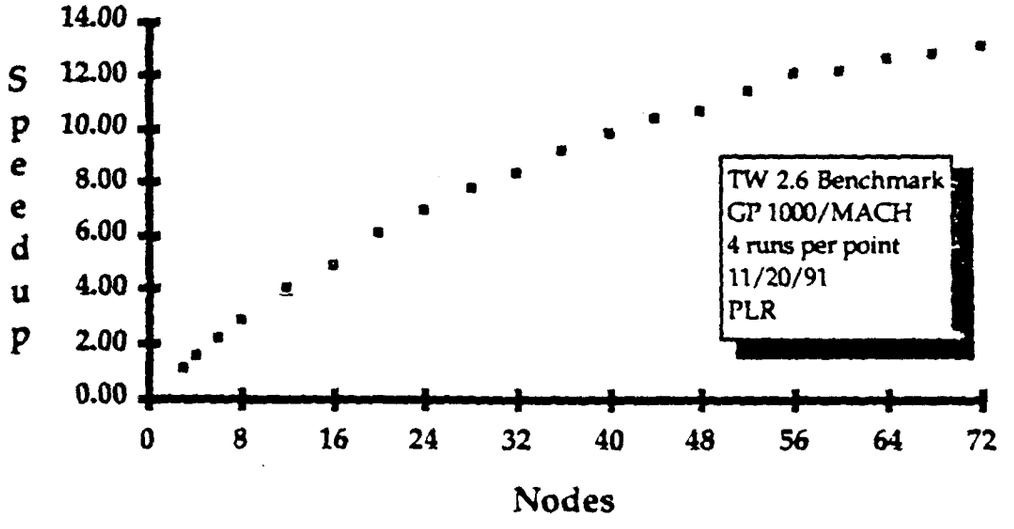


Pucks Benchmark Data

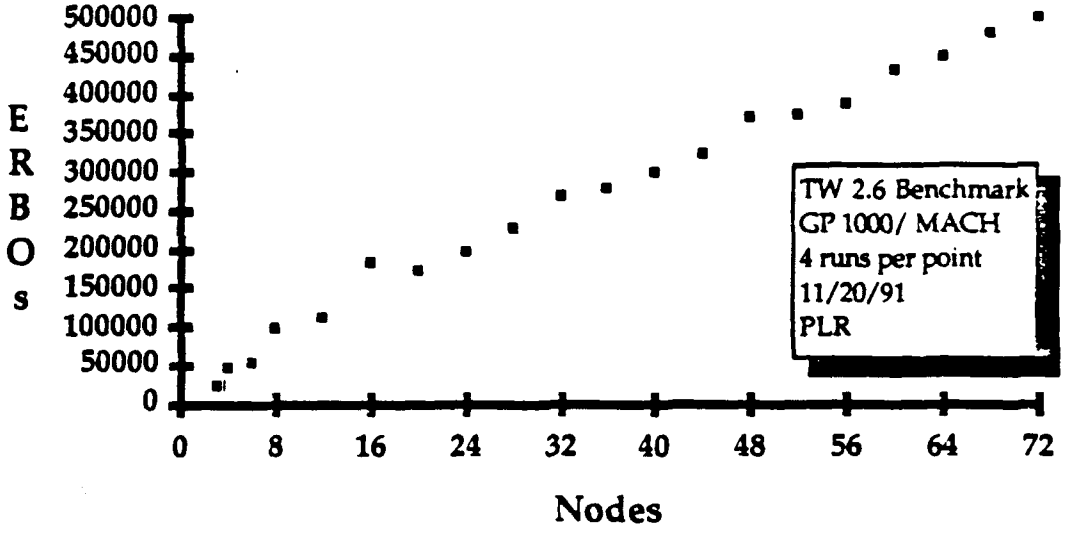
Pucks Timing Chart



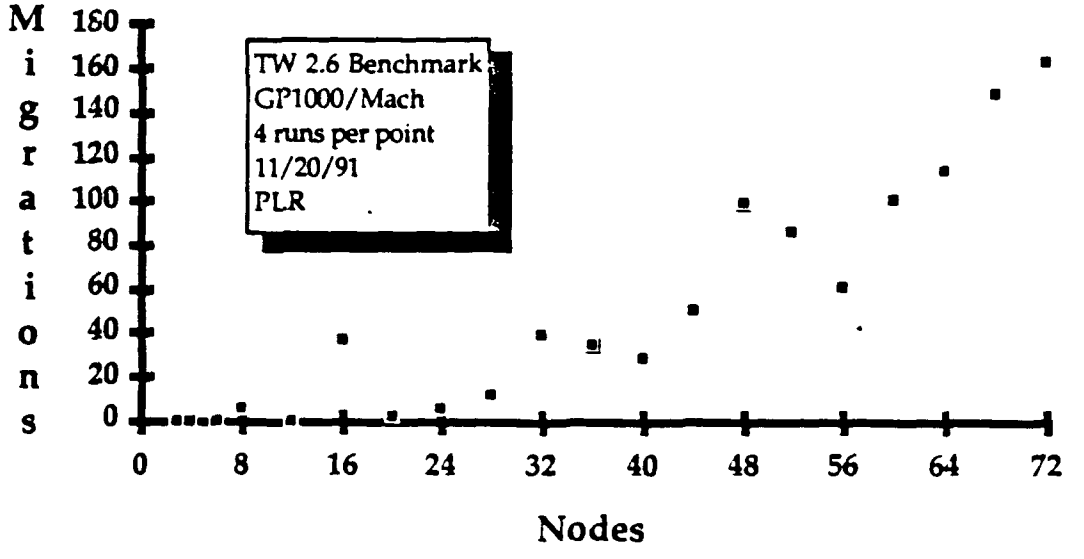
Pucks Speedup Chart



Pucks ERBOs Chart



Pucks Migrations Chart



Appendix D: TWOS Overhead Times

D1. Introduction

Memos PLR: 363-88-31 and PLR: 363-88-32 contained measurements of certain overheads for Time Warp for version 1.10. Since those memos came out, TWOS has changed considerably, as has the platform TWOS runs on. Those memos were based on data for TWOS running on the Mark3 Hypercube under Mercury, rather than the GP1000 under Mach. Also, TWOS has changed internally in many ways that might affect these overheads. This memo presents up to date overhead measurements for TW 2.5.1 running under Mach on the GP1000.

These measurements were gathered using a modified form of TW 2.5.1, with the ping simulation. The version of TW 2.5.1 used has a few changes not in the baseline TW 2.5.1, but none of those changes should affect these runs. In all cases, the measurements are the average over 10,000 occurrences of whatever is being measured.

D2. State Saving Time

State saving time for TW 2.5.1 was first measured using a 4-byte state (which is a header plus one integer, only), then with progressively larger states. Table D.1 shows the results. The minimal state saving time was .26 milliseconds. Saving 1000 byte states took .49 milliseconds.

State saving thus takes around .26 milliseconds plus .023 milliseconds per hundred bytes. This is slightly faster than TW 1.10, both for the zero byte case and the incremental cost of 100 extra bytes of state. Changes in the way TWOS makes memory allocations and changes in the assembly language byte copying routine probably account for these differences.

State Size (in bytes)	State Save Time (in milliseconds) TW 2.5.1	State Save Time (in milliseconds) TW 1.10
4	.26	.27
100	.28	.29
200	.30	.32
300	.33	.35
400	.35	.37
500	.37	.40
1000	.49	.53

Table D.1: State Saving Times

D3. Message Sending Overheads

Message sending latency was measured in two ways. The first way was to measure the time between when a message was requested by the user until the time that it was enqueued at the receiving process. The second way was to measure from the time of the request to the time at which the resulting event was started. This second way corresponds to the measurements made for TW 1.10.

It takes .97 milliseconds to get a zero byte message into the input queue of a receiver located on the same node as the sender under TWOS 2.5.1, and approximately 2.3 milliseconds to get it to the input queue of a receiver on a different node. (Provided the sending node is not node 0. It takes 2.6 milliseconds if the sender is node 0. This difference is probably due to delays caused by node 0's extra responsibilities in the GVT protocol.)

Note that these times are latencies, not entirely overhead. The sending node can perform work on other activities in less than 2.3 milliseconds, and the receiving node can do other work during the early part of that time. Given that transmission time on the GP1000 is negligible, almost all of this 2.3 milliseconds would represent overhead on some node, however.

The second method gives a delay of 3.76 milliseconds from the time a message is sent until the receiving process can start execution if the receiver is local, and a delay of 4.04 milliseconds if the receiver is remote. On TW 1.10, it took only 2.75 milliseconds for the same operation if it was local, and 3.71 milliseconds if it was remote. Of course, in the remote case a large component of the TW 1.10 latency was due to the time for physical

transmission of the message through the Hypercube, which is negligible in TW 2.5.1. Between TW 1.10 and TW 2.5.1, we have added a millisecond of delay (probably actual overhead) to the delivery of an on-node message. We should perhaps try to eliminate some of that overhead.

D4. Rollback Overhead

Rollback overhead is hard to measure, in general, for a number of reasons. Rollbacks do not take place in one continuously run piece of code, as state saves do, and rollback latency is not of much interest, unlike message passing latency. However, part of the rollback cost can be easily gathered, for a simple case. When TWOS delivers a message, it calls `rollback()` after queueing the message to set up the receiving process to run at the proper time. `rollback()`, despite its name, does not perform all activities associated with a rollback, but it does do many of them, including destroying states and cancelling messages. We can measure the overhead involved only in this routine with little difficulty.

In the case of ping, all rollbacks are trivial - they roll back from $+\infty$ to the time of the arriving message. There are no states to discard and no messages to cancel. The time necessary to perform such a minimal rollback is around .2 milliseconds. This overhead is part of the 3.76 millisecond overhead that is incurred to delivery a message on-node, as discussed in section 3.

D5. Per Event Overhead

Ping is well suited for measuring per-event overhead, when run on 1 node. TWOS running ping is never idle, so all time spent is either on user events or overhead. A one node run of ping for 10,000 events takes 41.74 seconds on 1 node, with 2.38 seconds being spent running user code. Thus, about 39 seconds are spent on overhead operations, yielding a per-event overhead of 3.9 milliseconds.

It's possible that the run would have finished up to one second sooner, due to timing of GVT calculations, but no longer. The difference between the value of GVT at the last tick before $+\infty$ and the virtual time of the last event was very small, so probably most of this final GVT interval was spent waiting for GVT to start. Therefore, the total time spent on overhead not counting waiting for the last GVT tick is closer to 38 seconds than 39, yielding a per-event overhead of 3.8 milliseconds.

This 3.8 millisecond per event figure matches closely the 3.76 millisecond one-node latency from message delivery to event start presented in section 3. TWOS running ping does very little other than deliver messages and run events. GVT calculations must take place, of course, but one might expect them to happen in between events, causing their effects to be factored into the 3.76 millisecond figure. Dynamic load management was turned off for these

measurements, so it contributes no overhead. The only other overhead costs not counted in the one-node latency are some startup costs and some context switching, which might account for the missing .04 milliseconds. Note also that the resolution of the Butterfly clock is 62.5 microseconds, so inaccuracies in the measurements could alone account for the missing time.

D6. Lazy Cancellation Overhead

All of the measurements produced so far have used TWOS with lazy cancellation. Aggressive cancellation is still an option in TWOS, but lazy cancellation is used by default. The ping benchmark cannot benefit from lazy cancellation, since none of its messages are ever cancelled. On the other hand, lazy cancellation should be very fast for ping, since the queue of messages to examine for lazy cancellation on either a rollback or a message send are quite short. Running ping with lazy cancellation can thus give us some idea of the overhead of lazy cancellation when it has no benefits.

Turning on aggressive cancellation decreases the run time of ping under TW 2.5 from 41 seconds to 39 seconds. Considering the run time of the processes (unchanged) and correcting for the possible lag in computing the final value of GVT (about 2/5 of a GVT cycle, or .4 seconds), the per-event overhead of TW 2.5 with aggressive cancellation is 3.55 milliseconds, a decrease of .25 milliseconds per event.

The time required to get a message from the sender to the input queue of the receiver, on node, decreased from .97 msec to .945 msec, almost the entire decrease due to aggressive cancellation. This measurement suggests that the routine that tries to determine if the message to be sent is already in the output queue is largely the cause of the extra delay in lazy cancellation, rather than extra costs in the rollback process when a process rolls back from positive infinity upon receipt of a new message. This routine is called `msgfind()`, so looking at ways to make that routine run faster would seem a promising way to reduce lazy cancellation overhead.

D7. Conclusions

Table D.2 shows the basic minimal TWOS overheads for TW 2.5.1 running under Mach on the BBN GP1000.

Operation	Overhead
Per Event Total Overhead	3.8 msec
Message latency (to queue)	
On node	.97 msec
Off node	2.3 msec
Message latency (to event)	
On node	3.76 msec
Off node	4.04 msec
State Saving	
0 byte state	.26 msec
Per 1000 bytes	+ .23 msec
Partial Rollback Overhead	.2 msec
Lazy Cancellation	.25 msec

Table D.2: TWOS Overhead Times Summary

Appendix E: TWOS Tools Internals

This appendix describes the internal details of some of the tools used in conjunction with TWOS. The appendix covers only the most frequently used tools, `check/measure`, and `collapse`. The internals of the configuration balancing tool, the instantaneous speedup tool, the graphic `fplot` and `mplot` tools, the memory analysis tool, and the graphic dynamic load management tool are not discussed here. Not only are they used less, but they are also changed less often, and are much more complicated.

E1. `check/measure` Internals

`check/measure`, whose use is described in the Time Warp User's Manual, Section 7.1, is meant to check the consistency of a single TWOS run, while at the same time optionally compressing a TWOS statistics files into a single line and saving it to a measurements file. The consistency checks are also made by reading the statistics file.

`check/measure` works in two phases. First, it opens the statistics file and reads it, line by line, accumulating various statistics in variables. Once the file has been completely read, `check/measure` moves on to its second phase, in which it compares some of these statistics to each other to check for consistency and optionally formats a line of output for the measurements file.

The first phase is complicated by the fact that the measurements file contains many types of lines. Chapter 22 described these many different types. `check/measure` must determine what type each line of the measurement file is so that the proper statistics can be extracted. The program loops repetitively, reading a single line of the statistics file into a buffer. Then the buffer is scanned to determine what sort of line was read. A normal line of phase statistics is checked for first. If it was not such a line, then every other possibility is tested. For each, if a match was found, the appropriate statistics variables are added to, and the program goes on to read a new line into the buffer. In some cases, there are no statistics variables to be changed, and the next line is simply read.

The most complicated case occurs when the line was indeed a line of per-phase statistics. A variable used to keep track of the total number of phases is incremented, in this case, then a large number of statistics variables are adjusted. For most of these variables, the contribution for this phase is simply added into those already collected. In a few cases, like queue length maxima, the new maximum is compared to the previously seen one, and a change is made only when the new one is bigger. After all variables are adjusted, the next line of the statistics file is read.

Once all lines have been read, `check/measure` is ready to print out its results to the standard output. It first prints the name of the configuration file used to run the simulation (gotten from the statistics file). Then it prints a header explaining the values of the third line, which contains the number of objects in the simulation, the name of the statistics file, the number of positive and negative event messages sent, the number of event messages committed, the number of events started and completed, the number of event bundles committed, the number of events interrupted and rolled back in the middle, the number of states saved, and the number of states committed.

`check/measure` then performs consistency checks. By matching various statistics against each other, `check/measure` can determine if this run completed correctly. (Some kinds of errors cannot be detected this way, but many can.) Some of these checks are very simple. For instance, the number of positive event messages sent must equal the number received. Others are a bit more complicated. For instance, the number of positive event messages received minus the number of negative event messages received minus the number of positive messages sent in reverse plus the number of negative messages sent in reverse must equal the number of event messages committed. (This equation tries to keep track of everything that can legitimately happen to a positive event message that was received - it can be committed, or cancelled, or returned to the sender, or cancelled when a negative message is returned to the sender.)

As each equation is checked, a line is printed out indicating the results of the check. If the equation balances, the line simply identifies the quantities in the equation. If the equation does not balance, the line is preceded by two asterisks, and followed by the amount by which the balance failed.

After all balancing equations are checked, a few more lines are printed out. One of these shows the version of TWOS used, the elapsed run time, and the number of nodes used. Another shows how much times was spent running objects, and another how much of that time was committed. The critical path length and the final object on the critical path are printed out, here, too. However, unless certain TWOS features were turned off, this length may be incorrect. (See Chapter 16 for details.)

Certain occurrences during a TWOS run can artificially lengthen the run. These include page faults, nodes going to sleep because of long running events (or other causes), and migration naks, which cause a migration to start again from the beginning. If any such occurrences happened during this run, the next three lines of the output from `check/measure` alert the user to their presence, and the fact that they may have corrupted the run times. The user might want to rerun the application to get a cleaner run time, as sometimes these problems disappear from run to run. If these problems recur frequently, the system developer might need to investigate them.

At this point, if none of the consistency checks failed, `check/measure` prints out a line indicating that they were all OK. Less happily, if some of the checks failed, `check/measure` instead prints a line indicating failure.

If the program is being run under the name "`check`," its execution is complete at this point. If it is being run as "`measure`,", it must also print out a line to a measurements file. The measurement file was opened earlier, if it was needed. If the user provided a file, that file name is used. If not, the file "`Measurements`" is used. In either case, `check/measure` tests to see if the file has any text in it. If not, `check/measure` first puts a header line into the file, containing short text descriptions of each field in the measurements lines. In either case, the collected data for the run is then printed out in a single line. Appendix B of the Time Warp User's Manual contains a description of each field in a measurements file data line.

When a new version of TWOS has been installed and compiled, the new version of the `check/measure` program should also be compiled. Often, old versions of this program will work correctly with the statistics files produced by new versions of TWOS, but not always. When this program is compiled, it should be installed under both the name check and the name measure. A link is usually the best way of ensuring consistency of the two copies.

E2. `collapse` Internals

A typical set of measurements for TWOS requires that multiple runs be made for every situation to be measured, because of the variations in run time and other TWOS statistics. Only by making multiple runs can the programmer have some confidence that the data gathered is representative of TWOS' performance. Even when measure has been used to compress a run's result into a single data line, however, scanning the multiple lines in a measurements file associated with a single situation can be confusing and difficult. `collapse` is a tool to help reduce the data gathered in multiple runs into a single measurements file line, for examination and plotting.

At first glance, `collapse`'s task would seem almost trivial. It needs to read all identical lines and average each field in the line. In actuality, however, a close examination of the data in the measurements file lines indicates that a bit more work is required. First, not all fields are numeric. It does not make sense to average the name of the configuration file, for instance. Second, some of the fields contain maxima. Those fields should not be averaged together. Instead, the maximum of the maxima should be selected.

To deal with these issues, `collapse` was written in a very general manner. It can deal not only with those problems, but with combining lines based on some key field other than the number of nodes, with fields containing minima, and with types of averages other than the standard arithmetic mean.

In order to support this generality, `collapse` uses a data file to instruct it how to handle each field of the measurement file's data lines.

A sample of this file can be found in the statistics subdirectory of the tools subdirectory of the TWOS directory on the distribution tape. The file is named `benchformat.mach`, and is the version of the `collapse` data file used for normal situations. This file contains five lines. The first line is

KEY nodes

This line instructs `collapse` that all lines in the measurement file with the same number of nodes are to be combined. The second lines is

AMEAN ALL

This line instructs `collapse` that the arithmetic mean of all fields is to be taken, unless further instructions are given. The third line is

AMEAN OFF filename nodes sqmax iqmax oqmax mpagfs

This line tells `collapse` that the arithmetic mean of the fields named above is not to be taken. The fourth line is

MAX filename sqmax iqmax oqmax mpagfs

which tells `collapse` to take the maximum of those fields. The fifth line is

COUNT time

This line instructs `collapse` to count the number of lines combined and output that count in a field titled "time".

`collapse` starts by opening some of its files. In addition to the measurement file from which the data is to be read, `collapse` requires a measurement file to write to, and a format file, as described above. The input measurements file and the format file are opened at this point. Then `collapse` reads the first line of the input measurement file. That line should be a header line containing the field names of all data fields in subsequent lines. If the header line isn't there, or is unreadable, `collapse` exits. If the line can be correctly parsed, the individual field names are saved in an array. This array consists of many entries, each of which is a data structure that can store several different pieces of data. The name of the field is one of those pieces.

Once the header line has been handled, the format file is opened. Each line of that file is dealt with individually. Generally, reading a line of the format file causes one or more entries in the array described above to have a part of the data structure altered. For instance, the "AMEAN ALL" lines causes every array entry to have a variable in the data structure set to indicate that the

arithmetic mean of that field should be taken, while the "AMEAN OFF" line removes that data structure information from only the named fields' entries.

After the format file has been completely read, the remaining lines of the input measurement file are read, individually. The key field is checked to determine if any lines with the same key have been read yet. If not, memory is allocated to hold data for all lines with this key, and certain data fields are initialized to zero. Then, whether or not a previous line with this key has been seen, the fields of the line that was just read are examined, one at a time. For each, the entry in the array of field information is checked to determine what to do with this field (average it, max it, etc.), and the proper operation is performed.

Once all lines have been read in, the output measurements file is opened. The header read from the input measurements file is printed out to the output file, in an augmented form. Each field is given a prefix that shows what `collapse` did with that field. Also, if the format file contained a COUNT line (as the example described above did), then an extra field marked as a count field with the provided name is inserted into the header line, just before the key field. After the header line has been written out, the data is written. One line is written for each key value found. Once all keys have had lines written for them, `collapse` is done, and closes its files.

Appendix F: Unimplemented TWOS Features

This appendix consists of two parts. Part 1 concerns the proposed event cancellation facility. Part 2 concerns the proposed speculative computing facility. Each part describes a proposed design for the facility in question. Little review has been done on these designs, and typically the design changes during implementation, but these designs are probably fairly close to what would have been implemented. These two designs can either assist someone who needs to implement these features later, or can serve as a model of the design of other TWOS facilities.

F1. Event Cancellation

F1.1. Introduction

Message cancellation is one of the two major new features for TW 2.7. The other is the ability to allow users to predict their output. The user interface for both features was described in memo PLR: 366-91-11. This memo will cover internal design issues of the message cancellation feature. A later memo will cover the design of output prediction.

The message cancellation user interface specified in the earlier memo allowed users to cancel their messages in two ways. First, they could use an `unschedule()` call, which took the same parameters as the `schedule()` call, and resulted in the cancellation of the matching message. The other method was cancellation by reference using the `cancel()` call. This call takes a single parameter, a unique identifier for the message that must be cancelled. Its result is the same as `unschedule()`. Both `unschedule()` and `cancel()` permitted processes to cancel messages sent by other processes. Neither call directly addressed the issue of cancelling dynamic creation and destruction messages, though probably they will both permit it in the actual implementation.

F1.1.1 TWOS Message Sending Review

Before proceeding to the detailed design of the new calls, a brief review of message sending internals in TWOS is worthwhile. When the user wants to send a normal event message in TWOS, he calls `schedule()`, providing the receiver, receive time, message selector, text length, and text pointer for his message. `schedule()` runs with the user's stack, not TWOS', so it has very limited capabilities. Currently, all it does is some simple error checks and some timings, then it calls `switch_back()`. `switch_back()` is the routine for switching from the user's context into the executive's context, and is given a number of parameters, the first of which is the system routine that should be performed to handle the user's request. In the case of sending a message, that routine is `sv_tell()`.

`switch_back()` sets up TWOS' stack to run `sv_tell()`, then calls it. `sv_tell()` sets up the OCB's argblock field for this message, then sets its run status value to cause the dispatcher to send out a message. `sv_tell()` then calls `dispatch()`, which will try to send the message immediately via `sv_doit()`. `sv_doit()` may fail, but whether or not it does, control will return to the main loop to determine what to do next. Eventually, the main loop will reschedule the process that tried to send the message, the message will be successfully sent, and control will return to the process.

On the destination end, when a message is to be delivered the system tries to find the appropriate place in the receiving process' input queue. The system will either find the correct place to put the message, or determine that the incoming message annihilates a message already in the queue. This search for the appropriate place in the input queue is performed on a message-by-message basis, starting from the front of the queue. Each message has its receive time and selector fields checked against those of the incoming message; if those both match, TWOS does a byte-for-byte comparison of their texts. If the texts fully match, and the messages are of opposite signs, they annihilate. If any comparison fails, or the messages are of the same sign, the new message is simply enqueued. Whether enqueueing or annihilation occurs, the process will be checked for rollback.

F1.2. `unschedule()`

The `unschedule()` call will rely largely on existing TWOS cancellation mechanisms. It will behave exactly like a `schedule()` call, except that the positive copy of the message will be retained in the local output queue and the negative copy will be shipped to the receiver, rather than visa versa. The positive copy is queued in send time order, meaning that it will probably not annihilate with the negative copy of the message that is to be cancelled. (Unless both the original message and its cancellation are sent from the same event, which is a special case.) The negative copy is sent forward to the receiving process. Assuming that this is a proper cancellation, the negative copy will find a matching positive copy in the receiving process' input queue, which was put there by the original `schedule()` call. The negative and positive copies will annihilate, possibly causing a rollback if the positive copy had already been processed.

Figure F1.1 shows an example of the results of a matching `schedule()` and `unschedule()` call. Process A ran an event at time 100 that sent a message X to process B, to be received at time 300. Process A then ran a second event at time 200 that called `unschedule()` for message X. Therefore, process A has a negative copy of X in its output queue timestamped 100, and a positive copy timestamped 200. Process B has no copies at all, since the other positive and negative copies of X annihilated in its input queue. Should the event at time 200 be rolled back, the positive copy of X will be transmitted to process B,

effectively rolling back the `unschedule()`. Should the event at time 100 also be rolled back, both the positive and negative copies of X would be transmitted to process B, where they would annihilate, effectively rolling back both the `schedule()` and the `unschedule()`. Should only the event at time 100 be rolled back, only the negative copy of X would be transmitted to B, leaving the positive copy in A's output queue. If this sequence of events was committed, TWOS would signal an error, since the user has tried to cancel an event that was never scheduled.

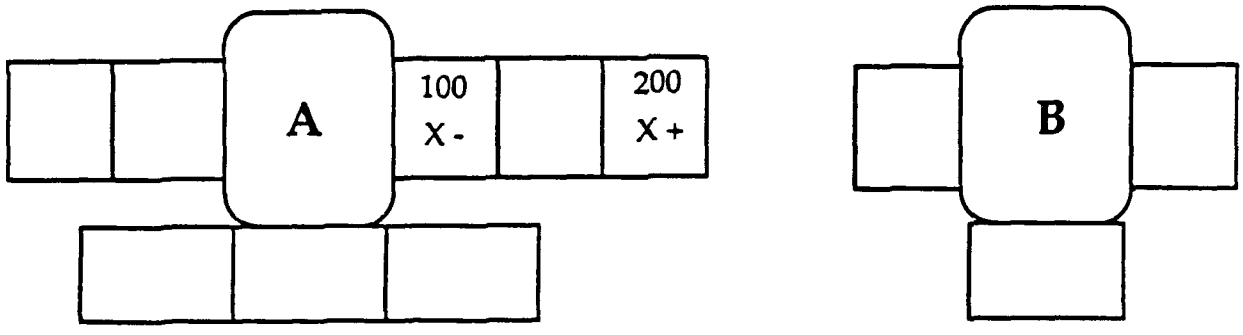


Figure F1.1: A Matching Schedule and Unschedule

F1.2.1 Basic `unschedule()` Internals

`unschedule()` will be a call looking very much like `schedule()`, as, in essence, there is only a "one bit" difference between the calls. It will set up certain fields and take clock timings, then use `switch_back()` to trap to the executive. One of the parameters of `switch_back()` specifies the routine that the executive should execute once its stack is in place; that parameter is set to `sv_tell()`, for `schedule()`, and should be set to the same value for `unschedule()`. `sv_tell()` should be altered to allow it to serve as a general purpose message posting routine. Relatively little code in `sv_tell()` will care about whether a normal event message or a cancellation is being handled, so the changes should be slight. The argblock field in the sending process' OCB will need to be expanded to permit subsequent routines to determine whether to send out the positive or negative copies of the message, and `sv_tell()` will need to accept another parameter to permit it to distinguish between the two routines that might have called it.

`sv_doit()` will also require minor alterations, as it currently assumes that the positive copy of the message is sent to the receiver and the negative copy enqueued. `unschedule()`, of course, needs to do the reverse, so some code in `sv_doit()` will need to be changed to permit either version of the message to be sent, and either version to be enqueued, depending on whether `sv_doit()` is servicing a `schedule()` or an `unschedule()` call. Otherwise, little in `sv_doit()` should need to be changed.

No new code or code changes should be needed on the destination end to support `unschedule()`. The message enqueueing code in `nq_input_message()` is already prepared to deal with an arriving negative message about to perform a cancellation. From the receiving process' point of view, there is no difference between this user-requested cancellation and a cancellation due to rollback. The rollback code is also already designed to do the correct thing when the negative message arrives.

F1.3. `cancel()`

`cancel()` is somewhat more complicated than `unschedule()`. `cancel()` takes a unique identifier for a message and causes the cancellation of the message that was issued that identifier. Unlike `unschedule()`, the user need not know the selector of the message or its text. (The receiver and receive time are contained in the unique identifier.)

`cancel()` could be implemented in several different ways. The unique identifier could be used simply as a key into the sending process' output queue. The identifier would be used to find the negative copy of the matching message, a new pair of message copies would be made (one positive, one negative), the negative copy could be sent and the positive copy queued. The advantage of this method is that it largely reuses existing TWOS code for message cancellation. The only new code is the code that finds the referenced message from its unique identifier and makes the additional copies. The disadvantage is that this method does not permit simple cancellation by third parties – process Z could not easily use `cancel()` to cancel a message sent from process X to process Y, since process Z would not have a copy of the message in its output queue. Perhaps more importantly, it also will not work well with temporal decomposition.

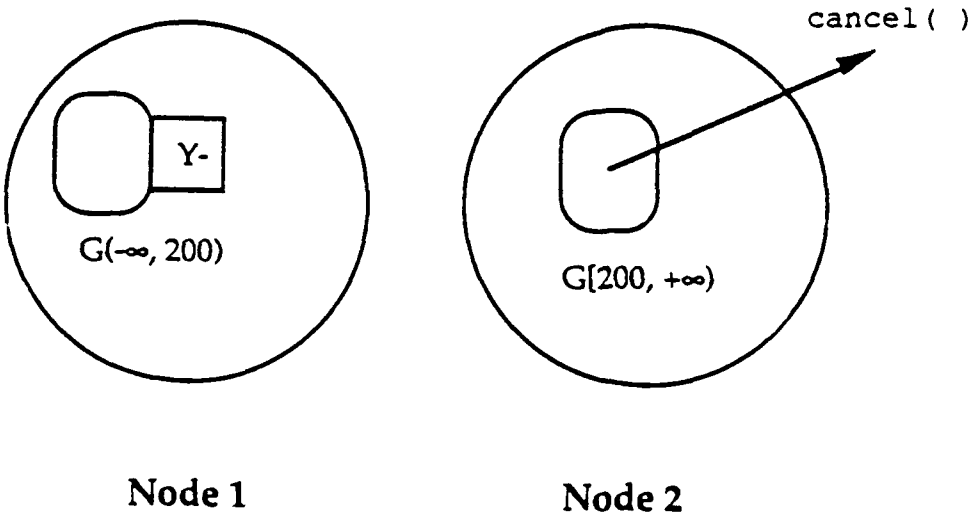


Figure F1.2: A Third Party Cancellation

If an object can be split into multiple phases, then a `cancel()` call may try to cancel a message that is not stored in the local phase's output queue. Figure F1.2 shows an example. Object C has been split into two phases, at time 200. The phases are named G(-∞, 200) and G[200, +∞), and are located on different nodes. G(-∞,200) has sent message Y to object H at time 100, and has a negative copy of Y in its output queue. G[200, +∞) runs an event that tries to cancel Y using `cancel()`. (Note that there would be no difficulty if G[200, +∞) used `unschedule()`, instead.) But G[200,+∞) does not have a local copy of Y. For the cancellation to succeed, G[200, +∞) would have to forward the cancellation request to G(-∞,200), which would find the negative copy of Y, make two new copies, and send the negative copy to object H.

There are still further complications, though, as the new positive copy must be queued by G[200, +∞), not G(-∞, 200). If the positive copy is not queued at G[200, +∞), then rolling back the event that caused the `cancel()` would be very difficult, as there would be no local record of that operation. So the positive copy would have to be sent to G[200, +∞) in such a way that that phase would realize the message should be put in its output queue. More complications arise if the phases split in the middle of this process. Similar complications arise if you want to support third party cancellation of messages.

An alternate method of implementing `cancel()` would be to have the call create an instance of a unique cancellation message type. Messages of this type would count as user messages for systems purposes, but would have only one effect – the cancellation of some other user message. Such a message would not need to contain the selector or text of the message to be cancelled, which would preserve it from the troubles illustrated in Figure F1.2. When G[200, +∞) executed a `cancel()` for message Y, two copies of a special cancellation message would be created. The positive copy would be sent to object H, the negative copy would be stored in G[200, +∞)'s output queue. The only contents of the cancellation message would be the unique ID. (The receiver and receive time would be implicit in the message header.)

When the positive copy arrived at H, H's input queue would be searched for a message with the same receive time and unique ID. When found, the positive copy and special cancellation message would not annihilate. Instead, they would simply be queued together. But no user event would be run for that virtual time at H, unless H has some other message also for that time. In effect, the special cancellation message would make the presence of the positive copy invisible to the Time Warp scheduler. (The messages cannot annihilate here because the `cancel()` call might be rolled back. If it is, then G[200, +∞) will have only a negative copy of the special cancellation message to send to H, which cannot take the place of a full copy of the message. Thus,

the receiver must make sure a copy of the actual message text is saved, in case the cancellation is rolled back.)

F1.3.1 Unique Identifiers

`schedule()` needs to return the unique identifier for the message being sent, in order to permit `cancel()` to later cancel it. This unique identifier consists of the name of the receiving object, the receive time, and a unique ID, which, in turn, consists of the node number of the node hosting the sending phase and a unique integer within that node. The receiver and receive time are needed to ensure that the `cancel()` call can correctly route its cancellation message. The unique ID is needed to guarantee correct identification of the message to be cancelled.

The unique identifier cannot be determined until lazy cancellation has been checked, because messages not resent due to lazy cancellation must return the same unique identifier as they returned the first time they were sent. Figure F1.3 illustrates the problem. Object A has sent message Y to object B. Y is assigned unique identifier `<uid1>` (figure F1.3a). The negative copy of Y is in A's output queue, the positive copy in B's input queue, and a copy of <uid 1> is stored in A's state, in case A later needs to cancel Y.

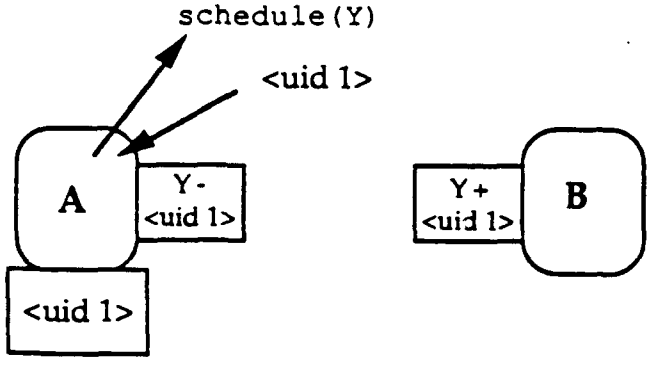


Figure F1.3a: Y Is Assigned a Uid

Object A then rolls back and re-executes the event, and again generates message Y. Lazy cancellation allows the system to avoid resending message Y, but this second `schedule()` call for Y returns `<uid2>` (figure F1.3b). Since lazy cancellation was used, the old version of Y is left undisturbed. (If the fact that a new uid was generated caused lazy cancellation to cancel Y, lazy cancellation would never avoid message resending, so the negative copy of Y must not be shipped to B.)

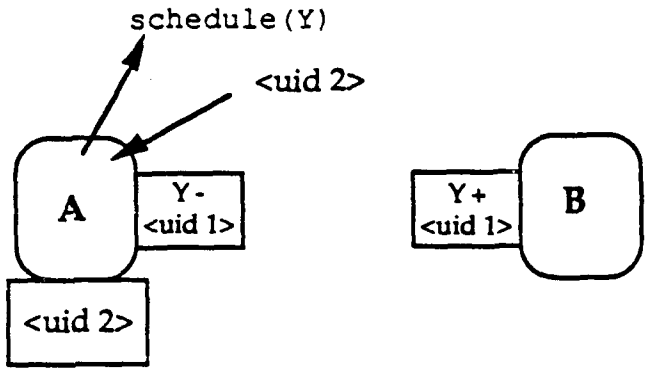


Figure F1.3b: Y Is Assigned a Second Uid

If `<uid1>` != `<uid2>`, then the `cancel()` call will be unable to properly cancel message Y. `<uid2>` will be saved in some special place in A's state, and, during a later event, A may choose to cancel Y. If it does, by calling `cancel()`, A will provide `<uid2>` as `cancel()`'s parameter and TWOS will send off a special cancellation message with that unique ID. Since the receiver and receive time of Y did not change, the special cancellation message will go to B, the right object. However, figure F1.3c illustrates what will happen at B. Message Y's copy at B still has `<uid1>`, since lazy cancellation did not update it. The special cancellation message contains `<uid2>`, so it does not match with Y and does not cancel it.

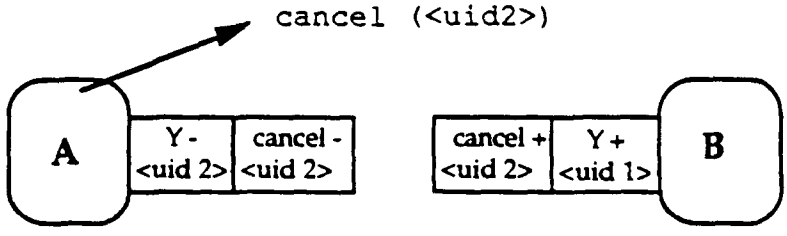


Figure F1.3c: The Cancellation Does Not Match Y

One alternative would be to change the uid in the negative copy of Y to <uid 2> when it is resent by the user. This method does not work either, however, as B's copy of message Y still has <uid 1> attached, so the cancellation mechanism cannot match the special cancellation message with Y. One can imagine special schemes for the system to automatically to send the new <uid

2> to B, making sure that the positive copy of Y was marked with the new unique identifier, but such schemes are cumbersome and would negate much of the benefit of lazy cancellation, since an extra message to carry the unique identifier update would be required.

Therefore, after resubmitting a message not sent due to lazy cancellation, the sender must be given back the same unique ID as he was when the message was first submitted. Fortunately, that unique ID is available in the negative copy of the message still in the output queue. (That copy must be there, or lazy cancellation does not take place and a normal send is performed, instead.) The only trick is to get the unique ID back to the user.

Referring back to section F1.1.1, lazy cancellation is checked early in the routine `sv_doit()`. That routine, however, already has a meaningful return value, so `sv_doit()` cannot return the unique ID already sent. The unique ID must be available when TWOS returns from `schedule()`, however. Therefore, we must use some other method of getting the information known only in `sv_doit()` to `schedule()`, several steps up in the stack.

Probably the easiest method will be to have `sv_doit()` store a unique ID in the sending OCB's argblock structure. Sending a message requires filling the argblock structure with all necessary information, including receiver, receive time, and a text pointer. These fields are all available in `sv_tell()`, so that routine fills them in for `sv_doit()`'s use. We must add one further field to the argblock structure, a unique ID field. Unlike the other fields in the argblock structure, this field would be filled in by `sv_doit()` for use by `schedule()`. If lazy cancellation saves retransmission of a message, the unique ID from the negative copy of the message would be copied into this field by `sv_doit()`. If the message actually had to be sent, `sv_doit()` would generate a new unique ID and copy it into the argblock field. When `schedule()` gets control again, as a result of a `switch_over()` call from the main TWOS loop, it can construct the entire unique identifier from the OCB's argblock fields.

F1.3.2 Basic `cancel()` Internals

When a `cancel()` call is made by the user, the system will proceed in much the same way as it would for any other message send. `cancel()` will set up `switch_back()` to call `sv_cancel()`. In `sv_cancel()`, the data provided by the user will be copied into the argblock structure. The receiver and receive time are retrieved from the unique identifier that serves as `cancel()`'s only parameter, and the text length is set to 0. There is no text body. The argblock's message type flag must be set to indicate that what is to be sent is a cancellation, rather than a regular event message, dynamic create message, or dynamic destroy message. `sv_cancel()` will need to generate a new unique identifier, as we will allow cancellation of cancellations, should application programmers want to use them.

`sv_doit()` should be able to handle this call with little or no alteration.

On the receiving end, the receiving process' input queue would be scanned for a normal event message matching the cancellation message. Text and selector would not be considered, only receive time and unique ID. Only a positive copy of the cancelled message will match a cancellation message. If a negative copy was found in the queue, it would not be affected by the arrival of this special cancellation message. In any case, whether a match was found or not, the special cancellation message is simply put in the input queue. If a positive copy of the message to be cancelled is in the queue, the cancellation message is put in back of it. If the positive copy of the event message has not arrived yet, the cancellation is enqueued in its proper place. When the positive copy does arrive, it is put in front of the cancellation message.

The rollback mechanism does the rest of the work. When `go_forward()` calls `earliest_later_inputbundle()`, that routine will have to check to see if the message it wants to choose is immediately followed by a matching cancellation message. If so, `earliest_later_inputbundle()` keeps looking for another eligible input message. Since `earliest_later_inputbundle()` will never return a pointer to an event message with a matching cancellation message, cancelled messages will never be scheduled.

However, the receiving process may have other messages for the same virtual time. If they are queued after the cancelled message, everything will go well. `earliest_later_inputbundle()` will set its pointer to the first such uncanceled message, and `objhead()` will set up its event. If the uncanceled messages are queued before the cancelled message, matters are a bit more complicated. `objhead()` will call `message_vector()` to make a copy of all messages for this virtual time for the user, but `message_vector()` must not make a copy of either the cancelled message or the special cancellation message. `message_vector()` should merely skip them and go on to the next message in the bundle.

If a cancellation message is enqueued without its matching event message, `go_forward()` and `message_vector()` should simply skip over it, too. If the cancellation message is the only message for this virtual time, no event should be scheduled. If there are other messages in the bundle, the user should get no indication that an unmatched cancellation message is also in the bundle.

At commit time, positive copies of message cancellations should be in input queues, and should have matching positive event messages. Should the commit mechanism find an unmatched committed cancellation message in an input queue, it should flag an error and call `tester()`. Multiple cancellation messages for a single event messages should also be flagged as an error, during commitment.

Normal TWOS rollback mechanisms will apply to cancellation messages. If the sender of a cancellation message rolls back the cancelling event (and lazy cancellation does not detect that the cancellation message is resent), then the negative copy of the cancellation message is sent to the receiver. Upon arrival, it will annihilate with the positive copy in the receiver's input queue, possibly causing a rollback, if the receiver's SVT is beyond the receive time of the cancellation message. If there is no corresponding positive copy of the cancellation message, the negative copy will simply be enqueued. An enqueued negative copy of a cancellation message must be ignored, for all purposes except annihilating its positive copy, when it arrives. If a negative copy of a cancellation message is ever committed without annihilating, TWOS should flag an error

Normal message sendback can be applied to positive copies of cancellation messages. For this purpose, they can be treated the same as event messages.

F1.3.3 Cancellation and Message Priority

Time Warp proceeds by generating a wave of computation, some of which is cancelled by a wave of anticomputation. Theoretical results demonstrate that, in the infinite processor case, the wave of anticomputation must travel faster than the wave of computation to guarantee that a finite computation will complete under Time Warp. In the finite processor case, the cost of having the anticomputation wave travel more slowly is not incorrect execution, but inefficient execution. If the wave of anticomputation does not travel faster than the wave of computation, Time Warp can process a lot of optimistic garbage before it is all rolled back.

Superficially, user-requested cancellations might seem like part of the wave of computation, rather than anticomputation. They are mirrored exactly in sequential runs, after all. However, a closer examination of the effects of user-requested message cancellations shows that they must be treated as part of an anticomputation wave. The issues involved here are theoretically complicated and not yet thought through, entirely. However, in the practical question of how Time Warp 2.7 should handle the issue, we will simply treat user-requested cancellations, both from `unschedule()` and `cancel()`, as high priority messages similar to negatively signed messages sent for rollback purposes.

`unschedule()` already sends normal negative messages, so few code changes should be necessary to give correct priority to that version of user cancellation. `cancel()`, however, sends a positive copy of a special message, so the lower level code will need to be changed to recognize that this message, while neither a system message nor a negative message, should receive priority handling. In the code dealing with send buffers on the Butterfly, for

instance, these messages should be permitted to use the send buffers currently reserved for negative messages.

A future memo will cover the theoretically correct methods of handling cancellations and rollbacks of cancellations, from a priority point of view.

F1.4. Commitment, Statistics, and Other Issues

As stated earlier, parts of the commitment code will need to be changed to handle this new feature. There are new types of committed errors to check for, for instance. Another change is that it will now become acceptable to have committed positive messages in a phase's output queue.

We will need to keep track of several new statistics to monitor performance of message cancellation. These will include the number of times `unschedule()` is called, the number of committed unschedules (calculated from the number of committed positive messages in processes' output queues), the number of times `cancel()` is called, the number of positive and negative cancellation messages sent forward, the number of positive and negative cancellation messages received forward, the number of positive and negative cancellation messages sent backward, the number of positive and negative cancellation messages received backward, and the number of times cancellation (of either type, kept in separate statistics) caused rollback. We may think of more important statistics, as they are needed.

These statistics must be accurately gathered, printed for each process into the XL_STATS, and properly handled by `check`. `check` should not only read these new statistics and produce a measurement file line including them, but should perform consistency checks, such as determining if the number of positive cancels sent equals the number received, and so forth. We may also need to update some of the existing consistency checks, as these new ways of sending messages may need to be taken into account in some of them.

These forms of message cancellation should not have difficult interactions with other parts of the TWOS source code. For instance, they should have no effect whatsoever on the code dealing with dynamic memory segments. One area of code that might require care is process migration. Process migration will need to be prepared to properly migrate the special cancellation messages, and it will need to properly migrate and queue negative messages in the input queue. Probably, little work will be required to make migration interact smoothly with message cancellation, but the migration code should be checked. `nq_input_message()` and `nq_output_message()` may also require changes to properly handle enqueueing of migrating cancellation messages.

F2. Event Prediction Design

F2.1. Introduction

Event prediction is one of the two major new features for TW 2.7. The other is event cancellation, whose internals were described in memo PLR:366-91-21. The user interface for event prediction was covered in memo PLR: 366-91-11. This memo will cover internal design issues of the event prediction feature. While neither major TW 2.7 feature will be implemented, this memo will cover the design to serve as documentation, in case of future need.

Event prediction permits users to gain performance advantage of their ability to guess at future events that are not yet certain. Event prediction has no semantic effect on a simulation – the simulation will produce the same committed results regardless of how many or how few times prediction is used, and regardless of how often the predictions are correct. If most predictions are correct, the event prediction facility may provide superior performance. If most predictions are incorrect, the event prediction facility's overhead may slow down the simulation. Test simulations can be written that will make event prediction look very good or very bad. With sufficient work, analytic formulae can be obtained to characterize when event prediction will win and when it will lose, but such formulae would be of little practical value, as they would involve terms that cannot be easily obtained from simple analysis of code, such as the probability that a given prediction is correct. Only experimentation with actual simulations will determine the value of this facility.

The prediction facility is based around one user call, `predict()`. This call is in many ways similar to the normal `schedule()` call, except that it requires two extra parameters, a send time and a sender. `schedule()` always assumes that the send time is now, and the sender is the process running, but prediction will not require those assumptions. In fact, predicting with a sender and send time of now would be of no value, as it amounts to a guess about what the current event will do. Since the current event either will or will not do that, regardless of the prediction, no benefit can be gained from such a prediction. Predictions of the currently running event will be treated as illegal by TWOS. Predictions for the current sender, but a different send time, are legal. Predictions for the current time, but a different sender, are not legal, since prediction is only defined for future send times.

The sequential simulator will accept `predict()` calls, but will treat them as no-ops, since it cannot take any advantage of them.

F2.2. Review of the Prediction Mechanism

The prediction mechanism relies on lazy cancellation to do its magic. When a user calls `predict()`, two copies of the requested message are made, as usual. However, instead of simply attaching a send time of now and a sender of the executing process, then shipping one copy and queueing the other, TWOS will attach the requested send time and sender name. As before, the positive copy of the message will be shipped to the receiver. Since that process never examines the send time of the message, the receiver will not be able to distinguish this prediction from a normally scheduled event message, and will treat it identically in all ways.

If the sender is the current process, the negative copy of the message will be queued, as before, but with a different send time, putting it further along in the process' output queue. If the sender is not the current process, then the negative copy of the message will be shipped to the predicted sender and stored in that process' output queue. If the predicted message is in the past of that process, the predicted sender will roll back and re-execute the event, possibly resulting in cancellation of the message. If the predicted message is in the future of the process, the predicted message will simply be queued and will await execution of an event at its predicted send time. Once that event is executed, if the predicted message is indeed sent, lazy cancellation will prevent resending of the message. Since the receiver already has a positive copy of the correctly predicted message, and the sender has a negative copy in its output queue, all effects of the actual `schedule()` call have already been achieved. If the predicted message is not generated by the running of the event, the negative copy will be sent to the receiver by normal lazy cancellation mechanisms, erasing all traces of the prediction. Similarly, if no event is run by the predicted process for the predicted send time, lazy cancellation will erase the predicted message.

Predictions will not be able to be cancelled by the `cancel()` primitive. A `predict()` call will not return a unique identifier, so `cancel()` cannot undo its effects. `unschedule()` will not properly undo the effects of a prediction in the event that the predicted event does not happen. `unschedule()` will work properly if the predicted event does happen, but such a case should be regarded as cancelling the actual generation of the message, not the prediction of it. Effectively, a user cannot cancel a prediction. If the prediction is wrong, it will cancel itself.

F2.3. Basic `predict()` Internals

`predict()` will look much like a `schedule()` call, and its treatment will be similar. Like `schedule()`, `predict()` will set up certain fields and take clock timings, then use `switch_back()` to trap to the executive. The parameter passed to `switch_back()` to tell it what kernel routine to call will be set to

`sv_predict()`. `sv_predict()` will be modelled on `sv_tell()`. However, event prediction will require the argblock structure of the OCB to be expanded to include the sender and send time as explicit fields. All routines that handle user requests to send messages will have to set these fields. Except for `sv_predict()`, all of these routines will set the sender field to the name of the currently executing process and the send time field to the current event time. `sv_predict()` will set the fields to the values specified in the `predict()` call. It may be necessary to make some modifications to `switch_back()` to permit it to accept the extra parameters of sender and send time, so that they can be available to `sv_predict()`.

`sv_doit()` will also require modifications. `sv_doit()` is the routine that handles most of the business of sending a message, such as allocating the buffers for the positive and negative copies of the message and enqueueing or sending them, as necessary. The event cancellation facility required `sv_doit()` to have the flexibility to enqueue the positive copy instead of the negative copy, and prediction requires `sv_doit()` to have the further flexibility to send both copies (with appropriate message flag values to ensure proper treatment), rather than enqueueing one of them. `sv_doit()` must therefore have access to some form of information telling it what to do with the various message copies. This information will probably be kept in a field in the sending OCB's argblock.

The sending of the positive copy is straightforward. It is merely shipped out to the receiving process and treated in all ways as a normal positive forward message. The negative copy requires more care. If the predicted event is eventually to be scheduled by the predicting process, the negative copy can simply be enqueued in send time order, just like a normally scheduled message. If the predicted event is to be scheduled by some other process, however, the negative copy of the message must be sent to that process' output queue. In order to determine that enqueueing it there is the proper thing to do, a special value of the message flag must be set. (Normally, negative messages arriving at an output queue are due to message sendback, and should annihilate; if they don't, they are sent forward, which isn't the proper treatment for predicted messages.) Once properly enqueued, no further special effort is required.

A negative message copy to be enqueued at a third party sender may arrive in its past. In such a case, we could enqueue it and do a full rollback, with correct results. However, the effect of such a rollback will be simply to send the predicted negative message forward, whether or not the prediction was correct. If it was correct, there is already a copy of the predicted message in the output queue, so the prediction copy makes two copies. Since the predicted event sent only one copy during correct execution, lazy cancellation would send one of the two forward, with the predicted version being the version of choice for cancellation. If the prediction was incorrect, the re-execution of the

event will not generate the predicted message, and its negative copy will be sent forward. Since, in either case, the only effect of the rollback would be sending the predicted message forward, at the cost of extra overhead, TWOS can instead simply send it forward immediately in the case of its arrival after the sending process has already performed the associated event. This shortcut requires a little extra code in `nq_output_message()`, but that code would be on a rarely used execution path, so it would add little cost to the system.

F2.4. Unique Identifiers, Statistics, Migration, and Other Issues

The `predict()` call will not return a unique identifier to its caller, but it will generate one. When the negative copy of a predicted message arrives at the sender's output queue and is stored there, the predicted copy's unique identifier will be used by lazy cancellation as the identifier for the actual version of the message, assuming that the prediction is correct. How lazy cancellation does this, and why, is described in appendix F1, section F1.3.1. The mechanism outlined there will work without alteration for predicted messages, as well. There will be one peculiarity, which is that a message sent by a given node may not have a unique ID containing that node's number, since the predicting process might have been on another node. However, the unique ID is used only for uniqueness, and its node number field should not be used either by the application programmer or the system as definite information, so this peculiarity is not a problem. Process migration could also cause this peculiarity, even in the absence of event prediction.

We will need to keep statistics to determine how many times processes predicted events, how many times the predictions were correct, and how many times they were wrong. These statistics should be kept on a per-process basis, like other process statistics. We might also want to keep track of how many third party predictions were performed, and perhaps separate statistics on the fate of such predictions. `cubeio.c` and `check` will have to be changed to deal with these statistics, `cubeio.c` to output them, `check` to read them, perform consistency checks on them, and print them out in summary format.

Migration might be slightly impacted by this facility. The only expected impact is that some special treatment might be required to deal with predicted messages in output queues. It's also possible that there will be no impact on migration.

Users will only be permitted to predict positive messages, at this point, and may not predict calls to `cancel()`. As a result, there are no special problems involving message priority and prediction. The positive predictive messages will be handled as normal positive messages. Any negative predictive messages that may need to be shipped to an output queue, because of third party prediction, could be shipped at lower priority than other negative messages (since they will usually not cause rollback), but they will probably be left at high priority. There are expected to be few such messages in most runs,

and therefore their impact on true high priority messages will be small. The cost of treating them as a special case at the low level is probably higher than the benefit.

Appendix G: GP-1000/Mach Specific Internals

G1. Time Warp Context Switching

By Paul Springer

This document describes how context switching is done between the Time Warp Operating System (TWOS) and the applications that run under TWOS. The first part of the document is of a general nature and applies to all ports of TWOS. The second part goes into detail and describes specifically the GP1000 port of TWOS.

G1.1. General Principles

In preparation for executing an object, TWOS calls `objhead()`. Among other things, the code in `objhead()` calls `loadstatebuffer()` which allocates space for the object's stack, and puts a pointer to that stack in the stk field of the object's ocb. After calling `loadstatebuffer()`, `objhead()` calls `setctx()`, which points the global variable `object_data` to the object's state, and the global variable `object_context` to the top of the newly allocated stack space. Then `setctx()` places the entry point for the object at the top of the object's stack space. The main loop in `timewarp.c` then makes the following call:

```
switch_over(object_context, object_data)
```

This invokes the switching code itself, which is written in the assembly language appropriate for the individual machine on which TWOS is running.

G1.2. The Switch Routine

On the GP1000 the switch routine is contained in the module `BBNswitch.s`. It has two entry points: `_switch_over`, which switches control to the application, and `_switch_back`, which switches control back to TWOS.

The first thing `_switch_over()` does is to save the registers onto the TWOS stack, and then saves the TWOS stack pointer into a global variable. Next it looks at the top 4 bytes of the object stack. If they are 0, it means that the object was in the middle of execution, and it needs to resume execution. It resumes execution by taking the value in the 4 bytes below the top 4 bytes of the object stack, and putting that value into register A7, the stack pointer. This effectively points A7 to the proper position in the object's stack. The registers are restored from the object stack, and an rts instruction returns control back to the object.

If TWOS regains control because the object has called a TWOS service routine, TWOS might not immediately return to the object that called it. It might instead determine that another object has higher priority, and begin executing the new object. Eventually TWOS would return control to the object which originally called it.

Appendix H: Tester Commands

By Paul Springer & Peter Reiher

This appendix gives a brief description of the purpose and use of each command available from the TWOS tester debugging and configuration file reading module. (The internals of the tester are discussed in Chapter 18.) Some of the following commands are applicable only when used in the configuration file, and some are applicable only when entered at the Tester prompt. In general, those commands which display information on the terminal should only be used when in tester mode.

By default, a tester command is directed to the node number specified in the tester prompt. (The prompt has the format "#--Tester:", where '#' designates the node number.) If the command is preceeded by a number, it is directed to the node which matches that number. The new node then becomes the default node to which future commands are directed. If instead of a number, the command is preceeded by an asterisk, the command is directed to all nodes.

Ordinarily a command is send to the destination node as a Time Warp command message, with the received time set to the current GVT. If, however, the command is preceeded by a second number (in addition to the node number described above), the command message is scheduled for a time equal to that second number.

If the beginning of the line in a configuration file contains the # character, the entire line is taken to be a comment and ignored. The C commenting technique /*...*/ is also recognized.

If a parameter contains special characters such as commas, asterisks, parentheses, semi-colons or spaces, it may be quoted with the double-quote character. Tester commands can be written in upper or lower case, or a combination of both.

Many tester commands accept the name of an object as a parameter. Almost all of these commands can also accept the name of a phase, instead. A phase is named by giving both the name of the constituent object and a simulation time between the begin and end times of the phase. Both the name and time should be contained within a single set of double quotes. Even if the name contains special characters, a single set of double quotes is sufficient. The tester is not able to name phases by virtual time sequence fields, even though TWOS can split them that way, so, in certain cases, some processes may not be reachable by name from the tester.

@ file_name

Open another configuration file and begin reading from it, and return control to the current configuration file when done. These can be nested up to 10 deep.

QUELOG log_size

Set aside memory to save node utilization information. This is used in conjunction with dynamic load management. The information can later be printed by using the DUMPQLOG command.

DLM

Toggle the flag which enables dynamic load management. This flag is initially off.

THRESH utilization_difference

The parameter must be a number between 0 and 100. The default number is 10. It is used as a percentage in the program. If dynamic load management is enabled, migration will occur only if the difference in utilization between the busiest and least busy nodes exceeds this value.

MIGRATIONS limit

Limit the number of migrations per DLM cycle. The default value is 1. The maximum allowable is the lesser of 128 or half the number of nodes used.

SPLITSTRAT strategy_type

This command determines which strategy will be used to split objects for migration. **strategy_type** may assume an integer value between 1 and 4 inclusive. 1 forces use of the NEAR_FUTURE strategy, 2 the MINIMAL_SPLIT strategy, 3 the NO_SPLIT strategy, and 4 the LIMIT_EMSGS strategy. The default is NEAR_FUTURE.

CHOOSESTRAT strategy_type

Determine which strategy will be used in selecting an object for migration. Here **strategy_type** may be either 1 or 2. 1 forces the use of the BEST_FIT strategy, 2 the NEXT_BEST_FIT strategy. The default is BEST_FIT.

MAXOFF limit

Limit the maximum simultaneous migrations per node to this value, a non-negative integer. The default value is 1.

IDLEDLM cycles

Set the number of initial DLM cycles during which no object migration is done. By default this number is 3.

DLMINT **interval**

Set the interval (in seconds) between DLM cycles. The value of **interval** must be a positive integer. The default value is 4.

DUMPQLOG

Display the current contents of the node utilization table.

CPULOG

Open a file for output, and name it "cpulog". Time Warp outputs a message into this file everytime an object message is processed and control is passed to an object. The message consists of a "B" (for event messages) followed by the name of the object. Another message is written to the cpulog file when the object exits back to Time Warp. This message gives the name of the object and the amount of time it used.

SPLIT **name sim_time**

Split an object phase into two new phases, with the boundary between them defined by **sim_time**. The object's name is specified by the **name** parameter.

MOVE **name sim_time node**

Move an object's phase to the node specified by **node**. The phase is specified by the name of the object in the **name** parameter and **sim_time**, any point of time spanned by the phase.

SENDSTATEQ

Print a list of information on states that are scheduled migrate but have not yet done so. The information includes such things as what the destination node is, and when it is to be delivered.

SENDOCBQ

Print information about the ocb of each phase scheduled to migrate.

SUBCUBE **node number config_file**

Spin off a subset of processors which will run the same simulation code using the set of parameters found in **config_file**. The number of processors in the subset is determined by the **number** parameter, and the **node** parameter determines which node will act as the CP. Both **number** and **node** are integers.

GVTSYNC **time**

Sync all nodes at the simulation time given by parameter **time**, and start the elapsed time calculations at that point. Do not collect any flow log statistics before **time**. The **time** parameter currently must be an integer, but that should be changed to allow floating point values.

GVTINIT

Start the gvt interrupt process. By default this is done after Time Warp finishes reading the configuration file.

WINDOW time

Prevent any object from running more than **time** past the current GVT. Here the **time** parameter is a floating point value.

DELFILE local_name

Release all memory used to store the contents of file **local_name**, and take it out of the Time Warp list of files.

HELP

Display a brief summary of the commands.

ACKS

On the Butterfly, print the number of output messages waiting in the queue to be sent.

QUEUES

Display information about each message in the low level input and output queues.

NOW

Display the simulation time for the currently executing object.

MYNAME

Display the name of the currently executing object.

NUMMSGS

Display the number of messages to be processed by the current event.

MSG message_number

Display the contents of a message to be processed by the current event. Which message to select is indicated by **message_number**, with 0 indicating the first message.

OBJEND

Terminate the current event of the manual object. A manual object is an instantiation of an object of type manual. It is invoked just like any other object, by sending a message to it. Once invoked, the object prompts for tester commands to execute (via the MANUAL_EVENT prompt) until the **OBJEND** command is entered.

MEMANAL

Display information about memory that's been allocated, and memory left in the pool.

LVT

Display the virtual time for a node, for PVT, LOCAL_MIN_VT and MIN_VT. PVT refers to the minimum simulation time of all the objects on the node. PVT is calculated not when the command is issued, but at the last GVT click. LOCAL_MIN_VT is the value calculated to be the node's minimum virtual time during the last GVT cycle. Following a GVT cycle, MIN_VT is set to the new GVT value.

GVT

Display the current GVT value.

CLR

Clear the screen by outputting a "\f".

GO

Leave tester and resume execution.

TIMEON

Start the interrupt process that triggers periodic GVT calculations.

TIMEOFF

Stop the interrupt process that triggers periodic GVT calculations.

SHOWSCHEDQ

Display a list of all the ocb's on the node.

SHOWDEADQ

Display a list of all the ocb's in the dead ocb queue (currently not used).

DUMPMSG msg_id

Display header information and contents of the message specified by **msg_id**. You can get the **msg_id** by looking at the output of the IQ or OQ commands.

DUMPSTATE state_id

Display the state specified by **state_id**.

STADDRT state_id
Display the sizes for each address table of the state specified by **state_id**.

DOCB object_name
Display the ocb for the object whose name is **object_name**.

IQ object_name
Display the input message queue for the object named **object_name**.

OQ object_name
Display the output message queue for the object named **object_name**.

SQ object_name
Display a single line of information about each state in the state queue for the object named **object_name**.

MIQ object_name
Display the migrating input message queue for the object named **object_name**.

MOQ object_name
Display the migrating output message queue for the object named **object_name**.

MSQ object_name
Display the migrating state queue for the object named **object_name**.

MOCB object_name
Display the migrating ocb for the object named **object_name**.

PDOCB object_name time
Display the ocb for the phase of **object_name** which includes **time**.

PIQ object_name time
Display the input message queue for the phase of **object_name** which includes **time**.

POQ object_name time
Display the output message queue for the phase of **object_name** which includes **time**.

PSQ object_name time

Display the state queue for the phase of **object_name** which includes **time**.

TMEM

This command results in output which shows the total amount memory used in the input, output, and state queues of each ocb. It also displays the number of memory segments and total memory in the free pool (used for allocating object stacks) and the number of memory segments and total memory in the message pool (used for allocating message buffers).

NOGVTOUT

This command will set a flag in Time Warp which will prevent the regular GVT output lines from appearing during the run.

MAXACKS number

Set the maximum number of outstanding messages. This limits the number of messages sent by a node but not yet received by the destination node.

AGGRESSIVE

Change the message cancellation mechanism from lazy (the default) to aggressive.

BPO object_name

Set a breakpoint for the object named **object_name**. The next time this object is about to execute, tester() will be called. Using this command clears any breakpoint set via the **BPT** command. Only one breakpoint at a time may be active. Currently the test for breakpoints (in timewarp.c) is #if'd out, and so this must be modified and the module recompiled in order to have the breakpoint feature available.

BPT time

Break as soon as an object whose svt is at or later than **time** begins execution. Once the break is encountered, tester() will be called. Using this command clears any breakpoint set via the **BPO** command. Only one breakpoint at a time may be active. Currently the test for breakpoints (in timewarp.c) is #if'd out, so the module must be recompiled in order to have the breakpoint feature available.

CBP

Clear any breakpoint that has been set.

BP

Display information about the currently set breakpoint.

WPO object_name

Set a watchpoint for the object named **object_name**. Every time this object is about to execute, TWOS will print a message giving the name of the object and the object's simulation time. Using this command clears any watchpoint set via the **WPT** command. Only one watchpoint at a time may be active. Currently the test for watchpoints (in timewarp.c) is #if'd out, and so this must be modified and the module recompiled in order to have the watchpoint feature available.

WPT time

Print a message whenever an object whose svt is at or later than **time** begins execution. The message contains the name of the object and the object's simulation time. Using this command clears any watchpoint set via the **WPO** command. Only one watchpoint at a time may be active. Currently the test for watchpoints (in timewarp.c) is #if'd out, so the module must be recompiled in order to have the breakpoint feature available.

CWP

Clear any watchpoint that has been set.

WP

Show any watchpoint that has been set.

MEMSIZE memsize

Set the amount of free memory TWOS has available on each node. Note that a command line parameter also sets the amount of free memory. (See the TWOS User's Manual, Section 2.3.2.) The command line parameter sets an upper limit that will be used unless this tester command is issued. This command cannot allocate more memory than was set aside by the command line parameter, or, if that parameter was not used, by the default in the system. The primary purpose of this command is to ease testing of TWOS' behavior in limited memory.

NOSENDBACK

Disable message sendback. (See Chapter 10 for a description of message sendback.)

PENALTY size

Assess a penalty for a process each time it performs some undesirable action. This command allows the size of the penalty to be set. See Chapter 15 for further details.

REWARD **size**

Give a process a reward each time it could not be scheduled due to accumulated penalties. This command allows the size of the reward to be set. See Chapter 15 for further details.

HOMELIST

Show the home list entries for all processes whose home node is the local node. See Chapter 17 for more details of process location.

PENDING

Show all entries in the local pending list. See Chapter 17 for more details of the pending list.

CACHE

Show the contents of the local process location cache. See Chapter 17 for more details of the cache.

CENTRY **object_name**

Dump any cache entries for the named object from the local cache. Note that, because of temporal decomposition (see Chapter 12), there may be more than one entry. See Chapter 17 for more details of the cache.

HENTRY **object_name**

Dump any home list entries for the named object from the local home list. Note that, because of temporal decomposition (see Chapter 12), there may be more than one entry. See Chapter 17 for more details of the home list.

HOME **object_name**

Show the home node for the named object. See Chapter 17 for more details of the home list.

STOP

End the TWOS run immediately. Do not write out any of the files written at the normal termination of a TWOS run.

QUIT

Identical to STOP.

FLOWLOG **size**

Create a flow log for use by the fplot tool. Set aside **size** entries on each node. If this area is filled, any further entries will be dropped, but the area limits the memory available for normal TWOS operations, and may slow down the simulation, or even prevent its completion. See the TWOS User's Manual, Section 7.3, for further details.

MSGLOG **size**

Create a message log for use by the mplot tool. Set aside **size** entries on each node. If this area is filled, any further entries will be dropped, but the area limits the memory available for normal TWOS operations, and may slow down the simulation, or even prevent its completion.

DUMPLOG

Dump the flow log immediately, instead of waiting for the end of the run.

DUMPMLOG

Dump the message log immediately, instead of waiting for the end of the run.

PROPDELAY **multiplier**

Increase the length of every event by **multiplier** times. This command can be used to artificially change the granularity of events for existing simulations without changing their code. The additional time is used in busy looping.

ISLOG **size delta**

Maintain an instantaneous speedup log. Set aside **size** entries, and check for data every **delta** seconds. See the TWOS User's Manual, Section 7.4, for further details.

IS_DUMPLOG

Dump the instantaneous speedup log immediately, instead of waiting for the end of the simulation.

HOGLOG **time**

Maintain a record of the event on each node that took the longest uninterrupted time to execute a segment of user code. Only consider events until GVT reaches **time**.

CRITPATH

Calculate the critical path of the simulation. See Chapter 16 for further details.

TSQ **object_name**

Show the truncated state queue for the named object. This queue is always empty unless the critical path is being computed. See Chapter 16 for further details.

SENDBUFFS

(GP1000 only.) Show the contents of the low level send buffers on this node.

Appendix I: Transputer Implementation Details

The following attached document is a description of an earlier attempt to port TWOS to a transputer platform. This information may prove of some use to others attempting a similar port. Because this document only exists in hard copy form, there are no page numbers for it. Page numbering resumes at the end of this appendix.

TIME WARP ON THE TRANSPUTER

—A PRELIMINARY INVESTIGATION—

Leo R. Blume

Section 363

Jet Propulsion Laboratory

October 14, 1988

INTERNAL REVIEW DRAFT

This document presents the findings of the Time Warp Project's preliminary investigation into parallel processors based on the Inmos Transputer.

The background section below details the setting leading up to actual Transputer implementation of Time Warp. This is followed by brief descriptions of the Transputer hardware and software environment. The next section describes the applications on which tests were run and gives the results of the performance measurements. A conclusions section summarizes the current state of development and the recommendations section indicates how to proceed from here.

Background

The goals of the Time Warp development effort are to :

- Increase the functionality of Time Warp through the addition of new features
- Study Time Warp's dynamics by conducting performance studies
- Evaluate parallel processors for suitability as platforms for Time Warp, and
- Ensure that Time Warp remains portable.

The port of Time Warp to the Transputer addresses the second, third, and fourth goals. Porting to the Transputer allows us to investigate both the performance of the Transputer and the performance of Time Warp on machines with Transputer-like architectures. The Transputer effort also promotes portability by requiring that Time warp be modularized such that machine-dependent components are appropriately isolated.

Investigation Strategy

The original strategy was to conduct a short-term study of the Transputer. Test programs were to be written and, if time allowed, some effort would be made to port all or part of Time Warp. The product of this investigation was to be a report that would return a recommendation as to the feasibility and merit of porting Time Warp to the Transputer. Given the cautiousness of this approach, it was decided that rather than buy hardware which might scarcely be used, hardware and software should be acquired on a trial evaluation basis.

A number of companies build Transputer boards for the IBM and Apple Macintosh series of personal computers. In the end, it was Definicon, a local manufacturer of Transputer boards for the IBM PC, who agreed to loan the Time Warp group a four-node board for a one-month evaluation. It is to the considerable credit of Definicon that they have been willing to renegotiate the loan agreement several times when technical difficulties and a more ambitious set of goals required that the Transputer work continue beyond the original planned completion date.

Parasoft, a local company specializing in parallel processing software, agreed to provide a copy of their Cubix system software for use on the Transputer board. Time Warp had previously been successfully run with the Cubix software on the JPL Mark III Hypercube. Thus, using Cubix promised to provide the quickest route to a successful Time Warp port.

In actuality, the Cubix software for the Transputer was not completely ready in time for the Time Warp port. Thus, as will be discussed later, we developed our own software to perform message routing and host interface services.

The Development Environment

Parallel Processor Hardware

The Transputer is a general-purpose CPU with built-in hardware support for concurrency. Parallel processors based on the Transputer are generally lower in cost and smaller in size than more traditional parallel processing components.

There are two types of Transputers used in board-level products for personal computers: the T800 and the T414. Both are machines with full 32 bit data and address paths. The primary difference between the two devices is that the T800 has on-chip floating point hardware. In addition the T800 has twice the fast, on-chip RAM of the T414.

The Transputer-based parallel processor that was the object of this investigation was the Definicon T4 board. The T4 is a single, standard-sized card for the IBM PC. The board contains four Inmos T414 Transputers with one megabyte of RAM each.

Host Hardware

The Transputer board was installed in an IBM PC AT having 512K of memory and 40 MB of disk storage. The PC provides power to the Transputer board and acts as both an I/O console and a file server for programs running on the Transputer.

The other element of the host environment was a Sun 3/50 on which cross-compiles for the Transputer were performed. The Sun and the PC were connected to one another by a serial line.

Software

Definicon has developed a C compiler that supports Transputer-specific extensions to the C language. This compiler was provided with the evaluation hardware. Another company, Logical Systems, also makes a C compiler for the Transputer. The Logical Systems C compiler has been in service longer than the Definicon compiler and runs on several different platforms including the Levco board for the Macintosh II. Thus, in order to promote portability across different Transputer products, we elected to purchase the more widely-accepted Logical Systems compiler (it could not be obtained on a trial basis).

Porting Time Warp

The port of Time Warp to the Transputer proved to be among the most difficult of all the machines recently evaluated. This difficulty arose because of several factors that will be listed and then elaborated below.

1. Lack of debuggers
2. Limited host I/O capabilities
3. Long compile-download-test cycle
4. Bugs in the Transputer hardware and software.

The Transputer hardware lacks any kind of interrupt facility. This absence makes the development of debuggers very difficult. As a result, we had no source-level debugger for the Transputer and, as far as we know, true assembly language debuggers do not exist. Consequently, a large part of the evaluation period was spent writing debugging utilities.

The host I/O capabilities that are provided as part of the Logical Systems C run-time system were useful but not sufficient for Time Warp's needs. Specifically, we needed the ability to interrupt the Transputer array from the host, and we needed the ability to do I/O from any processor (not just the processor that is directly connected to the PC host interface).

The Cubix environment, had it been fully functional in June, would have provided most of these capabilities. As it was, implementing this functionality ourselves was quite time consuming.

The third problem was a long compile-download-test cycle. Time Warp was compiled on a Sun workstation, downloaded over a 9600 baud serial line to the IBM PC AT and then loaded onto the Transputer board. Much of this arcane arrangement could have been avoided were it feasible to compile Time Warp on the Transputer's PC host. Unfortunately, the limited processing capacity of the IBM AT would have led to even longer compilations thereby worsening the situation.

Compiling and linking on the Sun typically took one to two minutes. An average transfer to the PC took four minutes. Communication between the PC host and the Transputer is performed a byte at a time through I/O ports. Consequently, the download time from the host to the Transputer board is rather slow — taking well over a minute.

The fourth factor hampering the porting effort was the appearance of two major bugs. One was an undocumented "feature" of the Transputer hardware itself, and the other was a bug in the linker software. The hardware bug prevented Time Warp from running successfully on even a single node. The linker bug, by contrast, caused intermittent failures in all node configurations. These bugs

were so serious and subtle that an entire month was probably lost tracking them down and developing appropriate curatives.

The result of safely negotiating all the hazards of Transputer development was the creation of several new Time Warp modules — namely:

kernel.c
tpq.c
debug.c
switcher.s

The kernel module provides message system, host interface and timer services to Time Warp. The tpq module contains a set of queueing functions used by the kernel's message system. As the name implies, the debug module contains debugging routines including a rudimentary assembly level debugger. The switcher module contains the assembly language functions that implement Time Warp's context switching mechanism.

Writing our own kernel was a time-consuming process, but it may yield certain benefits. For one, it may permit certain Time Warp optimizations that are not possible when the message system is embedded in code to which the Time Warp developers do not have access.

The Appendix contains a list of the kernel calls used by Time Warp and a brief description of their functionality. A more detailed description of these modules will be provided in an upcoming memo.

Performance Studies

The major activity of the Transputer effort was just getting Time Warp to work. Consequently, there was relatively little time left for conducting performance studies. The performance studies that were performed addressed several issues :

- Time Warp performance on the Transputer
- The Transputer's raw performance as a computational engine
- The performance of the underlying messaging system

Time Warp Performance

To test Time Warp's performance on the Transputer, Time Warp must be run with as realistic a simulation as possible. Due to memory limitations, the most sophisticated model that was runnable was the Queueing Model.

The Queueing Model implements servers as objects and clients as messages that are routed in between the servers. The queueing model was tested with the following parameters and found to exhibit speed up.

Number of servers: 12
Number of clients: 30
Busy Loop Count: 10,000

The busy loop count is a parameter that specifies the number of additions that are performed each time that a server object executes its event section. Each execution of the loop results in one floating point and one integer addition. The purpose of this loop is to adjust the ratio of computation to communication — a ratio that has a strong bearing on the potential speedup of a Time Warp application.

The test results were:

Number of Nodes	Execution Time
1	132.0
1*	111.1
2	75.6
3	44.6
4	44.2

The *-ed one node timing indicates a configuration in which the node used for running objects was other than the host node. The host node is more heavily loaded with system activities than the other nodes since it manages timer operations and host I/O.

Observe that these are relative speedup timings. Absolute speedup measurements are not possible because the Time Warp Simulator has not yet been ported to the Transputer.

The speedup curve flattens out going from three to four nodes. Apparently the slight gain in work distribution that results from distributing the 12 objects across four nodes rather than three is consumed by the increase in communication costs.

Note also that the loop constant of 10,000 needed to get this speedup is relatively large. This may indicate that there are inefficiencies in either the Transputer's messaging mechanism or in the kernel software.

Transputer vs. Sun Timings

In order to estimate the raw computational power of the Transputer, the Time Warp Queueing Model running on one Transputer node was compared with the same benchmark run on a Sun 3/50 workstation. The Sun 3/50 contains a Motorola 68020 rated at 1.5 MIPS. It also contains a Motorola 68881 floating point unit. Timings on the Sun will be biased, of course, by the burden of Unix operating system overhead.

The first test compared the integer performance of the two machines.
For this test, the floating point addition in the Queueing Model's delay loop was removed, leaving only the integer addition that increments the loop counter.

The loop count was increased from 10,000 to 40,000 as an ad hoc means of compensating for the decrease in computation resulting from the removal of the floating point addition. The results were:

Time on Sun:	13.12 seconds
Time on Transputer:	11.42 seconds

Thus in terms of mostly integer operations, the Transputer out-performed the Sun by about 15%.

The next test compared the floating point performance of the two machines. For this test, the floating point operation was restored in the model's delay loop, and the loop count was reset to 10,000. The results were:

Time on Sun:	13.1 seconds
Time on Transputer:	132.0 seconds

In other words, the Transputer was 10 times slower than the Sun — even though the Sun had the additional handicap of running Unix.

It turns out that a large part of this performance differential is caused by the absence of floating point hardware on the T414 Transputer. Thus, the third test compared floating point performance without the benefit of floating point hardware. As with the second test, the busy loop was coded to contain 10,000 floating point additions. This time, however, the version of Time Warp running on the Sun was compiled in such a way as to cause all floating point operations to be done in software. The results were:

Time on Sun:	50.1 seconds
Time on Transputer:	132.0 seconds

This result suggests that the floating point software is more efficient on the Sun than on the Transputer.

Transputer Communication Performance

The test results discussed thus far make no comment on the relative efficiency of Transputer communications as compared to communications on other processors. To estimate communication performance, a series of tests were run with the Ping-Pong model. This model does nothing but send messages back and forth between a Ping object and a Pong object and is thus uniquely suited to measuring message latency.

Using a model in which 10,000 messages are sent between the Ping and the Pong objects, the following timings were recorded:

Object Location (node) Ping	Pong	Total Execution Time (secs)
0	0	121.9
0	1	234.4
0	3	377.1

As expected, the best time is achieved when both objects are on the same node. This will be the case for virtually any machine since message communication is faster on-node than off-node.

When the Pong object is moved to Node 1, which is adjacent to Node 0 in the topology used for this experiment, the total execution time increases by a factor of about two. This is not surprising and is roughly in line with the factor of degradation experienced by other machines when off-node communication is involved.

In the third test, the Pong object is moved to Node 3. Node 3 is not adjacent to Node 0 and thus some intermediate node must do message routing. The substantial increase in execution time reflects the expected penalty of the store and forward messaging that is used on the Transputer.

To accurately interpret these results, a few observations about application-dependent effects are in order. In real applications running on multiple nodes, there is more than one kind of concurrency. There is concurrency of computation (this concurrency is the basis for the ability to achieve speedup) and there is also concurrency of communication with computation. The natural overlap of communication and computation obviously doesn't happen in the Ping-Pong model and accounts, in part, for the fact that performance steadily decreases as a function of the distance between objects.

Additionally, during realistic multihop message communication, pipelining effects are observed that tend to increase the effective communication rate over that which the above tests would indicate. This pipelining effect occurs because more than one message may be inserted into the logical "pipeline" between two nodes before the first message reaches its destination. This effect is not present in the Ping-Pong model.

Thus, to determine how well Transputer communications will scale for programs whose communication patterns are irregular and widely distributed across the network of processors, tests must be conducted on larger arrays of Transputers with more realistic applications.

Conclusions

• Despite the difficulties, we have greatly exceeded our original goals.

The goal was to write test software and produce a report. Instead, we successfully ported a multinode Time Warp to the Transputer.

Additionally, in creating our own message passing kernel, we have created the only version of Time Warp that does not run on top of another operating system. This "vertical integration" allows us to perform Time Warp specific optimizations that are not possible in other environments. For example it is possible to eliminate message acknowledgements in the Transputer version of Time Warp since Time Warp can access the message system's queues to perform GVT calculations. It will be interesting to note the effects of such changes on overall Time Warp performance.

• The worst is behind us.

To conduct a thorough exploration of Time Warp on the Transputer, a number of significant activities need to be accomplished. These activities are listed below.

Major milestones

1. Create host interrupt capability
2. Build operational Time Warp context switcher
3. Build operational message passing kernel
4. Run Time Warp successfully on one node
5. Run Time Warp successfully on multiple nodes
6. Conduct preliminary performance studies
7. Port a substantial Time Warp benchmark
8. Port Time Warp Simulator
9. Complete performance study on a large (> 32) number of nodes

Milestones 1 through 6 have already been completed. Milestone 8 is the payoff point, and we are about 70% of the way there. Given that we are so close to getting the "payoff" of meaningful large-scale performance results, this effort must not be abandoned without strongly compelling reasons.

• There have already been benefits.

Time Warp is much more portable thanks to undergoing the rigors of the Transputer port. Portability is always an advantage.

Furthermore, the message passing kernel may itself have intrinsic value beyond the scope of the Transputer implementation. It may be used in part or in whole on other machines for which sufficient communications facilities are lacking. In short — it provides additional options for greater portability.

• The cost of moving forward is modest

Cost has two components: a dollar component and a manpower component. In terms of dollars, the cost of four fully configured Transputer nodes is only about $14,000. This is about the cost of a "loaded" Macintosh II. Compared to other machines, this is quite a low price for a four-node parallel processor. Furthermore, "buying in" to the Transputer at this level will allow us access to much larger Transputer configurations at Definicon (in much the same way that our Butterfly purchase has opened the door to very large machines at BBN).

The manpower cost is also reasonable. The exact requirements needed to accomplish milestones 7 through 9 are discussed in the recommendations sections.

• The potential benefits to the project itself are great

Running a significant Time Warp benchmark on a large array of Transputers will certainly advance our understanding of both Time Warp and the Transputer (or future machines architecturally similar to the Transputer). The project may also gain the ability to outfit desktop personal computers with low-cost parallel processors if the results of the Transputer evaluation are favorable.

• This work is of major interest to others

It is worth pointing out that far from being of merely parochial interest, this effort will generate publication quality results that others will find valuable. Specifically, interested parties include:

- The parallel processing community
- The simulation community
- The sponsors of the Time Warp project.

The parallel processing community is interested because this effort constitutes perhaps the most significant validation activity yet of the Inmos Transputer — a machine that, despite its several faults, is the first microprocessor designed specifically for parallel processing. The success or failure of a major non-numerical application such as Time Warp would make a very telling statement about the general usability of machines with the Transputer architecture. Indeed the results, if strongly negative or positive, may even influence the design of future parallel processor hardware.

This effort is of interest to the simulation community because it holds the promise of delivering a solution to a long-standing problem — the lack of credible desktop simulation platforms.

The Transputer work is also of interest to the sponsors of the Time Warp project. Ultimately, our sponsors hope to use a future version of Time Warp. Just as

Time Warp is evolving into a more usable piece of software through the addition of new features and the elimination of bugs, the hardware on which Time Warp runs should be evolving towards greater usability.

Usability is a function of many variables. However, the history of the personal computer has demonstrated quite dramatically that major components of usability are accessibility and locality. A machine may be physically local, but not electronically accessible. Similarly, a machine may be electronically accessible but not physically local.

A machine that sits on a person's desk and is not shared with other users offers the ultimate in accessibility and locality. If its other shortcomings can be overcome, the Transputer may be the key to creating the long-awaited desktop simulation workstation.

• The time to act is now.

It has been suggested that in light of the limited personnel resources, the best policy might be to checkpoint the work done thus far — hoping to come back to it next year or as additional funding becomes available.

The problem is that this suggestion ignores the fundamental dynamics of our software-development environment. Every two months of delay between the end of the current experiment and its ultimate continuation will result in a measurable increase in the difficulty of resuming the work. The increase in difficulty will be the result of many factors. Currently, the version of Time Warp that is running on the Transputer is the official released version. As soon as two or more versions of Time Warp have been released without a port to the Transputer, the difficulty of the port becomes markedly non-trivial.

Worse yet, delays in the implementation of more than three or four months would begin to erode the hard-won expertise that the group has developed in working with the Transputer. Delays on the order of a year would impose the added threat of personnel changes and changes in the underlying hardware and software. The result would be a 0% carry-over of the technology that has been built here during the last four months. **Thus, a decision to defer indefinitely is a decision to abandon this effort.**

Recommendations

Despite the fact that Time Warp now runs on up to four Transputers, there remains a substantial amount of interesting, undone work. The limited memory in the evaluation equipment prevented us from implementing a large Time Warp benchmark. Additionally, because we never had access to more than four nodes, we were never able to test Time Warp on a large array of Transputers.

To answer the question of whether or not the Transputer would be a suitable platform (in either large or small configurations), Time Warp must be tested with a real application. To determine the Transputer's effectiveness in a large numbers-of-nodes configuration, tests must be run on just such configurations.

Recommended activities with estimated times of completion are given below:

• Procure four-node Transputer	3 days over 2 weeks
• Install Transputer hardware	1 day over 1 week
• Port latest version of Time Warp	5 days over 4 weeks
• Port TW Simulator	5 days over 4 weeks
• Port STB88	8 days over 6 weeks
• Conduct Performance Tests	15 days over 3 weeks
• Document results	10 days over 10 weeks
Total:	47 days over 30 weeks

Start date: November 1, 1988
End date: June 1, 1989

The key factor in the schedule planning is reducing the impact to other Time Warp activities. The stretched-out schedule of activities reflects this priority.

Performance tests would be conducted at Definicon on a large array of Transputers that they have agreed to assemble for the sake of this experiment. Clearly the whole effort is contingent upon the availability of this resource and so the agreement must be firmed up before any other steps are taken.

The above course of action also depends on the commitment and support of our sponsor both in terms of funding and relaxation of other scheduled activities, if such should be needed.

Recommended Hardware and Software

The recommended hardware for a continued Transputer effort depends in part on the seriousness of the effort. Configuration 1, below, is the minimal set of hardware needed to continue at a baseline level. Configuration 2 adds to the components in Configuration 1 those items that would be needed for a more aggressive implementation strategy.

Hardware Configuration 1

Item	Quantity	Estimated Cost
• Definicon 4 node T800 board w. 4 MBytes of memory / node.	1	$ 14,000
• Ethernet Card for the AT	1	$ 750

A four-node system is with four megabytes of memory per node is the minimum configuration on which a major Time Warp benchmark will run. We will specify that Definicon hardware be purchased to eliminate the risk of hardware incompatibilities.

The T800 processor is needed because the floating point libraries currently available for the T414 processor do not support trigonometric functions. Most Time Warp applications use these functions for geometry calculations.

The Ethernet card is needed to facilitate fast file transfers between the Sun, on which compiles and links are performed, and the PC AT serving as a host for the Transputer.

Hardware Configuration 2

Item	Quantity	Estimated Cost
• All items in Configuration 1	1	$ 14,750
• 80386-20 PC Clone	1	$ 2,000

The use of a '386 machine would allow compilations to be performed on the host machine rather than having to be done on Sun workstations. This would greatly reduce the compile-download-test cycle that has hampered development to date.

Note that the Ethernet card is still necessary in Configuration 2. Though compiles would be performed on the '386 host, Ethernet access will be needed to support source code downloads as Time Warp and its applications continue to evolve.

Recommended software consists of:

TOPS network software for the PC.

TOPS software is now being successfully used by the Time Warp project to allow the Sun workstations to act as file servers for Macintoshes. Extending this capability to the PC is probably the best way to achieve transparent Sun file access.

Another candidate for acquisition is Parasoft's Cubix system software. Though no longer strictly needed for its message passing capability, Parasoft indicates that the latest version of their software contains debugging tools that might prove useful.

APPENDIX

Transputer Kernel Call Summary

kernel_main (argc, argv)

```
int     argc;
char  * argv[;
```

The very first thing that the Transputer implementation of Time Warp does is call kernel_main. Kernel_main initializes the kernel's input, output, and free queues; it reads in a file called NODES on the host that describes how the Transputer nodes are interconnected and broadcasts this information to all other nodes; and it creates processes that handle input, output, and host I/O. Finally, it inspects the Transputer RAM for illegal instruction sequences.

xsend (buff, len, dest, type)

```
char  * buff;
int     len;
int     dest;
int     type;
```

Xsend is the function used for general message output. The message supplied to xsend is enqueued at the end of an output queue for transmission at some later time. Xsend copies the input message into a buffer it allocates from the kernel's free list. Thus, the calling routine may dispose of or reuse the buffer as soon as xsend returns.

xrecv (buff, len, src, type, rtn_src, rtn_type)

```
char  * buff;
int     len;
int     src;
int     type;
int     * rtn_src;
int     * rtn_type;
```

The receive function that provides the analog to xsend is xrecv. The caller to xrecv requests the next message from node "src" of type "type" at the head of the node's input queue. The src field may contain the symbolic constant ANY_SRC. Similarly the type field may contain ANY_TYPE. If both wild-card constants are used, xrecv will just return the first message in the input queue. Time Warp typically acquires messages from xrecv using ANY_SRC and

ANY_TYPE. The actual source and type of the message returned by xrecv is passed back through the pointer arguments rtn_src and rtn_type.

signal (signum, ufunc)

```
int     signum;
int     ( * ufunc )();
```

Signal takes a signal number from the set of numbers defined in the file tpsignal.h and associates with it the user function ufunc. When the kernel detects the event indicated by the signal number, it calls ufunc. The signal interface is identical to Unix, but only one signal is currently implemented — SIGALRM, for alarm signal notification.

alarm (interval)

```
int     interval;
```

The alarm function works in conjunction with the SIGALRM signal. Alarm takes as an argument an interval length in seconds. After interval number of seconds have elapsed, the kernel will call the function associated with the SIGALRM signal, provided that such a function exists.

get_node_num ();

This function returns the processor node number.

get_num_nodes ();

This function returns the total number of nodes in use during the current execution of Time Warp.

kprintf (format, arg1, arg2, ... , arg15);

```
char    * format;
int     * arg1, arg2, ... , arg15;
```

kprintf provides a node-independent printf capability. When executed on node 0, kprintf prints directly to the host interface. When invoked on some other node, kprintf sends a message to node 0 containing the text to be output. The output is performed when the message reaches node 0.

brdcst_interrupt ()

brdcst_interrupt causes a high-priority message of type INTERRUPT_TYPE to be sent to all nodes. Time Warp's tester() function calls brdcst_interrupt to

inform other nodes of the need to immediately enter the command and data exchange protocol used by Time Warp's *tester()* command interpreter.

[Ebling 89] Maria Ebling, Michael DiLoreto, Matthew Presley, Frederick Wieland, and David Jefferson, "An ant foraging model implemented on the Time Warp Operating System", *Proceedings of the 1989 SCS Conference on Distributed Simulation*, Simulation Series Vol. 21, No. 2, Society for Computer Simulation, San Diego, 1989.

[Fujimoto 88] Richard M. Fujimoto, "Performance measurements of distributed simulation strategies", *Proceedings of the 1988 SCS Conference on Distributed Simulation*, Vol. 19, No. 3, Society for Computer Simulation, February 1988.

[Fujimoto 89] Richard M. Fujimoto, "Time Warp on a Shared Memory Multiprocessor", *Proceedings of the 1989 International Conference on Parallel Processing*, August 1989.

[Fujimoto 90] Richard M. Fujimoto, "Performance of Time Warp under synthetic workloads", *Proceedings of the 1990 SCS Conference on Distributed Simulation*, Volume 22, No. 2, Society for Computer Simulation, January, 1990.

[Gafni 85] Anat Gafni, "Space Management and Cancellation Mechanisms for Time Warp", Ph.D. Dissertation, Dept. of Computer Science, University of Southern California, TR-85-341, December 1985.

[Gafni 88] Anat Gafni, "Rollback mechanisms for optimistic distributed simulation systems", *Proceedings of the 1988 SCS Conference on Distributed Simulation*, Volume 19, No. 3, Society for Computer Simulation, February 1988.

[Gilmer 88] John Gilmer, "An Assessment of 'Time Warp' Parallel Discrete Event Simulation Algorithm Performance", *Proceedings of the 1988 SCS Conference on Distributed Simulation*, Volume 19, No. 3, Society for Computer Simulation, February 1988.

[Hontalas 89a] Philip Hontalas, Brian Beckman, Mike DiLoreto, Leo Blume, Peter Reiher, Kathy Sturdevant, L. Van Warren, John Wedel, Fred Wieland, and David Jefferson, "Performance of the Colliding Pucks Simulation on the Time Warp Operating System (Part I: Asynchronous behavior and sectoring)", *Proceedings of the 1989 SCS Conference on Distributed Simulation*, Simulation Series Vol 21, No. 2, Society for Computer Simulation, San Diego, 1989.

[Hontalas 89b] Philip Hontalas, Brian Beckman, David Jefferson, "Performance of the Colliding Pucks Simulation on the Time Warp Operating System (Part II: Detailed Analysis)", *Proceedings of the SCS Summer Computer Simulation Conference*, Austin, Texas, July 1989.

[Jefferson 82] David Jefferson and Henry Sowizral, "Fast Concurrent Simulation Using the Time Warp Mechanism, Part I: Local Control", Rand Note N-1906AF, the Rand Corporation, Santa Monica, California, Dec. 1982.

[Jefferson 84] David Jefferson and Andrej Witkowski, "An approach to performance analysis of timestamp-oriented synchronization mechanisms", *ACM Symposium on the Principles of Distributed Computing*, Vancouver, B.C., August 1984.

[Jefferson 85] David Jefferson, "Virtual Time", *ACM Transactions on Programming Languages and Systems*, July 1985.

[Jefferson 87] David Jefferson, Brian Beckman, Fred Wieland, Leo Blume, Mike DiLoreto, Phil Hontalas, Pierre Laroche, Kathy Sturdevant, Jack Tupman, Van Warren, John Wedel, Herb Younger and Steve Bellenot, "Distributed Simulation and the Time Warp Operating System", *11th Symposium on Operating Systems Principles (SOSP)*, Austin, Texas, November 1987.

[Jefferson 90] David Jefferson "Virtual Time II: The Cancelback Protocol," *Proceedings of the Conference on Principles of Distributed Computing*, 1990.

[Kleinrock 89] Leonard Kleinrock, "On Distributed Systems Performance", ITC Specialist Seminar, Paper number 3.2, Adelaide, 1989.

[Lakshmi 87] M. S. Lakshmi, "A Study and Analysis of the Performance of Distributed Simulations", Technical Report 87-32, Computer Science Department, University of Texas at Austin, August 1987.

[Lavenberg 83] S. Lavenberg, R. Muntz, and B. Samadi, "Performance and Analysis of a Rollback Method for Distributed Simulation", *Performance '83*, North Holland, 1983.

eung 89] Edwina Leung, John Cleary, Greg Lomow, Dirk Baezner, and Brian Unger, "The effect of feedback on the performance of conservative algorithms", *Proceedings of the 1989 SCS Conference on Distributed Simulation*, Simulation Series Vol 21, No. 2, Society for Computer Simulation, San Diego, 1989.

[Lin 89a] Y.-B. Lin and E. D. Lazowska, "Exploiting lookahead in parallel simulation", Technical Report 89-10-06, Department of Computer Science and Engineering, University of Washington, 1989.

[n 89b] Y.-B. Lin and E. D. Lazowska, "The optimal checkpoint interval in Time Warp parallel simulation", Technical Report 89-09-04, Department of Computer Science and Engineering, University of Washington, 1989.

[n 89c] Y. B. Lin, E. D. Lazowska, J. L. Baer, "Conservative Parallel Simulation For Systems With No Lookahead",TR 89-07-07, Department of Computer Science and Engineering, University of Washington, 1989.

[Lin 90a] Y.-B. Lin, and E. D. Lazowska, "Optimality considerations for 'Time Warp' parallel simulation", *Proceedings of the 1990 SCS Conference on Distributed Simulation*, Society for Computer Simulation, Volume 22, No. 2, San Diego, January, 1990.

[Lin 90b] Y.-B. Lin, E. D. Lazowska, and Jean-Loup Baer, "Conservative parallel simulation for systems with no lookahead prediction", *Proceedings of the 1990 SCS Conference on Distributed Simulation*, Society for Computer Simulation, Volume 22, No. 2, San Diego, January, 1990.

[ipton 90] Richard J. Lipton and David W. Mizell, "Time Warp vs. Chandy-Misra: A worst-case comparison", *Proceedings of the 1990 SCS Conference on Distributed Simulation*, Society for Computer Simulation, Volume 22, No. 2, San Diego, January, 1990.

[ow 88] Greg Lomow, John Cleary, Brian Unger, and Darrin West, "A Performance Study of Time Warp", *Proceedings of the 1988 SCS Conference on Distributed Simulation*, Volume 19 Number 3, Society for Computer Simulation, San Diego, February 1988.

[Sokol 88] Lisa Sokol, D. P. Briscoe, Alexis P. Wieland, "MTW: a strategy for scheduling discrete events for concurrent execution", *Proceedings of the 1988 SCS Multiconference on Distributed Simulation*, Volume 19, No. 3, Society for Computer Simulation, San Diego, February, 1988.

[Su 89] Wen-king Su, Chuck Seitz, "Variants of the Chandy-Misra-Bryant distributed discrete event simulation algorithm", *Proceedings of the 1989 SCS Conference on Distributed Simulation*, Simulation Series Vol 21, No. 2, Society for Computer Simulation, San Diego, 1989.

[West 87] Darrin West, Greg Lomow, Brian W. Unger, "Optimizing Time Warp Using the Semantics of Abstract Data Types", *Proceedings of the Conference on Simulation and AI*, Simulation Series, Vol 18, No. 3, January 1987.

[West 88] Darrin West, "Optimizing Time Warp: Lazy Rollback and Lazy Reevaluation", M.S. Thesis, Dept. of Computer Science, University of Calgary, January 1988.

[Wieland 89] Fred Wieland; Lawrence Hawley, Abraham Feinberg, Michael DiLoreto, Leo Bloom, Joseph Ruffles, Peter Reiher, Brian Beckman, Phil Hontalas, Steve Bellenot, and David Jefferson, "The Performance of Distributed Combat Simulation with the Time Warp Operating System", *Concurrency Practice and Experience*, Vol. 1, No. 1, Sept. 1989.

[Wieland 92] Fred Wieland, Tapas Som, Peter Reiher, John Wedel, and David Jefferson, "A Critical Path Tool for Parallel Simulation Performance Evaluation," Hawaii International Conference on System Science, Koloa, Hawaii, January 1992.

Index