OPTIMISING TIME WARP USING THE SEMANTICS OF ABSTRACT DATA TYPES

Darrin West, Greg Lomow, and Brian Unger
Department of Computer Science
The University of Calgary
2500 University Drive, NW
CALGARY, AB
T2N 1N4

ABSTRACT

The Time Warp mechanism for synchronising the execution of simulation components offers significant potential for achieving concurrency within distributed simulations. This mechanism takes advantage of the independence of most events in a simulation, enforcing only a partial ordering of events which allows many events to occur in an arbitrary order. There are many situations, however, where the partial order imposed by Time Warp is too restrictive, i.e., events are unnecessarily forced to occur in a specific order. This paper presents several example situations where Time Warp semantics can be relaxed, describes an approach based on abstract data types that can substantially improve performance, and outlines an implementation of this approach.

1. INTRODUCTION

The study of distributed simulation techniques is currently of great interest because of the recent availability of highly parallel computer architectures. These parallel machines consist of a large number of asynchronously operating processors that communicate in a variety of ways. The major problem associated with utilising these parallel machines, or multicomputers, is the lack of programming methods for dealing with parallelism or concurrency. Time Warp [1] implements Virtual Time using an optimistic mechanism which relies on generalised process lookahead and rollback. Time Warp offers a new approach to the synchronisation of distributed simulations.

The definition of Virtual Time ensures that messages are received by a process in timestamp order. For some processes in some applications this condition is overly strict and other equally valid orderings of messages are possible. For example, in many distributed systems a process encapsulates a data structure or data type (e.g. a queue or a resource) and these processes can be thought of as implementing abstract data types. For most abstract data types, non-conflicting operations do not have to be applied in timestamp order to maintain consistency[2]. The cause of the problem is clear: the only information that Virtual Time makes use of when specifying the order in which messages are to be received is the timestamps associated with the messages. If Virtual Time "knew" more about the processes it was synchronising, it could validate more orderings of messages because some operations do not conflict.

One consequence of using Virtual Time's definition of valid ordering is that it requires additional synchronisation overhead in some cases where it is not necessary. With a Time Warp implementation, this means that processes will be rolled back needlessly in some situations. If state restoration (undoing the effects of a process's erroneous computation) is expensive, then eliminating unnecessary rollbacks is worthwhile.

The goal of this research is to reduce the number of rollbacks that are necessary for synchronising processes in a Time Warp system. This will be accomplished by augmenting the implementation of Time Warp with process specific information that can be used to recognise situations where rollback is not necessary. By reducing the number of rollbacks, it is hoped, that the execution time of a Time Warp system will be reduced.

This paper begins by describing both Virtual Time and Time Warp. An example is then presented which illustrates how a Time Warp rollback is triggered. Next a scheme for controlling rollback, and avoiding unnecessary overhead is presented After illustrating the application of this scheme with two examples, an implementation is outlined.

2. CONSISTENT SHARING OF DATA

The concept of Virtual Time resolves the difficulties associated with sharing the variable "simulation time", or just "time", consistently. Virtual Time involves concurrent processes executing forward through time, as time stamped messages are sent, received, and processed. It requires that incoming messages be processed in timestamp order.

2.1. Time Warp

Jefferson proposed the Time Warp mechanism to implement Virtual Time. In Time Warp each process has a local clock which is synchronised with messages received from other processes. It is possible for a process to receive a message whose time stamp is less than the receiving process's clock. Such a message is called a "straggler". For this to happen, the receiving process must have chosen to optimistically execute forward by processing messages it already had, in hopes that a straggler message would not arrive. In this situation, the process must be "rolled back". The effects of any erroneously sent messages must be undone, and the new message received in the correct sequence. Rollback is accomplished by overwriting the current state with a checkpointed state, saved at a time which precedes the straggler's time stamp. Rollback is invisible to the process, which sees Virtual Time monotonically increasing.

Other less optimistic types of synchronisation need to wait whenever a message "might be" received from another process. Time Warp involves gambling that all messages stamped with times earlier than its clock value, have already been received, or are waiting in its input queue (the list of messages that have arrived) to be processed. This gamble is often successful. If a straggler *is* received, the resulting rollback incurs significant overhead.

One difficulty with rollback is that the process may have sent messages to other processes while erroneously executing forward. As the process rolls back it must undo the effects of these sends. This is accomplished in Time Warp by sending anti-messages, to remove the erroneously sent messages from the other process's input queue. If the destination process has already processed the erroneous message, receipt of an anti-message will cause it to rollback also.

Time Warp provides a method by which two processes can communicate and be assured that they are at the same point in time. This gives the illusion that all processes are in one context, with only one copy of the Time variable, just like in a sequential simulation.

2.2. A Resource

Consider a server process in a Time Warp system which is maintaining control of a resource. Clients request mutually exclusive access to one unit of this resource by sending a message to the server. The same client sends a different type of message to the server when the resource is no longer required. Resource units are anonymous.

Let a message be a tuple, consisting of a time and a message type. In reality, it may also include a sender, a receiver, and more complicated contents. We have two types of requests coming to this server:

1) (Time,ACQ) - acquire a unit of the resource.
2) (Time,REL) - release a unit of the resource.

The state of the server process at a particular Virtual Time is simply the number of units available. When an Acquire message arrives, the server receives the message, decrements the state variable, and responds to the client. If the state is 0, the server will block the client and queue it until a unit of the resource becomes available. When a release message arrives the server receives the message, increments the state variable, and responds to the client. We have now defined a complete abstract data type, encapsulated in this server process. There is no problem understanding the logical flow of events at this server, since it is in a Virtual Time system where messages are defined to arrive only in timestamp order. However, during execution of a Time Warp simulation, other orders may occur. This will not affect the server level logic, since Time Warp will eventually straighten out the order. The final outcome, after discarding any temporary effects of erroneous forward computation, will be the same as if we were running a sequential simulation.

An example sequence of events for this resource is shown in Figure 1. If we have 5 items at time 10, and an Acquire message is received at time 15, the state variable is decremented to 4 and the local clock becomes 15. As long as requests continue to arrive in the correct sequence everything is fine. However if another request arrives at time 12, Time Warp will roll the state back to time 10 (with 5 items), then redo the initial receive. Since the message at time 12 has now arrived, it will be the one received by the server. If the message was an Acquire, the state variable goes to 4 and time goes to 12. When the next message is received the variable becomes 3 and time goes to 15.

3. CONTROLLING ROLLBACK

The Time Warp mechanism is often overly cautious, and will rollback in situations that do not call for it. The previous example points out this problem. It caused the server to roll back

State (Units)	Virtual Time	Input Queue (* denotes a read message)
5	10	{ }
5	10	{ (15,ACQ) }
4	15	{ (15,ACQ)* }
4	15	{ (12,ACQ), (15,ACQ)* }
		rollback!
5	10	{ (12,ACQ), (15,ACQ) }
4	12	{ (12,ACQ)*, (15,ACQ) }
3	15	{ (12,ACQ)*, (15,ACQ)* }

Figure 1. Resource Server Execution Trace.

and deal with the straggler message, and eventually arrived at state 3. Clearly, the rollback from time 15 to 10 was not necessary.

If the server had been allowed to process the straggler without rolling back, the state would still have been 3. We realise that noticing small inefficiencies such as this is difficult or impossible in general. However, if we have an adequate description of the abstract data type being modelled by the process, it may be possible to notice many such optimisations, and avoid needless rollback.

3.1. A Rollback Function

A Rollback Function can be designed which incorporates properties of the abstract data type, and tests its state to determine whether a rollback is required. The rollback decision can often depend upon the history of the process, so our Rollback Function may need to examine the list of checkpointed states of the abstract data type. It may be necessary to examine the lists of pending, sent, or already processed messages. ***In general, access to the entire state of the abstract data type (including the state that only the Time Warp kernel knows about) is required to best determine a course of action.*** This suggests that the Rollback Function must be installed into the Time Warp kernel. A different function will be required for each abstract data type. Therefore the Time Warp kernel can be implemented with a hook in place, where the abstract data type designer can leave this description of the actions needed to more precisely determine when rollback is necessary.

The Rollback Function, when passed a straggler message must determine how far the process must be rolled back in time. It "knows" what effects the message will have, since it has access to the properties of the abstract data type. No rollback is required if those effects do not change any part of the state which is subsequentially changed through the actions of a second message. The state resulting from the effects of the new message will be the same as if all messages were processed in order. However, the process must roll back if it is determined that the straggler message will cause changes which affect a part of the state already modified by a second message received at a later Virtual Time. The effects of the later message must be undone, this message must be processed, then the later changes may be made.

Certain operations triggered by a message may be transitive. This means two such operation may be performed in either order (eg. increment and decrement). This is why the resource example above is able to interchange messages and have the state variable remain correct.

3.2. Pseudo-Rollback

We are left with another difficulty. If the Rollback Function determines that rollback is not required, then ***at what time is the straggler processed?*** The message cannot be processed at the current Virtual Time, as that would imply receiving the message at the wrong time. We must set the Virtual Time of the process to the time stamp of the straggler message, even though we need not roll back the rest of the state. After we have finished the operations associated with the straggler message, the clock may be returned to the correct value. This ensures that any resulting output messages are generated at the correct time.

We have defined a Time Warp system which allows us to study alternatives in a range less restrictive than formal Time Warp. By varying the Rollback Function we can be more or less rigorous with the message orderings, and observe the effects on performance. It is easy to implement the Rollback Function in a way which specifies formal Time Warp semantics, or with no restrictions, and have messages processed in the order they arrive. The latter approach may enable the re-use of the simulation software in the target environment [3].

We are confident that in many situations over half the rollbacks required by Time Warp can be avoided, e.g. if adjacent pairs of messages are interchangeable without effect on the state of the abstract data type. In the worst case it will act exactly like Time Warp, and in the best case with a contrived abstract data type, there may not be any rollbacks.

4. EXAMPLE ABSTRACT TYPES

Let us further illustrate the reasoning behind the proposed optimisation, and the associated Rollback Function, with an abstract data type common in all discrete event simulations - the FIFO queue. After that we will discuss a simple readers/writers problem.

4.1. Abstract Data Type for a Queue

First, the data structure and the operations which can be performed on that structure are defined. A server process will manage the state of the queue and its contents, it will receive requests, and depending on these requests, it will change the state of the data structure. The legal messages will be:

1) (Time,PUT,Object) - enter an object onto the queue, after which the server replies "(Time)" (i.e. the null message).

2) (Time,GET) - wait until the queue is non-empty, then retrieve the first object, after which the server replies "(Time,Object)".

It is obvious that a PUT and a GET will never conflict, regardless of their relative time order, as long as there is something in the queue. If a GET was performed at time 10, and was successful (ie. it waited until the queue was non-empty, then retrieved the first object), then a PUT at an earlier time (say time 5) should not cause the abstract data type to rollback. We can foresee that no conflict would arise if we did not rollback, but simply put the earlier object into the queue.

However, two GETs must be properly ordered, to ensure fair access to the queue. If they occur at the *same* Virtual Time either order is reasonable. Two PUTs must be properly ordered as well, to ensure correct output order later, since objects are not anonymous.

Another concern arises when the queue becomes empty. If we have an empty queue at time 5, then have a PUT at time 10, then a GET at time 6 at a later real time, we must not simply reply at time 6 with the object, as it is not really there at that Virtual Time. We must increase Virtual Time to 10, and then reply (ie. wait until the object is there).

The function in Figure 2 provides the logic which decides if rollback is needed and how far to roll back when it is necessary. It is called as each new message arrives. The Data Type is rolled back to the correct state, so that when it receives this new message, it is in the correct state to deal with the operation.

4.2. Readers/Writers

The Readers/Writers problem provides a second common example that can benefit from our modified Time Warp system. The data structure in this case will be a file, which is modified completely when written to, and read completely when read from. There is no priority between read and write operations on the file. The file is managed by a server process, to which all requests come in the form of messages. Thus we have the two operations:

1) (Time,READ) - which is responded to with "(Time,File)", and
2) (Time,WRITE,File) - which is responded to with "(Time)" (i.e. the null message).

The optimisation which we would like our system to allow us, is that when a READ comes out of order, it will not cause rollback unless the file has been modified by a WRITE between the Virtual Time when the READ should have arrived and the current Virtual Time of the file server. WRITEs on the other hand, must be exactly ordered. Therefore we have the Rollback Function shown in Figure 3.

This allows a reader to READ if no WRITE occurred since the time the READ should have arrived, and now, no matter how many READs have occurred since then, nor at what time they occurred.

```
Rollback_Func (new_mess)
{   if (Clock <= new_mess.time) {
        /* Advance Time */
        Clock = new_mess.time;
        return(Clock);
    }
    if ∃ M,  M ∈ Input_Queue,
             M.type == new_mess.type,
             M.time > new_mess.time,
             M.time <= Clock {
        /* Do we have a conflicting message
           of the same type, at a later time? */
        Clock = (earliest M).time;
        rollback(Clock);
        /* Rollback to before conflicting message */
        Clock = new_mess.time;
        return (Clock);
    } else {
        /* Only Pseudo-rollback is necessary */
        Clock = new_mess.time;
        return(Clock);
    }
}
```

Figure 2. Queue Rollback Function.

```
Rollback_Func(new_mess)
{   if (new_mess.type == WRITE){
        if (Clock > new_mess.time) {
            /* Rollback required */
            Clock = new_mess.time;
            rollback(Clock);
            return (Clock);
        } else {/* Advance Time */
            Clock = new_mess.time;
            return(Clock);
        }
    }else
    if ∃ M,  M ∈ Input_Queue,
             M.type == WRITE,
             M.time > new_mess.time,
             M.time <= Clock{
        /* roll back to just before
           first offending write */
        Clock = (earliest M).time;
        rollback(Clock);
        Clock = new_mess.time;
        return (Clock);
    }
    else {
        /* Only Pseudo-rollback required */
        Clock = new_mess.time;
        return (Clock);
    }
}
```

Figure 3. File Server Rollback Function.

5. IMPLEMENTATION

During the past two years we have been involved in developing a Time Warp System [4] built on Jade [5] and have acquired some experience in applying Virtual Time concepts to distributed simulation. Experience obtained while implementing this version of Time Warp is being applied to the implementation described below.

We are developing our system in C^{++}, as it provides on object oriented language, easily expandable to a usable coroutining system, which in turn is necessary for most discrete event simulation systems. This approach allows the kernel to easily access the required state information for each process, and gives the user reasonable liberty of local and global variable declaration. We are also attempting a memory management scheme which will associate dynamic memory allocations with the correct originating coroutine, in order to be able to access the coroutine's full state information.

A checkpoint of a process's state is saved just before a time transition between two Virtual Times. In this way, a process can be rolled back to deal with another message at the same Time as the message before the transition, or it can be rolled back to deal with a message between the two times of the transition. This obviates some state saves which would occur between messages at the same time (no time transition).

C^{++} also gives us quick and reliable interfaces to known and tested interprocess communication mechanisms available from the C language, namely JIPC [5] and TCP/IP, as well as any of numerous others, available to the intended user.

5.1. Structure

The system has a user and an operating system level. The user level consists of a collection of user defined processes which are able to make Time Warp system calls. Also running at the user level are predefined server processes (for such things as name searching). The operating system level is subdivided into the Time Warp kernel and the Inter-Process Communication level. The Time Warp kernel consists of a set of user accessible routines for message passing and process creation, as well as a process scheduler. The IPC level gives the kernel the ability to communicate with other Time Warp kernels, so that we can have a distributed collection of kernels.

As an implementation detail let us state that the operating system and user processes will initially be contained in a single Unix process. Interprocess communication will be with other such processes on the same or another Unix host. Eventually, they will be running alone on separate nodes of a parallel architecture. Having a layered approach gives us the ability to implement the kernel using various IPC's on different machines.

5.2. Features

We desire process creation and destruction in the tradition of many sequential simulation languages, although it raises many otherwise avoidable problems, such as name servers and exit servers which help avoid rolling back these difficult actions.

Eventually we will be running our processes on small microprocessors, (nodes in a multiprocessor), so a bounded process or kernel size is required. This raises issues, such as what happens when there is no room to send a message to another kernel. We will garbage collect unneeded states and input/output queue messages that are no longer needed for rollback.

We will implement anti-messages such that they are not sent until it is determined that the replacement messages are different than the old messages. If the re-calculated messages are the same, why send a negative, then the same positive message? This is known as lazy cancellation, and relies on the fact that occasionally the target process is sent the correct message even before the corrective rollback and re-execution.

6. SUMMARY AND FURTHER WORK

We have discussed three separate examples where 'strict Time Warp order causes unnecessary rollback. We have proposed a mechanism which reduces those rollbacks. This mechanism uses certain properties of the abstract data type being modelled by the process.

We would like to investigate methods of automatically determining these properties and providing the necessary descriptions of the offending cases. This would free the abstract data type designers from having to consider Time Warp.

We would like to apply this technique to a larger scale application and empirically compare its performance to an execution where the rollback semantics parallel those of Time Warp. Evaluation experiments with a highly parallel architecture are also needed to demonstrate the system's efficiency.

We wish to further refine methods of calculating the correct time for pseudo-rollback situations. A possible alternative to calculating this Time, and temporarily setting the Clock to that value, is to have a simplistic rollback scheme, and a method to jump forward in Virtual Time, back to the point of at which pseudo-rollback became necessary.

REFERENCES

[1] Jefferson, D.R. (1985) "Virtual Time", ACM Transactions on Programming Languages and Systems, 7(3), pp.404-425, July.

[2] Herlihy, M. (1986) "Optimistic Concurrency Control for Abstract Data Types", Fifth Annual ACM Symposium on Principles of Distributed Computing, pp.206-217, Calgary, Alberta, August.

[3] Lomow, G.A. and Unger, B.W. (1985) "Distributed Software Prototyping and Simulation in Jade", Canadian Journal of Operational Research and Information Processing, 23(1), pp.69-89, February.

[4] Xiao, Z., et.al. (1986) "A Virtual Time System Built on Jade", Research Report 86/242/17, Department of Computer Science, University of Calgary, September.

[5] Software Research and Development Group (1985) "Jade User's Manual", Research Report, Department of Computer Science, University of Calgary, October.