A Parallelizing Compiler and Run Time System based on Optimistic Execution

Adam Back and *Stephen J. Turner*

Parallel Systems Research Group
Department of Computer Science
University of Exeter
Prince of Wales Road
Exeter EX4 4PT England

Abstract

Most parallelizing compilers to date have been based broadly on compile-time data flow analysis. This reliance on static information imposes an overly restrictive view of the parallelism available in a program. The static approach cannot in general provide exact data dependence analysis and must therefore adopt the worst case view: if it is possible for a data dependency to occur, then it must be assumed always to occur. This means that sections of code will be treated as sequential which might otherwise have been parallelized.

The use of optimistic execution techniques is seen as a new direction in parallelization technology. Optimistic methods such as the "Time Warp" mechanism have been successfully used in parallel discrete event simulation. In this paper, we discuss the use of optimistic execution as a way of parallelizing programs written in a general purpose programming language. We present the design and implementation of a compiler and run time system which uses optimistic techniques to execute "sequential" C ++ programs on a multicomputer architecture.

Keywords: parallel programming, parallelizing compiler, data dependence, optimistic execution, Time Warp, virtual time.

1 Introduction

Most parallelizing compilers to date have been based broadly on compile-time data flow analysis. That is, their selection of program constructs to parallelize is based purely on data flow information obtained by examining the source code at compile time. It is not in general possible to predict actual data flow at compile time, and such an approach must therefore indicate the worst case data flow paths in the program. This is a direct consequence of the fact that it is not generally possible to predict the branches taken in a program. This problem is exacerbated by the aliasing problems in modern languages. Consequently, few parallelizing compilers try, or are very successful at, predicting program execution outside of a certain few narrowly defined constructs.

Early vectorizing FORTRAN compilers were amongst the first parallelizing compilers. The techniques used were mostly centered around parallelizing "DO loops" [14]. This technique works well for the narrow range of applications where the loops have independence between different iterations, or where transformations can be performed to produce usable independence. For example, parallelizing matrix operations is an approach which works well on vector processor based parallel computers. However, the vectorizing approach only covers a particular range of problems and there are many scientific, and conventional, applications which do not fit well on vectorizing hardware.

The vector architecture has lost favour in recent years, with distributed memory multicomputers or shared memory multiprocessor workstations becoming more widely used. Also improvements in communications bandwidth have meant that the division between parallel architectures and networks of workstations is becoming increasingly blurred. With the availability of cheap commodity RISC processors, few parallel computer manufacturers are still using custom parallel processors.

However, these hardware trends have not been followed by corresponding improvements in parallel programming methodologies. Few would disagree that parallel programming is more difficult than sequential: there is the extra dimension of dealing with many parallel processes and their synchronization, and the ensuing dead-lock, live-lock, and communication problems. In many cases, the parallel programming options available are communications libraries (for example, PVM [11], p4 [4] or MPI [8]) which support a message-passing style of programming, or extensions to existing languages such as in High Performance Fortran [7].

For this reason automatic parallelization is an important area. The motivation for using optimistic execution techniques in parallelizing code stems from the restrictions associated with compilers based on static data dependence analysis alone. Where predictions are not possible due to either uncertainty or artificial data dependencies the worst case view must be adopted. This means that sections of code will be treated as sequential which might otherwise have been parallelized. Optimistic execution schemes are able to cope with *dynamic* data dependencies, for example where one section of code has a conditional dependency on another.

The aim of this paper is to present a design for a general purpose optimistic parallelization scheme, suited to object oriented languages, and to show the feasibility of this approach by describing the implementation of a compiler and run time system. With this approach, a parallel execution can be derived from an object oriented program automatically and this execution can be guaranteed to have identical semantics to the sequential one. Thus the programmer need not even be aware that there is a parallel implementation, the effect should be transparent.

The optimistic approach adopted is based on a technique used successfully in the field of parallel discrete event simulation (PDES). In PDES [10], there are broadly two approaches taken to ensuring that causality (the principle that events in the future cannot affect events in the past) is preserved in the simulation: conservative, and optimistic. The conservative approach [6] strictly avoids the possibility of any causality violation ever occurring. The basic problem conservative mechanisms must solve is determining when it is safe to process an event: this can lead to dead-lock if appropriate precautions are not taken.

On the other hand, the optimistic parallel execution of a discrete event simulation allows causality violations to occur, but provides a recovery mechanism which is able to roll-back and undo the effect of such causality violations. Optimistic simulations are more aggressive than conservative ones, and are thus able, in general, to exploit more parallelism. The most widely known optimistic execution technique is the "Time Warp" mechanism based on the virtual time paradigm [12] and this mechanism forms the basis of our system. Optimistic execution techniques can be used effectively on distributed memory architectures [13, 19]. Parallelism is achieved by distributing the objects over the different nodes of a parallel multicomputer, and events are scheduled on remote objects by sending asynchronous messages.

The novel parts of our compiler and run time system are concerned with the method by which a sequential object oriented program is transformed into a parallel program, and the execution scheme which allows a parallel execution semantically equivalent to the sequential one. In order to explain the transformations used in the compiler and the structure of the run time system, section 2 presents a brief discussion of the "Time Warp" mechanism as used in PDES. In section 3, the design issues and decisions are explained and an overview of the system is presented. The transformation system, time-stamp allocation scheme and run time system are then described in turn, followed by a detailed example. Finally, section 8 presents the conclusions to this research.

2 The "Time Warp" Mechanism

In discrete event simulation, the "physical" system is modelled in terms of events, each of which corresponds to a state transition of an object in the physical system. Simulation events have a time-stamp which corresponds to the time at which the event would occur in the system being modelled. A sequential simulator proceeds by taking the event with the lowest time-stamp and simulating its effect: this may be to alter the state of an object being modelled, and may also create further events which will be scheduled at some future simulated time. The simulation moves forwards in simulated time by jumping from the time-stamp of one event to the next. This is in contrast to time-driven simulation methods where time moves forwards uniformly.

In parallel discrete event simulation [10] the physical system is modelled by a set of logical processes which correspond to interacting objects in the physical system. The interactions between the physical objects are modelled by the exchange of time-stamped event messages. Each process has a logical clock which denotes the simulated time of that object. This is often referred to as local virtual time or LVT. An object's logical clock will be increased to the time-stamp on each event message as that message is processed. In this way the logical clock of each object will advance according to the time-stamps of events.

2.1 Roll-back

We must ensure that events are simulated in non-decreasing time-stamp order. To see why this is necessary, consider an object receiving a message which has an earlier time-stamp than its logical clock time (this is known as a *straggler*). This means that the message represents an event which should have been executed earlier, and events have been executed past this which could have been affected by that event. With the "Time Warp" mechanism, a roll-back will be invoked to recover from the causality violation. After the roll-back, execution resumes from that point in time along new execution paths.

An event that causes roll-back may require the mechanism to perform two actions: restoring the state of the object and cancelling all intermediate side effects by "unsending" previously sent messages [10]. The first action is accomplished by regularly saving the object's state, and restoring an old state on roll-back. "Unsending" a previously sent message is accomplished by sending an *anti–message* that annihilates the original when it reaches its destination. This anti–message may in turn cause another object on another processor to roll back.

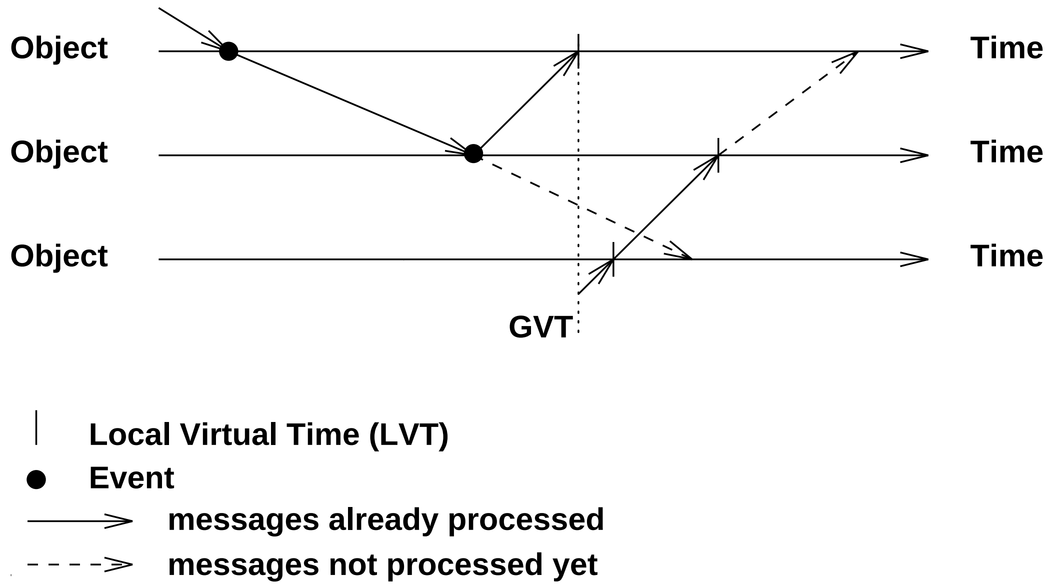

Figure 1: Virtual Time Paradigm

Each object must contain an output queue in which it keeps a negative copy of the messages it has already sent, in order to be able to send anti–messages. It must also have an input queue in which it keeps the messages it has received: some of these messages may have been processed already but they are not deleted from the queue because it may be necessary to roll back and reprocess them.

The progress of the simulation is measured by global virtual time [2] or GVT (see figure 1), which is the minimum of the logical clocks (and time-stamps of messages in transit). No event with a time-stamp smaller than GVT will ever be rolled back, so storage used by such events (e.g. saved states and messages) can be discarded. Such an event is said to be a *committed event*. GVT is also used for normal termination detection.

2.2 State-Saving

The general technique for state-saving is to save a history of the previous states of each object. Implemented simplisticly, state-saving can be expensive. Care must be taken to ensure that a workable balance is achieved and that the cost of the state-saving mechanism is not too high compared to the work done between state saving. There are two basic approaches to state-saving: periodic and incremental [16]:

1. *Periodic State-saving* is when the state of the object is saved at regular time-intervals. A complete snapshot of the object's state is saved. As the state is only saved periodically, there will be points in logical time for which no state is known. If a roll-back is required to such a point, then the object's state is first rolled back to the time of the nearest state-save prior to the state required. Then the execution is made to "coast-forward" to the required point, recomputing work up to the point of the required roll-back. Periodic state saving is a compromise: there is a trade off of state saving overhead against recomputed work.

2. *Incremental State-saving* is another state-saving technique, with different overhead characteristics. With incremental state-saving, only the parts of the state which have actually changed are saved. The changes are normally saved on *every* change, so there is a complete state-saved history. This eliminates the recomputing of unnecessary work due to missing

state information which occurs with periodic state-saving. Incremental state-saving is better in some cases, but there is also the danger that for frequently changing states, a large number of state-changes will be saved in a short time-period. There are cases where either method will win over the other.

3 Design Overview

In applying an optimistic execution scheme to general purpose object oriented programs, the essence of the approach is that selected program objects will be placed on the processors of a distributed memory parallel computer. The invocations of the methods of these objects are replaced by asynchronous remote procedure calls (RPCs). The asynchronous nature of the remote procedure calls, and the program transformations needed to enable this are an important part of the design, and are what creates the parallelism in the execution. Our compiler currently makes use of source-to-source transformations using the sage++ transformation tool [3].

Remote procedure call messages are time-stamped, in such a way that if the program were executed in increasing time-stamp order, an execution semantically indistinguishable from the sequential execution is obtained, just as executing a simulation using the "Time Warp" mechanism gives exactly the same results as a sequential simulation. An incremental state-saving memory system is used to encapsulate the behaviour of the optimistic execution system state, which will include out of sequence memory accesses.

Each of the objects placed on processors of the parallel computer acts as a *server object* processing asynchronous RPCs, in increasing time-stamp order. As no other synchronization constraints are imposed, it is a very flexible parallel execution, one capable of executing in parallel anything which is logically parallelizable, regardless of the ability to detect the exploited independence at compile time. A time-stamped message arriving out of order for a particular server object signifies a causality violation. The server object rolls back its state to just before the time-stamp of the straggler RPC message, and re-executes from that point forward.

To map sequential object oriented code on to the optimistic execution system model, a transformation system is used to transform method invocations into asynchronous remote procedure calls. Parallelism can thus be introduced into the execution of the general purpose code in a number of ways: the flexibility of the optimistic execution scheme enables different sources of parallelism to be exploited. A server object in the run time system thus provides the functionality of roll-back support based on an incremental state-saving mechanism, with the message passing system providing causality violation detection.

As in PDES, the asynchronous RPC invocations which correspond to the event messages in a discrete event simulation are time-stamped. The time-stamps are chosen such that an execution of the associated RPC invocations in increasing time-stamp order would produce an execution semantically equivalent to that of the original program. It is normal in PDES to stipulate that events be processed in non-decreasing time-stamp order, rather than in increasing time-stamp order as done here. This apparent difference is due to the use of a time-stamp allocation scheme which ensures unique time-stamps. The time-stamp allocation scheme is described in section 5.

One problem with using incremental state-saving for a general purpose programming language is that with the obvious implementation ultimately every variable or object in the entire program would need to be represented as a state-saving object. Performing incremental state-saving on low level objects such as integers, or floating point numbers, adds a significant overhead compared to the simple task performed, that of multiplication, addition, etc. Saving a previous value in these cases is many times more expensive than the operation itself, especially as the operands will typically be stored in registers, while the state-saved information will have to be stored in memory.

Taking advantage of the object oriented nature of the run time system, our approach to reducing this overhead is to adopt a hybrid scheme. We have provided a system which, though still incremental, performs the state-saving on objects at arbitrary levels in the object oriented class hierarchy. This allows states to be saved less frequently than with a full incremental scheme if this would be desirable. The choice of level at which to perform the state-saving can be made according to the characteristics of the object.

4 The Transformation System

4.1 Asynchronous Messages

Object oriented programming has a natural message passing implementation: each method invocation can be represented as an RPC consisting of an invocation message, and arguments sent with that message. The return value, and any modified parameters are then sent back as a result message. Communication in a message passing implementation of a sequential object oriented program is therefore synchronous: the sending object, when there are return values, waits for those values. In the special case that there are no return values, the invoking message is typically synchronous also, as an acknowledgement message would be sent back to indicate that the remote method invocation has been completed.

The aim of the transformation system is to transform this object oriented view so that it can be implemented with a PDES like optimistic execution scheme. This sub-section describes a transformation of the above message passing mechanism so that it can be implemented using asynchronous messages only. This is required as the aim of the system is to introduce parallelism: thus there is no need for the sending object to wait for a return value or acknowledgement message indicating completion of the remote procedure call.

In the situation where the invoked method has no return value, it is easy to see how this can be implemented using asynchronous message passing. The problem that remains is that of dealing with return values without blocking. The basis of the transformation is the use of the fact that the destination of the return value is known before the remote procedure call is invoked. To understand this mechanism, it is first necessary to explain what it means for an object oriented programming language to be "pure".

In a pure object oriented language there are no objects outside the object oriented type system: all objects support the inheritance mechanism and even the existing primitive objects, such as integers, are considered first class objects and the operations on them as method invocations. Some object oriented languages deviate from this pure view for efficiency reasons: C ++ is an example of such a language. In C ++ the built-in types are treated as a special case, and are outside the object oriented type system for the purpose of inheritance.

In order to apply the transformations for asynchronous messages, it may be necessary first to "purify" the program. This can be achieved by simulating the built-in types by providing replacements which are specially defined: these will allow inheritance, and can be constructed to have all of the required operations available as methods. If the program is first transformed to use these replacement objects, then the whole program can be viewed for asynchronous message passing purposes as a pure object oriented program. The purifying transformation need not necessarily be applied to the whole program, but just to selected areas to enable the transformation described below to be performed.

The asynchronous message transformation is only concerned with methods which have a return value. In a pure object oriented program, the return value will by definition be passed as a parameter to a method of another object. The way this transformation is used in the run time system is to pass, along with the arguments for the remote object method invocation, instructions for where the result should be sent. For a simple case, such as an assignment statement, the returned value would be sent to the assignment method on the object which was on the left hand side of the assignment statement in the untransformed program.

Consider the following small example with three matrix objects, the invert member function returns the inverted matrix.

```
class Matrix;

{
  Matrix A, B, C;

  A = B.inv();
  :
  A = C.inv();
}
```

This is transformed to:

```
class Server_Matrix;

{
  Server_Matrix A, B, C;

  B.inv( A, assign );
  :
  C.inv( A, assign );
}
```

where `Server_Matrix` is a server object with methods which are invoked as asynchronous remote procedure calls (RPCs). To avoid waiting for the returned matrix, the invert operation is given the server object and the method to invoke with the returned value. That is, the main thread of control continues immediately after dispatching the RPC to server object `B`. The server object `B` receives the invert RPC message and when it has finished, sends the inverted matrix (the return value from the invert member function) to server object `A`, requesting that it be assigned to `A`. A diagram of the server objects in the example and the asynchronous RPC messages passed between them is shown in figure 2.

This pattern of communication is used in the transformations as it is not known which message will be sent to a server object next. Several objects, distributed over the processors, may send a request to a server object, each with a different time-stamp. In the transformed code, the server object `A` does not wait for the message representing the return value from the request to invert matrix `B`, because the return value from the request to invert matrix `C` may be returned first. The matrix `A`, and server objects in general, act as servers only: all messages are handled as RPC requests.

Another problem, which might be apparent, is that in some cases the returned value is actually required immediately for further computation to proceed in the invoking method. The solution here is to split the invoking method into sub-methods, each acting as an entry point, and with the required value which would otherwise cause the object to block as an argument. The invocation which computes the value is sent instructions to invoke the next sub-method. In this way the server objects are kept simple: they need only be single threaded.

It might seem possible to extend the model to include multi-threaded server objects, but this would significantly complicate the design when the full optimistic execution scheme is

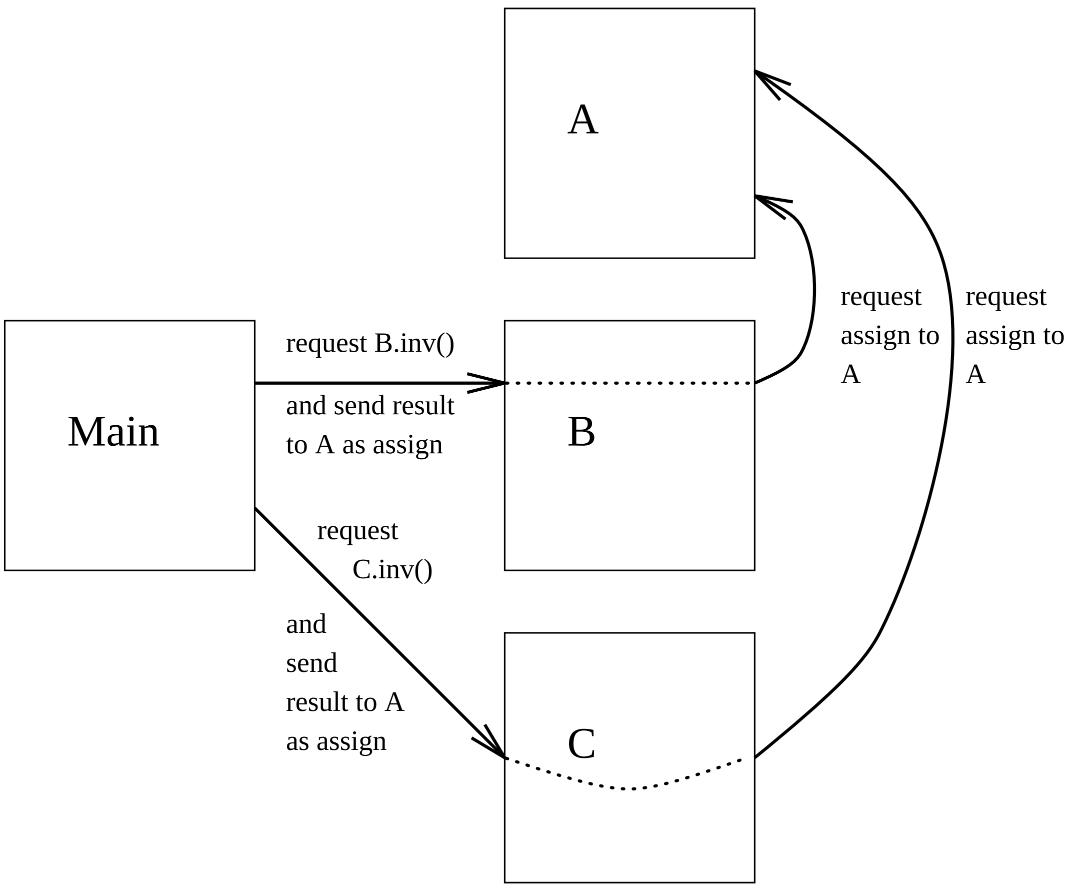

Figure 2: Server Objects for Matrix Class

Figure 3: State Saving Options

considered. The extra server object functionality of having a logical clock, and a state-saving mechanism, means that the object would in effect have multiple sub-parts with possibly different logical clock values. The desired parallelism which could conceptually be exploited by having a multi-threaded server object can in any case be obtained by having more server objects, that is, by making the component objects of the server object themselves server objects.

To summarize, the aim of the object transformation system is to allow an asynchronous implementation of the object oriented programming view. This is achieved by having each method invocation implemented as an asynchronous message dispatch. The invoking server object can then continue execution or wait to receive other RPC messages. It is important that the server be ready to receive any message at this point because, in an optimistic execution scheme, it will not be known which method invocation will arrive next. In the parallel execution of the system, methods can be invoked out of normal sequential order.

4.2 State-saving

Each object either knows how to save its own state or relies on the state-saving abilities of its contained objects. There are straight forward transformations which will produce a state-saving object from a conventional programming language object. A state-saving object may be defined as a replacement object with the same functionality as the original object, but which automatically saves its state as it is used, and with extra methods to invoke roll-back. State-saving objects are created by the transformation system, and are transformed from the equivalent non state-saving object.

As all objects are built from component objects, the selection of state saving mechanism is made by choosing between building the object from the functionally equivalent state-saving objects, or from unmodified objects. The selective use of state-saving objects at various levels in the class hierarchy, and with containment, allows for a hybrid system, which is capable of varying its characteristics between purely incremental state-saving, and more periodic behaviour. For example, figure 3 shows the two possibilities when an object `A` contains two objects `B` and `C`. Either `A` can be composed of state-saving objects `B` and `C`, or `A` itself can be state-saving and `B` and `C` are untransformed objects.

There are a number of parts to the transformation: firstly parameterizing by time-stamp, then adding the calls to the raw state-saver at the end of each method, and finally choosing to use either transformed state-saving objects, or untransformed objects, for composition to build hybrid state-saving objects.

The parameterizing by time-stamp transformation is basically to add an extra parameter to each of the member functions of the object, so that when it is called its time-stamp is passed to it, and the state-save at the end of the method is given that time-stamp. The main complication is concerned with overloaded operators which are called implicitly, or explicitly. C ++ considers overloaded operators to be just ordinary functions, but there are restrictions on the number of parameters they may have, to fit in with their semantics.

Another problem is time-stamping destructor calls. Destructors are usually called implicitly, and in any case can have no parameters. Several other implicit parts of C ++ cause similar problems by virtue of restrictions on the number of parameters. A transformation phase called "explicify" has been designed to make explicit all such language features, and to convert them all to ordinary function calls.

After the explicify operation, the adding of an extra time-stamp parameter is straight forward. The parameter is added as the first parameter, this is necessary because C ++ has a language feature which allows optional arguments, but these must appear as the final parameter or parameters to the function. If the time-stamp parameter were not given first, additional transformations would be required to fill in the optional parameters, so that they were all given.

The next part of the transformation is the calls to the low level state-saver. The object which is being transformed is additionally transformed to derive the low level state-saving capability from the run time system component. This automatically gives it the capability of rolling back, once the appropriate calls to the state-save operation, and composite roll-back operation are added.

The constructor of the transformed object is given an extra parameter, the time-stamp of the object's creation. This is passed in to the constructor of the run time system low level roll-back base class.

Next a call to the low level state-saver function is added at the end of each method. A roll-back method is constructed for the transformed object, this method merely calls the low level roll-back method, with the appropriate size argument, and start of object argument. The size argument will be `sizeof(object)` where `object` is the object being transformed, and the start of object argument will be `this`, the C ++ reserved word meaning a pointer to the start of this object.

The above transformations modify a program level object into an incrementally state-saving object. The last part of the transformation, is the combination of base-level incremental objects, and untransformed objects into hybrid state-saving objects. Each object can be composed of or derived from combinations of incrementally state-saving objects, or raw program level objects. At present this choice is driven by annotations, but it is envisaged that in a future system, heuristics will be used to make the selection, depending on the characteristics of the objects concerned.

5 Timestamp Allocation

With the optimistic execution of a general purpose program, a problem arises which does not occur in parallel discrete event simulation: it may be necessary to allocate an arbitrary number of new time-stamps between any pair of previously allocated time-stamps. If we consider a program control structure consisting of an unbounded loop, we could be executing the individual iterations of that loop on one node whilst executing in parallel the events corresponding to the code following the loop on another node. We therefore need to allocate the time-stamps for the events following the loop before we have allocated all of the time-stamps for the iteration events.

For this reason in the optimistic parallel execution of a program the number and ordering of time-stamps is not known at time-stamp allocation time. In this scheme we would expect to make an initial allocation of a sequence of ordered time-stamps as in the simple case. After this we will be required to allocate similarly ordered sets of time-stamps which fall between pairs of previously allocated time-stamps. This process is presumed to be repeatable to arbitrary depth. A time-stamp allocation scheme which is able to meet these requirements must have the property that an arbitrary number of time-stamps can be allocated between any pair of previously allocated time-stamps.

For our representation of time-stamps we require the operations to compare and generate

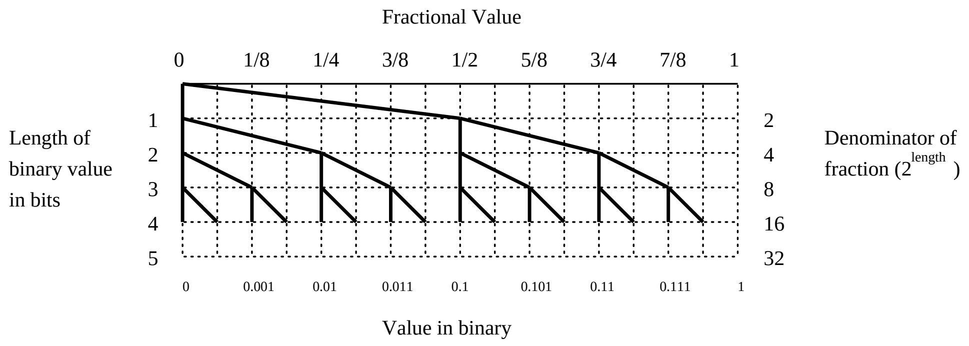

Figure 4: Variable Length Time-stamps

the next time-stamp in a sequence to be efficient. A variable length time-stamp has therefore been adopted which satisfies these requirements. We define our variable length time-stamps:

$$\text{a time-stamp is the pair } (\mathit{length}, \mathit{value})$$

where *value* is a binary number of length *length* whose value will be non-negative. Time-stamps can be viewed as variable length binary fractions, whose values fall in the range 0 to 1, as shown in figure 4. The fractional value of a time-stamp (l, v) is

$$f = \frac{v}{2^l}$$

We define an ordering relation on time-stamps:

$(l_a, v_a) = (l_b, v_b)$ iff $f_a = f_b$ and $l_a = l_b$
$(l_a, v_a) < (l_b, v_b)$

$$\text{iff either: } \begin{cases} f_a < f_b \\ \text{or } f_a = f_b \text{ and } l_a < l_b \end{cases}$$

The time-stamps can be thought of as variable precision binary fractions. We can represent time-stamps pictorially, as the nodes of a binary tree, as shown in figure 5.

The ordering defined by the pre-order traversal of the time-stamp tree is the same as the ordering defined above. The time-stamps which fall between a pair of consecutive time-stamps of a particular length are the descendants in the time-stamp tree of the first time-stamp.

The left child of time-stamp (l, v) is $(l + 1, 2v)$ and the right child $(l + 1, 2v + 1)$. More generally at depth d below l, the set of time-stamps at level $l + d$ which fall between (l, v) and the consecutive time-stamp at the same level $(l, v + 1)$ is given by:

$$\begin{aligned} &(l, v) < \\ &\quad \left\{(l + d, 2^d v) < \ldots < (l + d, 2^d v + 2^d - 1)\right\} \\ &\qquad < (l, v + 1) \end{aligned}$$

Informally, for time-stamps with different fractional values, integer compare can be used on the *value* component to correctly order the time-stamps. However, if the fractional values of a pair of time-stamps are equal, the shorter time-stamp is taken to be smaller. Only if both the fractional values and the length of time-stamps are equal are the time-stamps considered equal, that is, trailing zeros become significant in the ordering, so that $0.10 \neq 0.1$. A full discussion of the time-stamp allocation mechanism can be found in [1].

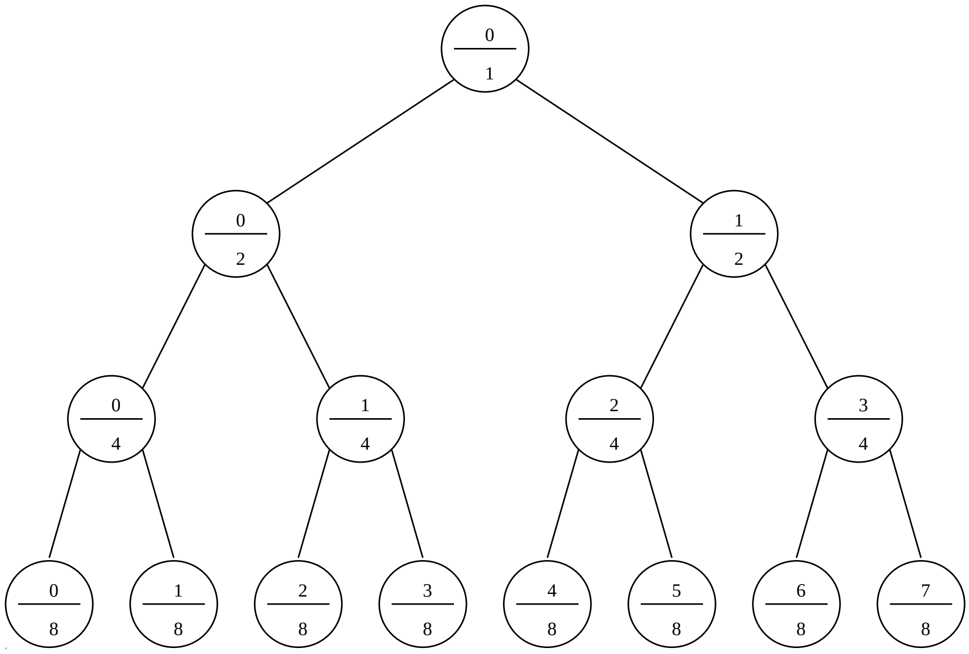

Figure 5: Tree Representation of Time-stamps

6 Run Time System

6.1 Server Objects and Roll-Back

A server object is a remote procedure call (RPC) server for an object. It has a single main thread of control which accepts RPC invocation messages, decodes the message to decide which method was invoked, decodes the methods arguments, and executes that method. As described above the transformation of the program object has ensured that the server object has a disjointed set of sub-methods for each of the methods it had prior to transformation, which can be invoked asynchronously. The code associated with handling a return value will also be a sub-method of an object.

A server object is a transformed C++ object which is placed on a node of the parallel computer. Its method invocations will have been transformed into time-stamped messages. Server objects are the level at which roll-back is managed in the system. Each server object has a logical clock which is set to the time-stamps of RPC messages as they are processed. In this way the logical clock increases in discrete steps with the arrival and processing of RPC messages.

Broadly speaking (there are exceptions, optimizations which will be outlined below) a causality violation is signalled by the arrival of a straggler message, a message with a time-stamp lower than the server object's logical clock. A causality violation indicates that the RPC should have been processed earlier in logical time than RPCs already processed by that server object. This invokes the roll-back mechanism: if the RPC relies on prior state changes of the object, and will read and modify the object, the server object must be rolled back, and the RPCs re-executed in the correct order.

A number of optimizations from parallel discrete event simulation can be applied to the server object: lazy cancellation, lazy re-evaluation, and avoidance of roll-back for methods which do not modify the object's state. If a causality violation is detected, and it cannot be avoided by one of the optimizations, the state of the object must be restored. For roll-back, prior state information, a history of values must be retained, as provided by the incremental state-saving mechanism discussed in section 4.2.

6.2 Message Passing System

Server objects are distributed across the different processors of a distributed memory parallel computer. The message passing system is required to transport the messages between server objects. To facilitate parallel slackness [20], many objects are placed on each processor, so the message passing system must also be able to handle messages between processes on the same processor. Messages are time-stamped in accordance with the requirements of the optimistic execution scheme.

The message passing system was built on top of the p4 message passing library [4]. The p4 model was chosen as a portability layer [18], to ensure that the system could be ported to a wide range of parallel computers. A description of the p4 library is given in [5]. It has the functionality of tagged messages, where messages are buffered at the receiving process. Messages may therefore be received out of order, the call to receive a message has as a parameter the tag type to receive. Unfortunately, p4 also has the restriction that dynamically allocated processes cannot receive messages.

The processes in an optimistic execution system are required to be created dynamically. The message passing system we have built on top of p4 allows dynamically allocated processes to exchange messages. In addition to this functionality, the messages are time-stamped, and logged on sending. The messages are received in increasing time-stamp order, which means that they may potentially be received in a different order to that in which they are sent even if they have the same tag. This is to facilitate causality violation detection by the optimistic execution scheme.

Messages could actually arrive out of order, and be processed in order, as messages are sorted by time-stamp on arrival. A causality violation is said to have occurred when the next message available for processing has a lower time-stamp than the logical clock of the server object receiving the message. There is code in the message passing part of the run time system which checks for this situation, and invokes the roll-back mechanism and sending of anti-messages. The state is restored to a point in logical time just before the time-stamp of the straggler message.

For the sending process, messages are logged, so that there is information for the sending of anti-messages (and to compare against for lazy re-evaluation). If a causality violation is detected, an anti-message can be sent by just sending the logged time-stamps of the messages to cancel. This is an optimization proposed by Rönngren [17]: only the time-stamp is required, as this is sufficient for cancellation where time-stamps are unique. The uniqueness of time-stamps is a feature of our time-stamp allocation scheme, but is not generally the case with optimistic PDES systems.

6.3 Optimizations

One optimization of this scheme is lazy cancellation [10]. This optimization stems from the fact that sometimes, even though a message arrives late, no difference in sent messages results. If this turns out to be the case, the anti-messages may have caused unnecessary roll-backs. The way that this optimization would be implemented would be to delay sending the cancellation messages when the object is question is rolled back. The input messages are then reprocessed and a check is made to see if any messages that would be sent are different to the previously logged messages from this object.

If there are no differences in the output messages, the anti-messages do not need to be sent, and a possible cascade of roll-backs will have been saved. However, if the messages do turn out to be different, at this point the anti-messages are sent. Apart from the work involved in comparing messages, the negative aspect of this optimization is that the objects for which the anti-messages were destined will have had time to do more work: work which will need to be rolled back.

Another optimization to the optimistic execution scheme is lazy re-evaluation [10]: this says that even when an object has been rolled back to take account of a straggler message, it is not certain that the object's state will be affected. For lazy re-evaluation, execution and the state-saving system is switched to a different mode: one where the state-saving operations compare against the already existing saved state. If a difference in state is observed, the state-saving mechanism is switched to normal saving mode, and the possibility of lazy re-evaluation is abandoned.

If, however, no differences in state are detected during the re-evaluation of the code corresponding to the straggler message, this implies that a complete re-evaluation can be avoided. The system jumps forward over the rolled back events without re-executing them as their re-execution would be identical to their original execution. A special case of lazy re-evaluation which is particularly significant involves the handling of read-only methods. Here, it is known that the state will not be affected and a straggler message for a read-only method may be always be processed without re-executing the rolled back events [9].

6.4 Memory Management

The run time system also contains a specialised set of memory management routines, which have the property of being able to undo memory management operations. This is required for the implementation of the optimistic execution scheme, because it is necessary to be able to roll-back memory management operations as well as all other program constructs.

In C ++ nomenclature, there are two memory management operators `new`, and `delete`, which provide the operations of memory allocation, and of freeing the previously allocated memory. For purposes of roll-back, the roll-back of an allocate operation requires only the freeing of the memory. The roll-back of a free operation, is however more complex than re-allocating it. To unfree a previously freed memory area requires that all of the previous values be retained.

The way that this is implemented is not to actually delete the memory area when it is freed, but rather to just mark it as pending deletion. In this way a roll-back of a free operation can be implemented by just removing this mark. The actual deletions, which are necessary for the heap not to be used up quickly, are delayed until GVT. When the time-stamp of the deletion is passed by GVT, the space can be safely reclaimed, as it is known that there will be no roll-backs to a time before GVT.

6.5 Input/Output and Exceptions

Input and output are also delayed until GVT. This is implemented by having special versions of the C ++ classes in the run time library which have the property of delaying processing until GVT. This ensures that they will never be rolled back. Although it may be possible to roll back input and output to a limited extent with non-interactive programs, in the general case, neither input nor output can be rolled back.

Exceptions, in the sense of operations which cause the program to halt with an error condition, also need to be delayed until GVT. This is for the reason that they may have been erroneously caused by messages being processed out of time-stamp order. A subsequent straggler message may cause the exception to be rolled back. So errors such as dereferencing a null pointer, resulting in a segmentation violation, are trapped, and held pending GVT. If the violation still exists at GVT, the exception is reported, and the program terminated.

7 Example

We consider a simple example which has a conditional dependency. Static analysis of this code would reveal that the loop has a possible dependency and hence must be executed sequentially. The optimistic execution of this loop can ignore the possible dependency and hence parallelize the loop. The loop is unbounded as we do not know at compile time how many iterations of the loop will be required.

```
void proc(double v[])
{
  for ( int i = 0; v[i] < MAX; i++ )  // loop
  {
    v[i] = func( v[i] );        // statement A
    if ( i > 0 && v[i] < 0 )    // condition
    {
      v[i] = v[i - 1];          // statement B
    }
  }
}
```

Generating time-stamps for the unbounded loop, with a value of $b = 3$:

loop	statement A	condition	statement B
0.000	0.000_{00}	0.000_{01}	0.000_{10}
0.001	0.001_{00}	0.001_{01}	0.001_{10}
0.010	0.010_{00}	0.010_{01}	0.010_{10}
0.011	0.011_{00}	0.011_{01}	0.011_{10}
0.100000	0.100000_{00}	0.100000_{01}	0.100000_{10}
0.100001	0.100001_{00}	0.100001_{01}	0.100001_{10}

Each element of array v will be transformed to become an object with a local clock. The three statements within the loop have time-stamps of the form:

$0.x_{00}$	$0.x_{01}$	$0.x_{10}$

where 0.x is the time-stamp of the containing loop, and the statement number is shown in small type.

The transformation (stylised) for this example uses server objects for the elements of the array, and has server objects which provide the functionality of roll-backable control flow statements. Parallelism is introduced into the system by the asynchronous dispatch of server object invocation messages. As described previously, messages are not waited for as this would block the server object: to avoid waiting, methods with return values are transformed to have extra parameters giving a server object to which to send the result and the method to invoke with that result.

In the example transformation there are server objects for the elements of the array `v` and for the loop index variable `i`. The `for` and `if` control constructs use the values of server objects and so must be able to roll back, this is achieved by using server objects to represent them. Server objects are used because any action which may need to be rolled back is represented, at least conceptually, as a message. The action may then be rolled back by sending an anti-message, and re-executed by reprocessing the message. A diagram of the server objects in the example and the asynchronous RPC messages passed between them is shown in figure 6.

```
class Server_Double;
class Server_Int;
class Server_If1;
class Server_For1;
class Server_Func;

Server_For1::iter1()
{                   // RPC send value to func
  i.read( func, arg1 );
  Server_If1 if1( i, v );
  i.read( if1, eval1 );      // RPC to read i
  i.operator++();       // RPC to increment i
  i.read( for1, next1 );     // RPC to read i
}

Server_For1::next1( int i_val )
{
  v[ i_val ].read( for, next2 );
}

Server_For1::next2( double v_i_val)
{
  if ( v_i_val < MAX )
  {
    iter1();         // RPC to self next iter
  }
}


Server_If1::eval1( int i_val )
{
  if ( i_val > 0 )
  {
    v[ i_val ].read( if1, eval2 );
  }
}

Server_If1::eval2(int i_val, double v_i_val)
{
  if ( v_i_val < 0 )
  {
    then1( i_val );
  }
}

Server_If1::then1( int i_val )
{                    // RPC to assign to v[i]
  v[ i_val - 1 ].read( v[ i_val ], assign );
}
```

```
Server_Func::arg1( int i_val )
{
  v[ i_val ].read( func, call );
}

Server_Func::call(int i_val, double v_i_val)
{                  // RPC send result to v[i]
  func( v_i_val, v[ i_val ] );
}

void proc( Server_Double& v[] )
{
  Server_Int i( 0 );
  Server_Func func( v );
  Server_For1 for1( i, v , func );
  i.read( for1, next1 );
}
```

In the diagram the messages are labelled with the RPC function they request, so for example the RPC message requesting the invocation of the `iter1` method of the `for1` server object is labelled `iter1`. The dotted paths drawn through server object boxes represent the messages sent as a result of invoking the requested method. We will describe the messages in the system on a per method basis. Note that the messages corresponding to the construction of the server objects are not shown in the diagram.

The main thread sends a `read` RPC message (1) to the `i` server object, requesting that the result be sent to `for1::next1`. When `i::read` sends its value back to `for1::next1` it is used to send the RPC for `v[i]::read`, with the request that the value of `v[i]` read be sent to `for1::next2`. If the value of `v[i]` given to `for1::next2` satisfies `v[i] < MAX` then a message is sent to invoke `for1::iter1`. As a result of this the `for1` server object dispatches four asynchronous RPC messages (in order of dispatch: 3, 4, 11 & 2) to the server `i`. Messages (3, 4 & 2) request the value of `i` and request the value to be forwarded to `func::arg1`, `if1::eval1` and `for1::next1` respectively. Message (11) requests that `i` be incremented.

`If1::eval1` requests the value of `v[i]` in message (13), the value of i is chosen to be 1 in the diagram so the value of `v[1]` is asked for. The request for the value of `v[1]` specifies that the result be sent to `if1:eval2`, this allows the `if1` server object to resume the logic of the if statement when the value of `v[i]` has been obtained. It may serve other RPC requests before receiving the value of `v[i]`. If the condition of the if statement is satisfied, `if1` sends itself a `then1` RPC (12), to invoke the then part of the code. The method `if1::then1` requests an assignment (15) of `v[i-1]` to `v[i]`.

This will repeat itself by sending the message (5) to the `for1` server. The test for termination is in the method `for1::next2` which sends a message to invoke `for1::iter1` if another iteration is required. In this way if an iteration of the loop turns out not to have been required, then it will be rolled back by the sending of an anti-message to the `for1` server. The disjointed nature of the methods of the server objects is required to allow asynchronous reading of the return values. It is not known which return value will come back first, so all return values are treated as standard requests and are handled by the server object as RPC requests.

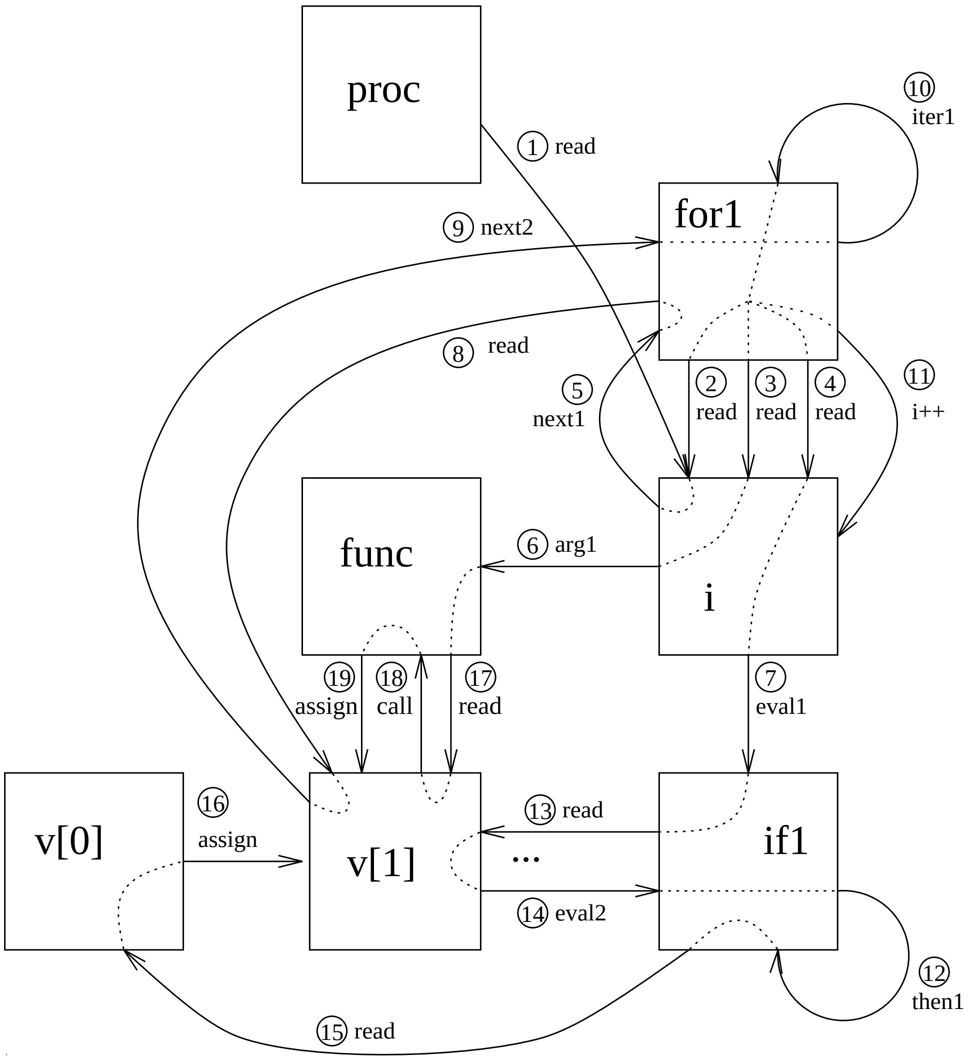

Figure 6: Asynchronous Message Passing

8 Conclusions

This paper has shown how it is possible to extend the virtual time paradigm of optimistic simulation to the execution of general purpose programs. In many ways, the virtual time paradigm is the inverse of Lamport's algorithm [15]: we assume that every event is labelled with a clock value from some totally ordered time-scale in a manner consistent with causality. A partial ordering may then be obtained which allows a fast parallel execution.

By assigning time-stamps to program control structures which are consistent with the sequential execution of that program, a conventional object oriented program may be executed in parallel using the Time Warp mechanism, with a guarantee that the same results are obtained as would be the case if the program was executed sequentially. Each object executes without regard to whether there are synchronization conflicts with other objects. When a causality violation is detected, the object is rolled back in virtual time, and is then allowed to continue along new execution paths.

A parallelizing compiler has been described which converts sequential C++ code into a form which is suitable for optimistic execution. The assignment of time-stamps to program control structures requires a flexible time-stamp allocation scheme and we have presented a solution to this problem. A run time system has been implemented which supports roll-back based on incremental state-saving. Optimizations which reduce the overhead of the Time Warp mechanism have also been discussed.

This research is an attempt to overcome the barriers imposed by the overly pessimistic view static analysis imposes on programs whose behaviour cannot be fully predicted at compile time. Although our system does not currently take advantage of existing parallelization techniques based on compile time analysis, a more complete system could also include such techniques and only make use of optimistic execution when insufficient parallelism is obtained from static data dependence analysis alone.

References

[1] A. Back and S. Turner. Time-stamp generation for optimistic parallel computing. In *Proceedings of the 28th Annual Simulation Symposium, Phoenix, Arizona*, pages 144–153. IEEE Press, April 1995.

[2] S. Bellenot. Global Virtual Time algorithms. In *Proc. SCS Distributed Simulation Conference*, pages 122–127, 1990.

[3] F. Bodin, P. Beckman, D. Gannon, J. Gotwals, S. Narayana, S. Srinivas, and B. Winnicka. Sage++: An object–oriented toolkit and class library for building Fortran and C++ restructuring tools. *Object Oriented Numerics*, 1994.

[4] J. Boyle, R. Butler, T. Disz, B. Glickfeld, E. Lusk, R. Overbeek, J. Patterson, and R. Stevens. *Portable Programs for Parallel Processors.* Holt, Rinehart, and Winston, 1987.

[5] R. Butler and E. Lusk. User's Guide to the p4 Parallel Programming System. Technical report, ANL-92/17, Argonne National Laboratory, 1992.

[6] K.M. Chandy and J. Misra. Distributed Simulation: A case study in design and verification of distributed programs. *IEEE Trans. Software Engineering*, SE5(5):440–452, 1979.

[7] High Performance Fortran Forum. High Performance Fortran Language Specification. Technical report, Rice University, Houston, Texas, May 1993.

[8] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical report, University of Tennessee, Knoxville, Tennessee, May 1994.

[9] R.M. Fujimoto. The virtual time machine. *SPAA (Symposium on Parallel Algorithms and Architectures)*, pages 199–208, 1989.

[10] R.M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

[11] A. Gest, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. PVM3 User's Guide and Reference Manual. Technical report, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, ORNL/TM-12187., May 1994.

[12] D.R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[13] JPL. *Time Warp Operating System User's Manual.* Jet Propulsion Laboratory, 1991.

[14] L. Lamport. The parallel execution of DO loops. *Communications of the A.C.M.*, 17(2):83–93, Feb 1974.

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

[16] A. Palaniswamy and P.A. Wilsey. An analytical comparison of periodic checkpointing and incremental state saving. In *Proc. 7th Workshop on Parallel and Distributed Simulation*, pages 127–134, July 1993.

[17] R. Ronngren. *Performance Issues in Discrete Event Simulation.* Royal Institute of Technology, Stockholm, 1995.

[18] S.J. Turner and A. Back. A T9000 implementation of the p4 parallel programming model. In *Proceedings of the North American Transputer User Group*, 1994.

[19] S.J Turner, M. Damitio, and S. Trivett. Distributed simulation with a transputer version of the time warp operating system. In *Proceedings of Transputers '94*, pages 39–54. IOS Press, 1994.

[20] L.G. Valiant. A bridging model for parallel computation. *Comm. A.C.M.*, 33(8):103–111, 1990.