Time–Stamp Generation for Optimistic Parallel Computing

Adam Back and Stephen Turner
Department of Computer Science,
University of Exeter,
Prince of Wales Road,
Exeter EX4 4PT, England
Email: {aba,steve}@dcs.exeter.ac.uk
Tel: +44 392 264048
Fax: +44 392 264067

Abstract

Optimistic execution techniques are widely used in the field of parallel discrete event simulation. In this paper we show that optimistic execution can also be used to parallelize program control structures.

We discuss the requirements for handling unbounded constructs and demonstrate the need for a flexible time–stamp allocation scheme.

We present a scheme using variable–length time–stamps which allows an arbitrary number of time–stamps to be generated between any pair of existing time–stamps. The ordering relation defined for these time–stamps is similar to that for fractional numbers: for two consecutive numbers of a given length it is always possible to generate a number whose value falls between them.

Optimizations which improve the efficiency of time–stamp allocation for typical program structures are presented, together with an analysis of the cost. We show that the size of time–stamps is manageable even for programs with large, complex, control structures. Finally we give an example of the use of time–stamps in parallelizing a simple control structure.

1 Introduction

Optimistic execution schemes such as *Time Warp* [12] which involve causality violation detection and recovery have been successfully used for parallel discrete event simulation. The aim of our research is to investigate the use of such techniques for the execution of general purpose programs. We are using the Time Warp mechanism as the basis of an automatic parallelization tool.

The most important principle that is encountered in parallel discrete event simulation is the *causality principle*: "The future cannot affect the past". Whereas the conservative approach [7] strictly avoids the possibility of any causality error ever occurring, optimistic methods detect and recover from causality errors. The Time Warp mechanism, based on the *virtual time* paradigm, is the most well-known optimistic protocol. This allows a simulation object to advance in time with no regard for the risk of having its simulated past affected. If its past is changed (due to interaction with an object further behind in virtual time), it must be able to *roll back*, and then be allowed to continue along new execution paths.

An event that causes rollback 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 periodically saving the object's state, and restoring an old state on rollback. "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.

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 objectives of our research into the use of optimistic mechanisms in general purpose computing are to make use of more of the available parallelism than is possible with automatic parallelization schemes based on static program analysis only. Some work has been done by Bacon [2] on the optimistic execution of CSP (Communicating Sequential Processes) [11], but the use of optimistic execution as a parallelization tool has been largely unexplored.

We can take any general purpose computation and split the program into blocks of code which will correspond to events. This idea forms the basis of our research [18] into automatic program parallelization: by assigning a time–stamp to each program control structure, the program may be executed in parallel using an optimistic mechanism, but will give the same results as it would do if the control structures were executed sequentially in strict time–stamp order, in the same way as a parallel simulation using the Time Warp mechanism gives the same results as a sequential one.

The optimistic execution of a program can parallelize some code which static program analysis would indicate was sequential. The conservative approach always has to assume the worst case: if it is possible for a branch to be taken it would assume that it is always taken. The optimistic approach can execute more of the program in parallel, but it also means that it may do some work which it later decides is wrong because of a causality violation.

The areas for research are in reducing the overhead of maintaining the roll back information to a level where it is practical to use for this application. In generating time–stamps for program constructs we have the problem of needing to generate time–stamps for unbounded control structures. After a brief description of our prototype system, we describe the time–stamp requirements in section 2, and present a solution involving variable–length time–stamps in section 3. The rest of the paper goes on to show the optimization and implementation of the basic variable–length scheme. We finish with an example of the use of time–stamps in parallelizing a simple control structure.

1.1 Prototype system

In this section we discuss the approaches taken in our implementation of a prototype system for the optimistic parallel execution of sequential C++ code. There are two approaches to the optimistic execution of the sequential program: using pure time–warp and using time–warp combined with static dependency analysis of the program. We can achieve correct results with a pure time–warp system though it may optimistically execute statically dependent code. We can combine static program dependency analysis with optimistic execution by making use of static information to avoid optimistically executing anything which has unconditional static dependencies.

The prototype system makes use of source–to–source transformations using the sage++ transformation tool [3]. The transformations are directed by annotations added to the C++ program. This allows us to investigate heuristics for adding the annotations automatically. The partitioning of object–oriented programs onto processors takes place on a per object basis. The annotated objects are transformed into server objects which can then be placed on different processors. We choose the higher level objects as candidates for server objects to ensure a suitable granularity of tasks. A large enough level of granularity is necessary in order to achieve good efficiency. Currently object placement is decided manually via source–code annotations, with a view to investigating heuristics for automatic object placement.

1.2 Server objects

A server object is a transformed C++ object which is placed on a node of the parallel computer. Its method invocations are transformed into messages. Server objects are the level at which roll–back is managed in the system. Each server object maintains its own local virtual clock, and roll–back information. Server objects have an independent thread of control which performs optimistic execution of method invocation messages.

Parallelism is introduced into the system via the asynchronous dispatch of method invocation messages. Methods with return values are transformed to send back their results asynchronously. 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`. 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 processes, distributed over the processors, may send a request to a server object, each with a different time–stamp. A diagram of the server objects in the example and the asynchronous RPC messages passed between them is shown in figure 1.

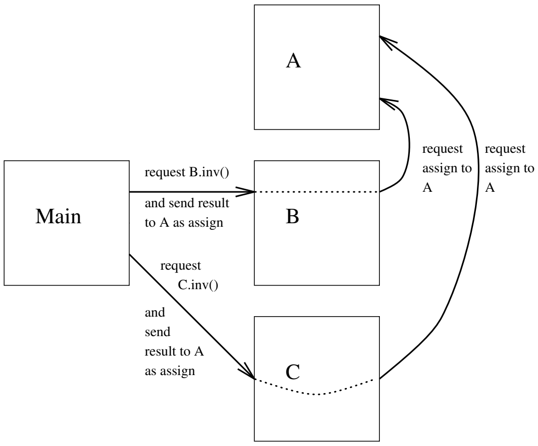

Figure 1: Server objects for Matrix class

In the transformed code, it is not possible for the server object `A` to 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. Instead what is done is that matrix `A`, and server objects in general act as servers only: all messages are handled as RPC requests.

The prototype system does not currently take advantage of existing parallelization techniques based on static analysis. A more complete system would include such techniques and only make use of optimistic execution when insufficient parallelism is obtained from static analysis.

2 Time–stamp requirements

The concept of an artificial time–scale was originally proposed by Lamport [14] in 1978 as a way of defining a global ordering of events in a parallel or distributed system. A causal relation between two events A and B, such as the sending and receiving of a message, can be represented as $A \rightarrow B$ and we say that A *happened before* B. Events A and B are said to be *concurrent* if neither can causally affect the other, that is $A \not\rightarrow B \; and \; B \not\rightarrow A$. Thus the ordering of events defined by the *happened before* relation is a partial ordering.

Lamport showed how to extend this partial ordering to a total ordering by assigning a logical clock value or *time–stamp* to each event in such a way that if $A \rightarrow B$ then the time–stamps will be ordered $T_A < T_B$. The converse is not true, $T_A < T_B$ does not imply that event A causes event B: all we can say is that event A *may* cause event B. However, if the time–stamps for two events are equivalent, $T_A = T_B$, then we know that A and B must be concurrent, that is there is no causal relation between these two events.

Logical clocks have been used in a number of theoretical and practical areas, such as the monitoring and debugging of parallel systems [5, 8, 20]. For such applications, it is usually possible to have a simple time–stamp allocation scheme, where time–stamps can be allocated from a sequence of integers in the order required. Although some extensions of this scheme have been proposed to reflect the partially ordered nature of logical time [9], it is still generally the case that each process maintains an integer valued counter for its own local time.

In many ways, the *virtual time* paradigm [12] is the inverse of Lamport's algorithm: 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 concurrent execution of those events. This is achieved using an optimistic execution mechanism such as Time Warp: each process executes without regard to whether there are synchronization conflicts with other processes. When a causality error is detected, a process must be rolled back in virtual time, and then be allowed to continue along new execution paths.

Although virtual time has been used in distributed database concurrency control [16], its main success has been in parallel discrete event simulation [10, 13, 17, 21]. Here, the interactions between the objects of the simulation are modelled by the exchange of time–stamped event messages. The Time Warp mechanism allows different nodes of the parallel computer to execute events out of time–stamp order provided that no causal relation exists between them. Although events of the simulation are executed in parallel, the mechanism guarantees that the same results are obtained as would be the case with a simulator that executed the events sequentially in non–decreasing time–stamp order.

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.

3 Representation of time–stamps

For our representation of time–stamps we require the operations to compare and generate the next time–stamp in a sequence to be efficient.

A variable length time–stamp is proposed in this paper which satisfies these requirements. The use of multi–level time–stamps to sub–divide the event time–line [15] for use in the integration of events into active object–oriented databases suggested the use of variable length time–stamps.

We define our variable length time–stamps:

a time–stamp is the pair $(length, value)$

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. 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, in figure 2 as the nodes of a binary tree.

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:

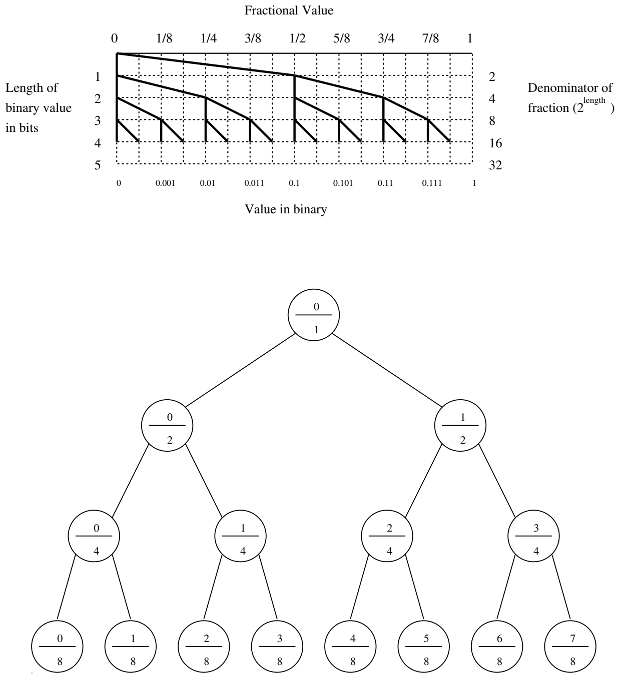

Figure 2: Variable length time–stamps

$$(l, v) < \left\{(l + d, 2^d v) < \ldots < (l + d, 2^d v + 2^d - 1)\right\} < (l, v + 1)$$

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. What we are proposing is that trailing zeros become significant in the ordering, so that $0.10 \neq 0.1$.

So for example:

fractional values		*arithmetic*	*time–stamp*
0.0	0.1	$<$	$<$
0.0	0.00	$=$	$<$
0.10	0.1	$=$	$>$

4 Mapping program control structures

In this section we will consider the use of the variable length time–stamps defined above in allocating time–stamps for typical program control structures. As an example of the type of program structure for which we will be generating time–stamps, consider:

```
loop A(3)
  loop B(<unbounded>)
    statement C
  end
end
```

The tree representing the sequential execution ordering of this code block, is shown in figure 3:

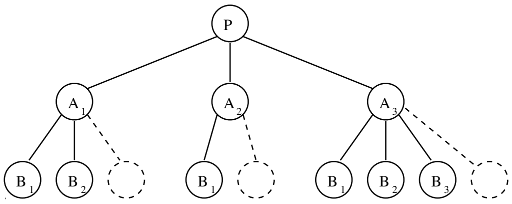

Figure 3: Code block

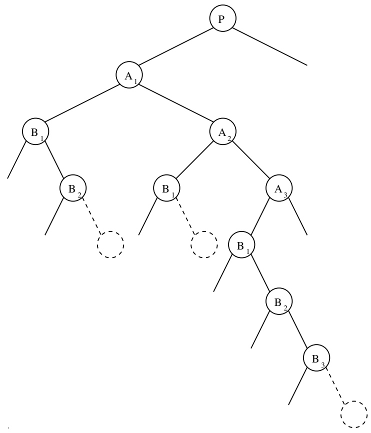

Figure 4: Code block, as binary tree

This representation allows the tree to be arbitrarily expanded for the unbounded loop. Arbitrary N-ary trees can be represented by binary trees without loss of information. In figure 4 we show the same code block as in figure 3, represented as a binary tree rather than an N-ary tree.

We can see that the binary tree representation shown in figure 4 makes it possible for us to assign time–stamps from the time–stamp tree in figure 2. All we need to do is assign the corresponding time–stamp for each tree node.

Using this scheme it is possible to assign time–stamps for arbitrary programs containing any number of unbounded constructs. However, the number of bits required to represent the time–stamp varies as $O(n)$ where n is the number of iterations of the loop. So for example, a loop A(128) requires a sequence value with 128 bits. This is a costly solution, in terms of bits per time–stamp.

5 Optimization of time–stamps

For loops with fixed bounds, and for sequences, we know that we require n time–stamps, for some fixed n. We can use this information to use less bits per time–stamp. To generate n time–stamps for parent time–stamp (l, v) we create a complete tree rooted at (l, v) with depth $\lceil \log_2 n \rceil$. We then allocate the n time–stamps as the leaf nodes in this tree. We are able to reduce the extra bits required for a loop or sequence of size n from n bits to $\lceil \log_2 n \rceil$ bits.

5.1 Unbounded loops

For unbounded loops we cannot predict the number of time–stamps to allocate. We allocate an initial amount of time–stamps N where $N = 2^b$ and b is the number of bits which will be used by the initial time–stamp allocation. We use the first half of these time–stamps for the first 2^{b-1} time–stamps. If more time–stamps are required we create complete binary trees of depth b under all of the remaining 2^{b-1} time–stamps. This will create 2^{2b-1} additional time–stamps. We again use the first half of these (2^{2b-2}) and reserve the second half for further time–stamps.

The choice of b is important in this scheme as the efficiency of dynamic allocation compared to the static allocation used for fixed bound loops depends on the choice of b.

We define the cost $\mathcal{C}$ of an allocation scheme as the number of bits used to represent time–stamps in that scheme. We also define the efficiency $\mathcal{E}$ of the allocation as the actual number of time–stamps used divided by the total number of time–stamps available for that cost, that is:

$$\mathcal{E} = \frac{n}{2^{\mathcal{C}}}$$

The cost in bits $\mathcal{C}_{\text{static}}$ and efficiency of static allocation $\mathcal{E}_{\text{static}}$ is given by:

$$\mathcal{C}_{\text{static}} = \lceil \log_2 n \rceil$$

$$\mathcal{E}_{\text{static}} = \frac{n}{2^{\mathcal{C}_{\text{static}}}}$$

The cost in bits $\mathcal{C}_{\text{dynamic}}$ and efficiency of dynamic allocation $\mathcal{E}_{\text{dynamic}}$ is given by:

$$\mathcal{C}_{\text{dynamic}} = Lb$$

$$\mathcal{E}_{\text{dynamic}} = \frac{n}{2^{\mathcal{C}_{\text{dynamic}}}}$$

where L, the number of levels of dynamic allocation required is:

$$L = \left\lceil \frac{\mathcal{C}_{\text{static}}}{b-1} \right\rceil$$

We can sometimes make use of the remaining time–stamps for allocation to the program control structures after the loop. In the static case we can use all of the remaining time–stamps, and so there is no wastage. In the dynamic case it is not generally possible to make use of the remaining time–stamps even at the current level, and we will also have lost some time–stamps at previous levels as compared to the static case.

A balance must be struck in the choice of b, too small and the efficiency will be poor for large unbounded loops as many levels will be required and time–stamps will be lost at each level. However, too large a value of b and there will be large numbers of unused time–stamps generated for unbounded loops which happen to have small bounds.

5.2 Recursion

The depth of time–stamp tree which recursion requires is $2r$ in the case of a simple recursive control structure, where r is the number of levels of recursion. Consider the simple recursive construct:

```
recur(A_i)
  statements(B)
  recur(A_{i+1})
  statements(C)
end
```

The execution tree is shown in figure 5, and it can be seen that in the unoptimized allocation scheme, two levels of the binary time–stamp tree are required to generate time–stamps for each level of recursion.

We can use a similar optimization for recursion as we presented for unbounded loops. We use the same scheme of initially allocating a number of time–stamps $N = 2^b$ as described above. We make use of the first half of this initial allocation by expanding the recursion. If more time–stamps are required, we create complete binary trees of depth b under the remaining half of the initial allocation, and continue in this way until we have the required number of time–stamps.

So for example, if we expand the above recursive construct once we would have:

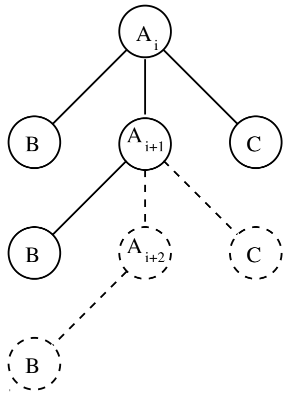

Figure 5: Recursive code block

```
recur(A_i)
  statements(B_1)
  expanded-recur(A_{i+1})
    statements(B_2)
    recur(A_{i+2})
    statements(C_2)
  end
  statements(C_1)
end
```

For this simple recursive construct we can see that we increase the number of time–stamps required for the block for each expansion we perform by 3. For more complex mutually recursive structures the increase in time–stamps per expansion would be greater. This optimization allows us to improve the efficiency of our use of time–stamp bits to $\mathcal{E}_{\text{dynamic}}$.

6 Implementation

On a machine with a word size of w we can represent time–stamp (*length*, *value*) as an array of integers, where the first element in the array is the length of the time–stamp in bits.

Time–stamp $t = \{\textit{length}, a_1, a_2, \ldots, a_n\}$ where:

$$n = \left\lceil \frac{\textit{length}}{w} \right\rceil$$

as n words are required to store a value of length *length*.

We can define an efficient function to compare time–stamps which computes:

$$\{l_a, a_1, a_2, \ldots, a_{n_a}\} = \{l_b, b_1, b_2, \ldots, b_{n_b}\}$$

$$\text{iff } l_a = l_b \text{ and } a_i = b_i \;\; \forall\, i,\ 1 < i < n_a$$

$$\{l_a, a_1, a_2, \ldots, a_{n_a}\} < \{l_b, b_1, b_2, \ldots, b_{n_b}\}$$

$$\text{iff} \begin{cases} l_a = 0 \text{ and } l_b > 0 \\ \text{or } a_1 < b_1 \\ \text{or } a_1 = b_1 \text{ and, recursively} \\ \{l_a - w, a_2, \ldots, a_{n_a}\} < \{l_b - w, b_2, \ldots, b_{n_b}\} \end{cases}$$

For many programs, it will only be necessary to use time–stamps with $length \leq w$. To give an idea of the complexity of program which can be represented with these time–stamps before an extended length time–stamp is required, consider:

```
loop <fixed>
  recur(A)
  end
end
```

With a choice of $b = 6$ for dynamic loop allocation, and assuming a word size $w = 32$, the recursion could reach depth 1,000 in a fixed loop of 1,000,000 before an extended time–stamp would be required.

With double length time–stamps more complex structures are possible:

```
loop <fixed>
  recur(A)
    loop <unbounded>
    end
  end
end
```

For a fixed loop of 1,000,000 and recursion of 30,000 the unbounded loop could reach iteration 1,000,000 before it would be required to extend it further.

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 rollbackable control flow statements. Parallelism is introduced into the system by the asynchronous dispatch of server object invocation messages. As described in section 1.2, 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 );
}
```

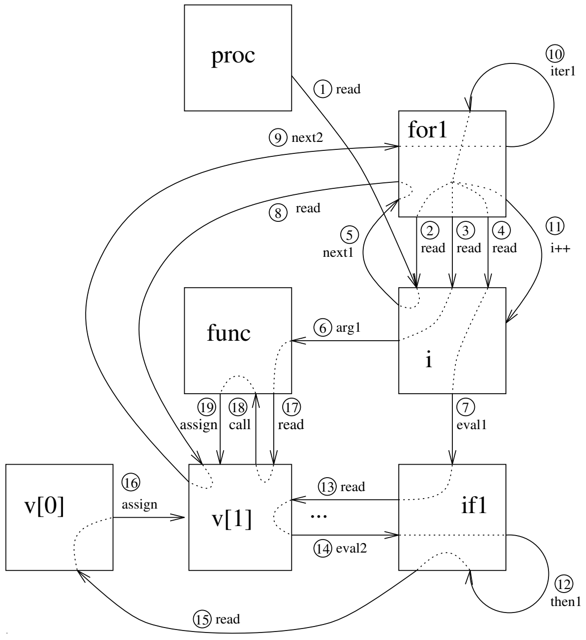

Figure 6: Asynchronous message passing

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.

8 Conclusions

The problems of time–stamp generation for unbounded control structures are addressed in this paper. An allocation scheme which uses variable–length time–stamps has been presented in which it is possible to allocate an arbitrary number of time–stamps between any pair of existing time–stamps. Optimizations have been presented, together with an analysis of the efficiency of such optimizations. This shows that for many programs, a single word is sufficient for the representation of the time–stamp values. Even when an extended time–stamp is required, the operations of generating and comparing time–stamps can be implemented efficiently.

One measure of the importance of parallel and distributed simulation has been its influence on other branches of computer science, such as concurrency theory [6]. This paper has shown how it is possible to extend the virtual time paradigm of optimistic simulation to the execution of general purpose programs. By assigning time–stamps to program control structures, 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. We are currently implementing a parallelizing compiler which converts conventional code into a form which has calls to a parallel run–time system which implements the optimistic mechanism, using the lightweight p4 [1, 4, 19] cluster model of parallel computing.

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. Our implementation of the prototype system should enable us to investigate the likely efficiency of the optimistic execution of sequential programs and to explore heuristics for reducing overhead. We have demonstrated the need for a flexible time–stamp generation scheme and have presented a solution which supports the optimistic execution of general purpose programs.

References

[1] A Back and S J Turner. Portability and parallelism with lightweight p4. In *BCS PPSG Conference on General Purpose Parallel Computing*, 1993.

[2] D Bacon. Optimistic parallelization of communicating sequential processes. In *Proceedings of the 3rd ACM Symposium on the Principles and Practice of Parallel Programming*, pages 155–166, 1991.

[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 (OON-SKI)*, 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] W Cai and S J Turner. An approach to the run-time monitoring of parallel programs. *Computer Journal*, 37(4):333–345, 1994.

[6] K M Chandy. Keynote address. *Parallel and Distributed Simulation Conference*, 1993.

[7] 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.

[8] C J Fidge. Reproducible tests in CSP. *The Australian Computer Journal*, 19(2):92–98, 1987.

[9] C J Fidge. Logical time in distributed systems. *Computer*, pages 28–33, 1991.

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

[11] C Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

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

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

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

[15] B Lings and L Foster. A temporal model for active object-oriented databases. Technical report, Exeter University, R284, 1993.

[16] M Livesey. Distributed varimistic concurrency control in a persistent object store. Technical report, University of St. Andrews, 1990.

[17] M Presley, M Ebling, F Wieland, and D Jefferson. Benchmarking the time warp operating system with a computer network simulation. In *Procedings SCS Distributed Simulation Conference*, pages 8–13, 1989.

[18] S J Turner and A Back. General purpose optimistic parallel computing. In *Proceedings of 7th PARSYS User Group Meeting*, 1994.

[19] S J Turner and A Back. A T9000 implementation of the p4 parallel programming model. In *Proceedings of the North American Transputer User Group*, pages 305–316. IOS Press, 1994.

[20] S J Turner and W Cai. The 'logical clocks' approach to the visualization of parallel programs. In *Performance Measurement and Visualization of Parallel Systems (eds G Kotsis and G Haring)*, pages 45–66. Elsevier, 1993.

[21] 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.