

Timestamp Representations for Virtual Sequences

John G. Cleary, J. A. David McWha, Murray Pearson

Dept of Computer Science, University of Waikato, Private Bag 3105, Hamilton, New Zealand.

{jcleary, jadm, mpearson}@cs.waikato.ac.nz

Keywords: optimism, Time Warp, timestamps, parallel execution

Abstract

The problem of executing sequential programs optimistically using the Time Warp algorithm is considered. It is shown how to do this, by first mapping the sequential execution to a control tree and then assigning timestamps to each node in the tree.

For such timestamps to be effective they must be finite, this implies that they must be periodically rescaled to allow old timestamps to be reused. A number of timestamp representations are described and compared on the basis of: their complexity; the frequency and cost of rescaling; and the cost of performing basic operations, including comparison and creation of new timestamps.

1. Introduction

Optimistic execution algorithms using causality error detection and recovery have been used successfully for speeding up parallel discrete event simulation [6, 7]. *Time Warp* [9], the best known of these algorithms, allows events to be scheduled for execution without regard for causality conflicts. When a conflict does occur the simulation recovers using *rollback*, which returns the simulation to an earlier correct state.

The *virtual time* paradigm [9] forms the basis for the Time Warp algorithm. Virtual time is a totally ordered time scale, which restricts the causality between events. Each event is assigned a virtual time value and can only causally affect events with a later virtual time. In simulations the virtual times are typically integer or real values explicitly computed in the model code.

Although most widely used for parallel discrete event simulation, Time Warp can also be used as a synchronisation mechanism for more general parallel computation tasks. For example it can be used as an undo mechanism in editors [12], consistent checkpointing for fault tolerance, incremental recovery in databases, selective undo in groupware tools [11], or debugging for parallel and distributed programs [4]. The application we will focus on, though, is general purpose sequential computation. We associate timestamps with each part of the sequential computation. The ordering of the timestamps is the same as the original sequence and the timestamps can be generated freely so long as they conform to this requirement. To distinguish this from simulations, where timestamps are computed explicitly, it will be called a *virtual sequence* (rather than a virtual time).

Given some efficient way of generating such a virtual sequence, any of the conservative or optimistic algorithms for parallel simulation can be used to parallelize the sequential execution. Two different systems using Time Warp to implement these ideas have appeared in the literature. Back and Turner [1, 2, 13] have constructed a software based system for C++ and Cleary et al [5] have designed a general purpose CPU which transparently executes sequential code.

As observed by Back and Turner [2] a major problem of using Time Warp with a virtual sequence is that it may be necessary to allocate an arbitrary number of new timestamps between any pair of existing timestamps. The solution in Back and Turner (infinite length timestamps similar to the length representation described below) works well where the size of the timestamp representation is not important. Unfortunately, in real systems there are always storage and bandwidth limits. Many events get spawned during execution and when a large amount of parallelism is being extracted many events are active at one time and many messages are being passed around the system. If the sequential computation is in an imperative language then it is necessary to record the timestamp of each write to and read from memory. If the timestamp representation is allowed to become arbitrarily large then storage and bandwidth limits will be quickly reached. Since timestamps are fundamental to an optimistic computation system and will be used in many places this is an issue of crucial importance in the design of the system.

In section 2 of this paper we review the execution structures used by the Time Warp algorithm and the mechanisms (fossil collection, cancelback and rescaling) used to guarantee execution progress. Section 3 describes two basic fixed length timestamp representations. In sections 4 and 5 we build more refined representations on this basic form. Section 6 draws some conclusions about the best all round representation to use.

2. Optimistic Execution

In this section we review the basic mechanisms necessary to map a sequential execution onto a parallel optimistic execution.

2.1. Execution Trees

As a first step toward timestamping and parallelising sequential code, it is mapped onto an *execution tree*. The idea is that basic blocks of execution are mapped to each node of the execution tree. A depth first left-to-right traversal of the tree gives the correct sequential order of execution. By arranging the control in this way it is possible to remove (or ameliorate) control dependencies during parallel execution.

To exploit this it is essential that a compiler be able to generate code that maps well onto this tree structure. For example, it is important that most nodes generate more than one child, otherwise control dependencies force sequential execution. Figure 2.1 shows a tree that might be generated for the code sequence: A; B; C; D;

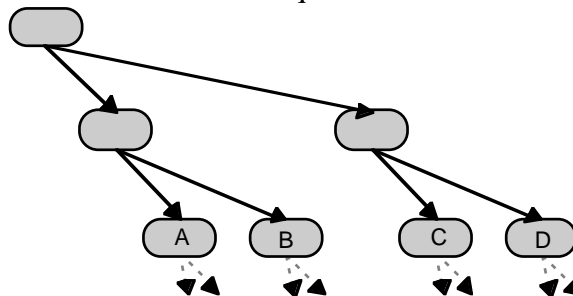


Figure 2.1 Tree generated for sequential code

In the case of a *for* loop, where the bounds are known statically, a balanced binary tree can be generated. Figure 2.2 shows a tree generated for:

```

for i = 1 to 8 do iter;
follow; /* statements following for loop */

```

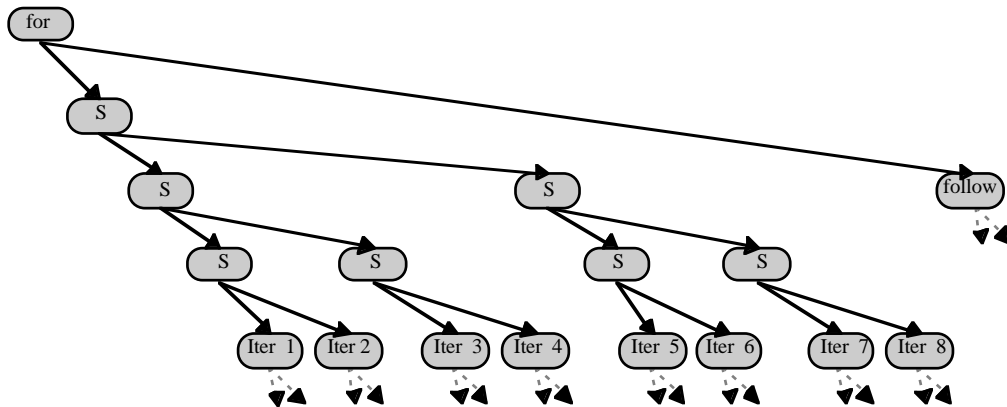


Figure 2.2 Execution tree for simple *for* loop

Notice that the statements immediately following the *for* loop are available for execution in parallel with the *for* loop.

While loops are more difficult, as the number of iterations to be executed may be unknown. However, it is possible to schedule iterations of the loop speculatively as shown in Figure 2.3.

```

while Cond do
    iter;
follow; /* statements following while loop */

```

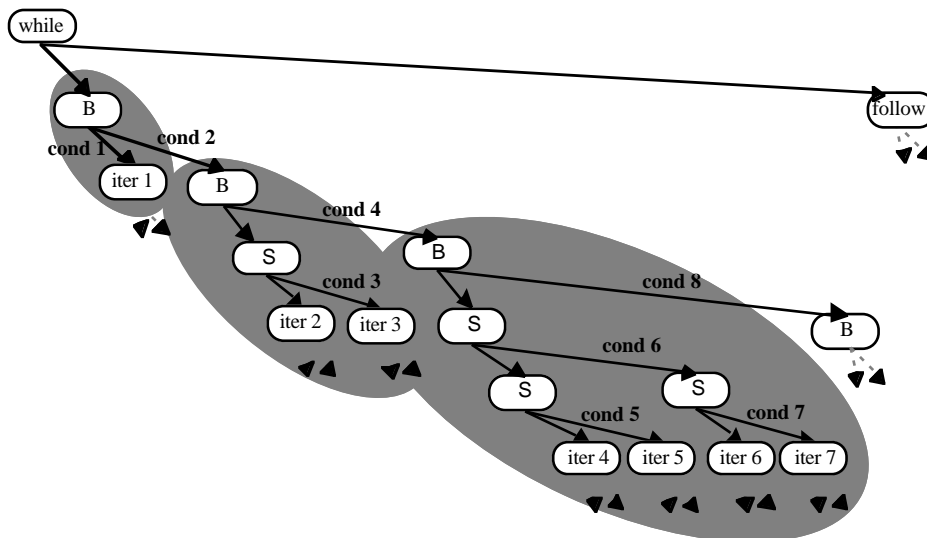


Figure 2.3 Execution tree for a *while* loop

In this example the arcs labelled *cond x* point to subtrees which are scheduled conditionally. To allow these subtrees to be executed speculatively each of the conditionals is optimistically set to true allowing the execution tree shown to grow quickly. As soon as a conditional evaluates to false (ie. the loop terminates), execution of the subtree below it is rolled back and terminated. Also notice in this structure there is a linear “backbone” (the nodes labelled B), which starts exponentially increasing numbers of iterations. This structure balances the overhead of executing loops which

contain only a small number of iterations (which is the most common case [8]) against the cost of executing loops with a large number of iterations in them.

When generating code for conditional statements (such as *if-then-else*) a number of possibilities exist. In the most conservative case the condition is evaluated first to determine which of the branches should be executed. A more aggressive approach is to execute the conditional and one of the branches speculatively. In the event that the correct branch was executed speculatively a speedup occurs. However, if the wrong branch was executed its execution has to be rolled back and the correct branch taken. A still more aggressive approach is to execute both possible branches, as well as the condition, in parallel and always roll back one of the branches.

2.2. Fossil Collection and Cancelback

The above figures represent a static picture of the execution trees. When Time Warp is applied to their execution, multiple nodes of the tree can be executed in parallel without regard for any data dependencies that may exist between them. In the case that a dependency does exist between two nodes and they are executed out of order the dependent node must be rolled back. This means that enough information must be maintained within the system to ensure that any data dependencies which are not met can be detected and corrected.

As nodes cannot produce results which affect nodes executing earlier in the virtual sequence there will always be an earliest active node in the system, which cannot be rolled back. The virtual time of this node is known as *Global Virtual Time* (GVT) and represents the current point in the program which can be considered to have safely completed execution. Any resources (including timestamps) that have been allocated to nodes with timestamps less than this time can be released back to system for reuse. The process of reclaiming these resources is known as *fossil collection*. In most cases this is enough to ensure execution can progress, because if a node is scheduled for execution and there are not enough resources available to do so, then the node can wait for GVT to progress, freeing up the necessary resources.

However, there are situations where fossil collection alone is not enough to ensure execution will always be able to progress. It is possible for a node scheduled for execution to become the GVT holder before the resources necessary to execute it become available, as they are being used by nodes later in the virtual sequence. *Cancelback* is an additional mechanism that ensures that execution can proceed. It allows a node and all its descendants to be cancelled freeing up resources within the system. The node is then rescheduled for execution later. The original description of the cancelback algorithm [10] showed that it guarantees that a Time Warp simulation can always complete in the same space as that required by a sequential implementation.

2.3. Rescaling

Each node in the execution tree can be associated with a string that gives the path from the root to the node. A zero is used for a left branch, one for a right branch and Δ to terminate the string (Figure 2.4). A lexicographic ordering where $\Delta < 0 < 1$, places these strings in the same order as the sequential execution order of the associated nodes. Thus the strings can be used as timestamps for the virtual sequence.

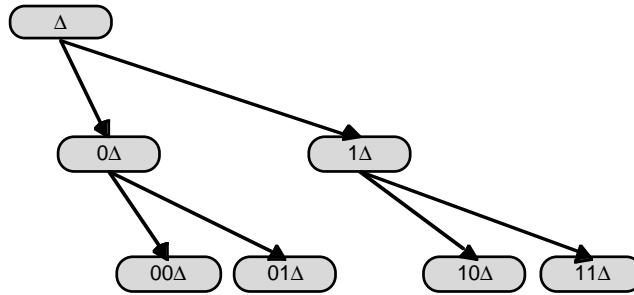
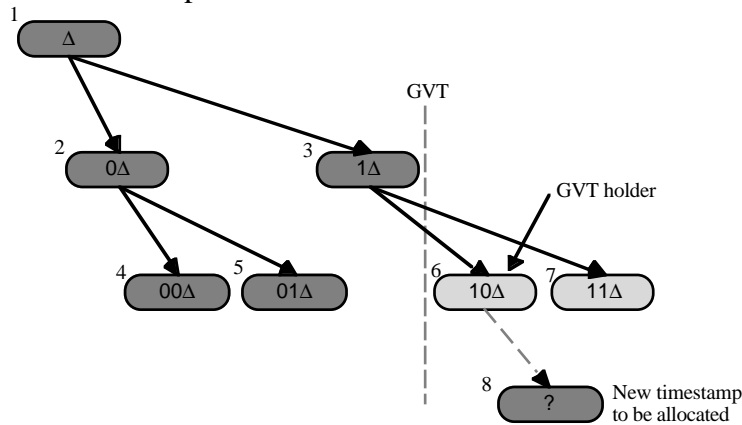
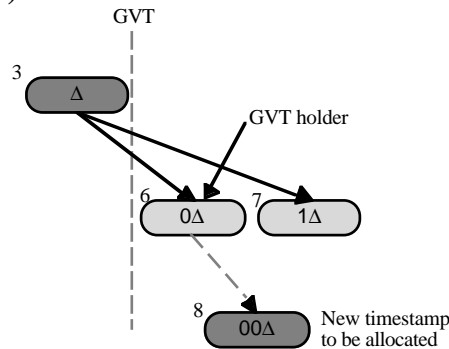


Figure 2.4 Example execution tree showing timestamps

The main problem addressed by this paper is that of finding an efficient representation of these strings. The first problem that arises is that the strings can be of unbounded length. For efficiency it is essential that each timestamp be represented in a single machine word (32 or 64 bits). The important operations on timestamps should also be fast. These operations include the ordered comparison of two timestamps and the generation of new timestamps for child nodes.



(a) Execution tree before rescaling



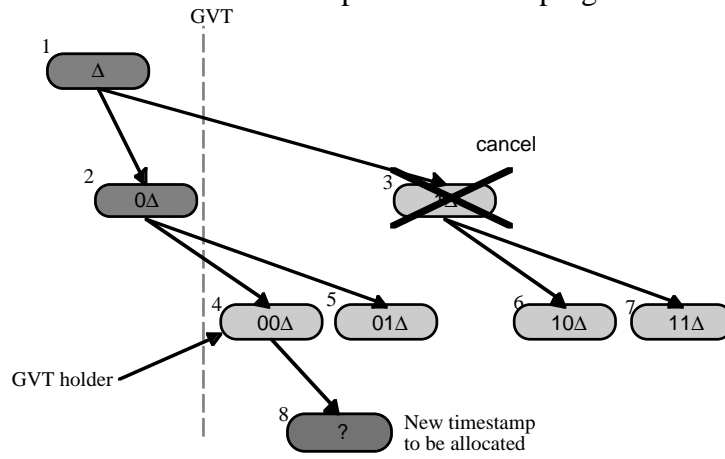
(b) Execution tree after rescaling

Figure 2.5 Rescaling to the root of the execution tree

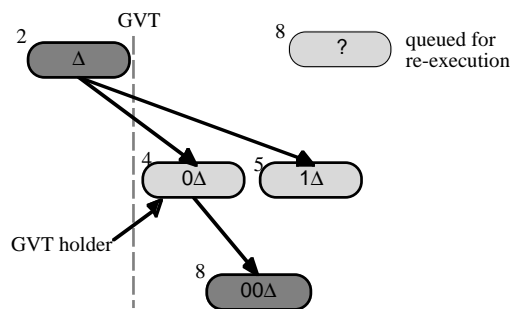
A problem that arises for any finite representation is the need to re-use old timestamps. As the execution tree grows, eventually there will be more nodes than can be represented. As GVT advances, early timestamps will become available for re-use. However, to make use of these requires that the current timestamps be re-allocated while retaining their ordering. For example, consider the execution tree shown in Figure 2.5 which uses a timestamp representation with at most three symbols (including the terminating Δ). A new node 8 is to be added to the tree. In this example all the nodes shaded with cross hatching have been fossil collected, so their timestamps can be reclaimed for reuse. This can be achieved by selecting the node which is the GVT holder and traversing up the tree until a node is found which has all currently active

nodes in its subtree and making this the new root of the tree. All the nodes below it can then be rescaled.

In some cases the rescaling operation cannot reclaim space in the tree even though space may exist. For example in Figure 2.6 it is not possible to rescale because the root node has a right-most child that is not an ancestor of the current GVT holder. A solution is to cancel node 3, and delete its parent, node 1. This gives enough room for a rescaling which provides room to create node 8. For all of the finite representations investigated in this paper, cancellation is sufficient to permit forward progress.



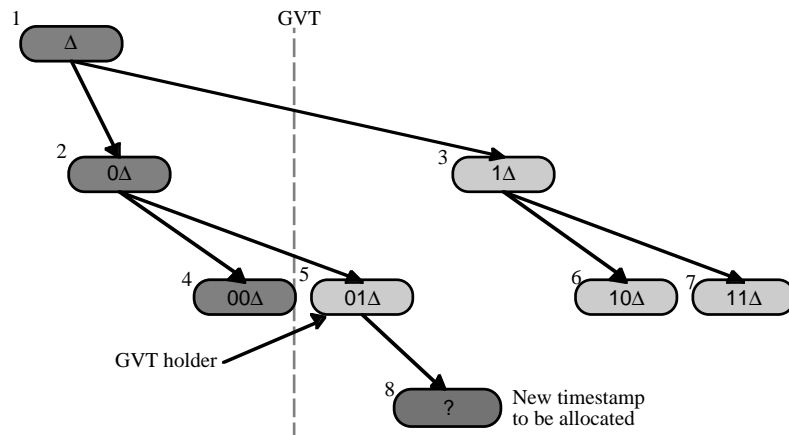
(a) Execution tree requiring cancel operation



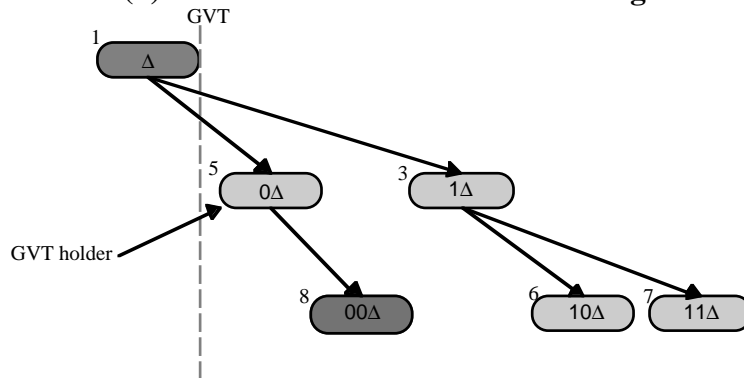
(b) Tree after cancel operation and rescaling

Figure 2.6 Cancelback of an exhausted execution tree

Cancellation, however, has a high cost, as work has been done and then thrown away to no good effect. It is better if possible to avoid it. Figure 2.7 shows a less extreme version of the previous example. In this case it is possible to rescale just the left half of the tree (without cancelling any nodes). Such a rescaling is achieved by moving the node that is the GVT holder as far to the left as possible and then up the tree until a node which has a right most child is detected. All the descendants of the GVT holder can then be rescaled making space for the new node to be added.



(a) Execution tree before rescaling



(b) Partially rescaled execution tree

Figure 2.7 Rescaling part of the execution tree

With the timestamp representation described so far, there is a maximum depth to which the execution tree can grow. This will lead to situations where the timestamps will have to be frequently rescaled. However, rescaling is expensive as every timestamp in the system has to be altered. Also, while rescaling is being performed it may be necessary to suspend execution of the program to avoid inconsistencies in the timestamps, further raising the cost of rescaling. To minimise the performance losses due to rescaling two main types of solutions exist. The first is to use a representation that requires rescaling less frequently. The second solution is to reduce the cost of rescaling. The following sections describe three different timestamp representations which can be used to address this problem.

3. Timestamp Representations

3.1. Integer Representation

To make timestamp comparison easy we can map the timestamps from bit strings to integer values. This makes it possible to compare timestamps bitwise, as for integers. This is only possible because the timestamps are finite length and, hence, we know the maximum number of timestamps that can be inserted between any two arbitrary timestamps in the virtual sequence.

Each timestamp also needs its depth in the tree to be recorded. This is needed for the calculation of child timestamps and for rescaling, but not for comparison. Timestamps are allocated to children as shown in Figure 3.1, where n is the maximum number of levels in the tree and d is the depth of the new child. If the parent has the (integer)

timestamp P , then the left child is allocated the timestamp $P + 1$ and the right child the timestamp $P + 2^{n-d}$. The resulting coding for 3 bit integers is shown in Figure 3.2.

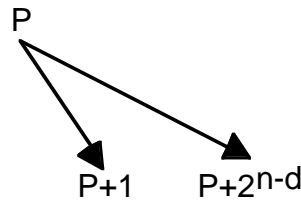


Figure 3.1 Allocation of timestamps to children

Rescaling is a costly operation for this representation because there is no simple arithmetic operation that can be applied to rescale all timestamps. Rescaling and partial rescaling require, essentially, that the original bit string be recovered, the leading bits deleted and then that it be recoded. This takes $O(n)$ operations.

3.2. Length Representation

An alternative way to convert the bit string timestamp to an integer is to pad the bitstring out to n bits with zeros and then append the length of the string (less one) to form a 2's complement integer. Figure 3.2 compares this representation using two bits to represent the string and two bits for the length with the integer one above. It is a slightly less efficient than the integer representation as it requires an additional $\lceil \log_2 n \rceil - 1$ bits.

String	3-bit integer representation		length representation, 2+2 bits		
	I		I	L	
Δ	000	(0)	00	00	(0)
0Δ	001	(1)	00	01	(1)
00Δ	010	(2)	00	10	(2)
01Δ	011	(3)	01	10	(6)
1Δ	100	(4)	10	01	(9)
10Δ	101	(5)	10	10	(10)
11Δ	110	(6)	11	10	(14)

Figure 3.2 Comparison of integer and length representations

Comparison can be done with a bitwise comparison on the complete bit string. Given the parent's timestamp is the integer P , the left child's is $P + 1$ and the right child's is $P + 2^{n-d+1} + 1$ where n is the number of bits used for the string and $l = \lceil \log_2 n \rceil - 1$ - the number of bits needed for the length. Rescaling operates on the two parts of the integer separately. The string part is left shifted and the length is decremented.

4. Rescaling Less Often

By allowing the maximum depth of execution trees to be greater than in the basic representation the frequency of rescaling operations can be reduced. One representation

that can achieve this uses a mantissa and exponent and varies the range of timestamps allocated to children.

4.1. Exponential Representation

This representation uses a scheme similar to floating point number representations to allow different parts of the tree to grow to different depths. It is comprised of two parts: a *mantissa* and an *exponent*. The exponent is the number of leading zeros in the timestamp, while the mantissa is the normalised tail of the timestamp. In the example in Figure 4.1 the timestamp 000101 Δ becomes 3,101 Δ , where the number before the comma is the exponent and the string after the comma is the mantissa. Using this representation, the left-most path from the root of the tree can grow to (2^n+m) levels for a representation that uses an n bit exponent and m bit mantissa. Moving to the right, the depth of the paths decreases until reaching the right-most path where the maximum depth is m levels.

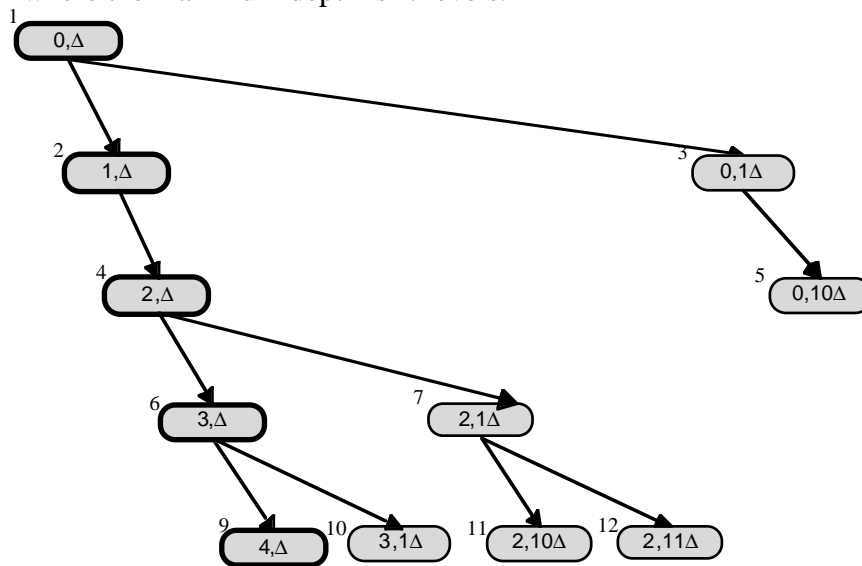


Figure 4.1 Timestamps in exponential form

This representation favours execution of the left side of the tree. That is, nodes that are early in the virtual sequence can have longer strings and so will not exhaust the maximum depth so soon. Thus rescale and possibly cancel operations can be avoided. This has two additional advantages. First, by delaying execution of nodes to the right of the tree which are more speculative it helps balance the overall execution. Second, a compiler can take advantage of the representation by pushing more computation down to the left of the tree.

A complete exponential representation requires that the mantissa be coded using either the integer or length representation above. Unfortunately the details of this are complex as allowance must be made for mantissas that contain only Δ after normalisation and for the “wasted” bit resulting from the fact that the leading bit for all other mantissa’s will be 1. Whichever representation is used, comparison of two timestamps is complex requiring decoding and comparing the exponent and mantissa separately - it does not seem possible to construct a simple bitwise ordered representation.

The rescaling and creation operations are equally complex. So while this representation has promise in that it may be able to avoid rescaling and cancellation operations the details make it unsatisfactory. The next representation seeks to retain the ability to avoid rescaling without the attendant complexity.

4.2. Arithmetic Coding

Rescaling is necessary no matter what finite representation is used. However, consider a perfectly balanced execution tree with leaves where the only leaf node with a child is the right-most one. In such a tree rescaling would only be necessary once every 2^n nodes, thus amortising its cost over much computation. It is difficult however for either the compiler or the run-time system to achieve such a perfect balance. This section describes a time stamp representation, based on arithmetic coding [3], that allows a dynamic allocation of timestamps and the possibility of approximating the perfectly balanced tree.

Each node is allocated a range of (finite integer) timestamps, for itself and its descendants. The interval is described by a lower bound, which becomes the timestamp of the node itself, and an upper bound. A descendant can then be allocated a timestamp sub-interval anywhere within its parent's interval. By allocating the sub-intervals in proportion to the size of the respective sub-tree then a balanced allocation of timestamps is possible.

Figure 4.2 shows a simple example of arithmetic coding using four bit timestamps. The children are allocated timestamp intervals in the proportions shown on the arrows. For example, the interval at the root node is split with one part to the right and three parts to the left. The limits of the tree are reached when the interval contains just one timestamp, as in, the right most leaf. Such trees may have leaves at widely varying depths according to the size of the allocated sub-intervals.

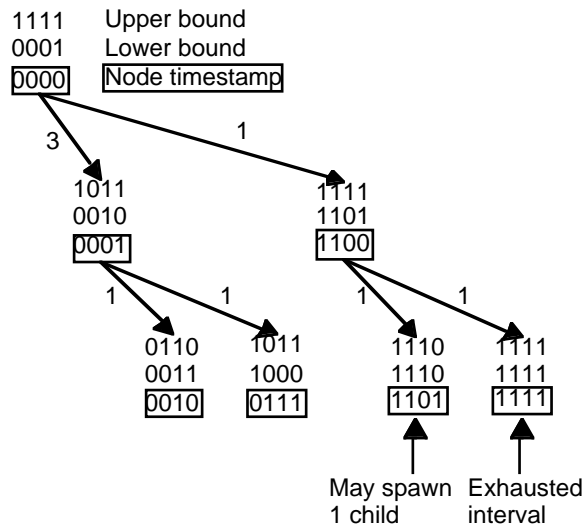


Figure 4.2 Arithmetic coding using four bit timestamps

The proportions of sub-intervals can be calculated statically by the compiler or dynamically at run-time. Dynamic allocation is indicated for some situations, for example, for loops where the bounds are not known till run time. Imperfect estimation of the sizes of sub-trees causes problems in the parts of the tree that are under-allocated and extra rescaling and cancellation may be necessary for computation to continue. This representation will do no worse than the static unbalanced techniques described earlier (allocating equal weights to each child gives almost the same codes as the integer representation). The better the estimates of sub-tree size, the less often rescaling is necessary.

The individual timestamps associated with each node can be represented as simple integers, with timestamp comparison using normal bitwise compares. Calculation of the child intervals needs only integer multiplication (albeit with careful attention paid to rounding, see [3] for details). Rescaling requires that both the upper and lower bounds

at each node be left shifted. In the case of the upper bounds 1's are shifted into the low order positions and in the case of the lower bounds 0's are shifted in.

5. Reducing the Cost of Rescaling

5.1. Moving Head Representation

While the representations described in the previous section can reduce the frequency of rescaling operations, rescaling is still very expensive as every timestamp in the system must be altered individually.

A technique to reduce the cost of rescaling is based on the observation that when rescaling occurs in either the length or the arithmetic coding representations, the same bits are removed off the front of all the timestamps being rescaled. To take advantage of this a global pointer (the *head pointer*), which defines the start of all timestamps in the system, can be used. Rescaling requires only this single global variable to be altered. The bits in the timestamp are assumed to wrap around, so that no precision is lost when rescaling, as shown in Figure 5.1. A tail pointer is used to terminate each timestamp because, after the head has been moved, no assumptions can be made about the value of the unused bits.

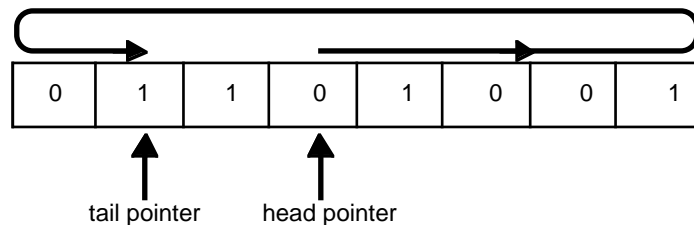
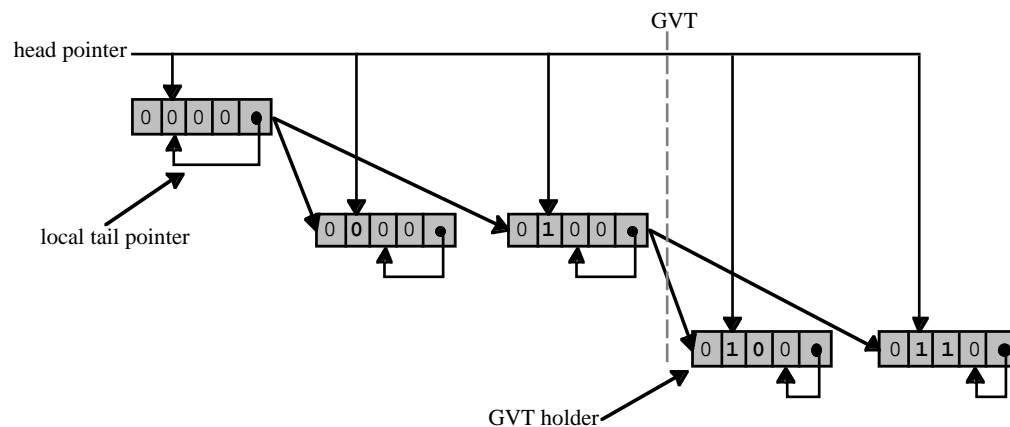
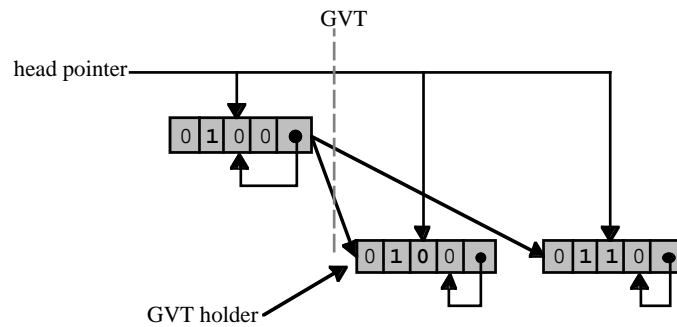


Figure 5.1 Moving head timestamp representation

The general rescaling operation described for the length representation is the same for this representation except that it is only necessary to alter the head pointer as shown in the example in Figure 5.2. Note that it is not necessary to perform any operation on the timestamps themselves and hence the cost of rescaling is greatly reduced.



(a) Execution tree before rescaling



(b) Execution tree after rescaling

Figure 5.2 Rescaling the moving head representation

As moving the head pointer alters all timestamps in the system it not possible to perform the partial rescaling operation using this method.

The main advantage of this representation is that the cost of rescaling has been significantly reduced as only the head pointer has to be altered. However, as there is no partial rescaling operation the number of cancel operations will be greater.

6. Conclusions

A number of different mechanisms have been examined for representing timestamps to be used with TimeWarp based execution of sequential computations. An efficient representation for these timestamps is important because they are repeatedly compared and manipulated within the TimeWarp algorithm. The technique of storing a range of integer timestamps with each node and using arithmetic coding to allocate a new range to children seems to give a well balanced representation. Each of the comparison, creation and rescaling operations is reasonably simple for it both in software and hardware implementations. The most complex operation is the arithmetic coding itself which requires an integer multiplication.

One major problem with any finite timestamp representation is the need to periodically rescale all or part of the timestamps in the system and that sometimes this requires cancellation of parts of the execution tree. The arithmetic coding representation can ameliorate this by allowing the execution tree to be balanced and thereby make more efficient use of the timestamps. Arithmetic coding can also be combined with the use of a global head pointer which allows a “zero cost” rescaling operation, provided all the timestamps in the system are simultaneously rescaled. The tradeoff here is that it may be necessary to cancel some nodes in order to allow global rescaling.

The major remaining work is assessing how effectively arithmetic coding can be deployed in practice to rebalance the execution tree and make efficient use of the timestamps. To do this perfectly is impossible but the question remains of whether, in practice, it can sufficiently reduce the number of cancel and rescale operations. We intend to simulate a number of sequential programs with different estimation algorithms to see how many rescale and cancel operations will be necessary.

7. References

- [1] Back, A. and Turner, S., (1995) “Using Optimistic Execution Techniques as a Parallelisation Tool for General Purpose Computing,” Proceedings of HPCN Europe, pp21-26, May.
- [2] Back, A. and Turner, S., (1995) “Time-Stamp Generation for Optimistic Parallel Computing,” Proceedings of 28th Annual Simulation Symposium, pp. 144-153, April.

- [3] Bell, T.C., Cleary, J.G., and Witten, I.H. (1990) "Text compression," Prentice Hall, Englewood Cliffs, NJ.
- [4] Choi, J.D., Miller, B.P. and Netzer, R.H.B., (1988) "Techniques for Debugging Parallel Programs with Flowback Analysis," Dept. of Computer Science, Univ. of Wisconsin Tech. Report No. 786, later in ACM Transactions on Programming Languages and Systems.
- [5] Cleary, J.G., Pearson, M. and Kinawi, H., (1995) "The Architecture of an Optimistic CPU: The WarpEngine," Proceedings of HICSS, vol 1 pp. 163-172.
- [6] Fujimoto, R.M., (1989) "Time Warp on a Shared Memory Multiprocessor," Transactions of the Society for Computer Simulation 6(3), pp. 211-239, July.
- [7] Fujimoto, R.M., (1990) "Parallel Discrete Event Simulation," Communications of the ACM, 33(10), pp. 30-53, October.
- [8] Hennessy, J.L. and Patterson, D.A. (1996) "Computer Architecture: A Quantitative Approach," second edition, Morgan Kaufmann Publishers, San Francisco.
- [9] Jefferson, D.R., (1985) "Virtual Time," ACM Transactions on Programming Languages and Systems, 7(3), pp. 404-425, July.
- [10] Jefferson, D.R., (1990) "Virtual Time II: Storage Management in Distributed Simulation," Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing, pp. 75-89, August.
- [11] Prakash, A. and Knister, M.J., (1992) "Undoing Actions in Collaborative Work," ACM Conference on Computer Supported Cooperative Work: Sharing Perspectives, Toronto, pp. 273-280, November.
- [12] Thimbleby, H., (1990) "User Interface Design," ACM Press, pp. 261-286.
- [13] Turner, S.J. and Back, A. (1994) "General Purpose Optimistic Parallel Computing," Proceedings of 7th PARSYS User Group Meeting, April.