Applying Time Warp to CPU Design

Murray W. Pearson, Richard H. Littin, J.A. David McWha and John G. Cleary

Department of Computer Science, University of Waikato,
Private Bag 3105, Hamilton, NEW ZEALAND
Email: {mpearson, rhl, jadm, jcleary}@cs.waikato.ac.nz

To appear High performance Computing Conference, 18-21 December, 1997, Bangalore, India

Copyright 1997 IEEE. Published in the Proceedings of HiPC'97, December 18-21, 1997 in Bangalore, India. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE. Contact: Manager, Copyrights and Permissions / IEEE Service Center / 445 Hoes Lane / P.O. Box 1331 / Piscataway, NJ 08855-1331, USA. Telephone: + Intl. 908-562-3966.

Applying Time Warp to CPU Design

Murray W. Pearson, Richard H. Littin, J.A. David McWha and John G. Cleary

Department of Computer Science, University of Waikato,
Private Bag 3105, Hamilton, NEW ZEALAND
Email: {mpearson, rhl, jadm, jcleary}@cs.waikato.ac.nz

Abstract

This paper exemplifies the similarities in Time Warp and computer architecture concepts and terminology, and the continued trend for convergence of ideas in these two areas. Time Warp can provide a means to describe the complex mechanisms being used to allow the instruction execution window to be enlarged. Furthermore it can extend the current mechanisms, which do not scale, in a scaleable manner.

The issues involved in implementing Time Warp in a CPU design are also examined, and illustrated with reference to the Wisconsin Multiscalar machine and the Waikato WarpEngine. Finally the potential performance gains of such a system are briefly discussed.

1. Introduction

Computer designers currently face a very interesting set of challenges. The steady increase in the number of transistors on a chip and the speed at which a chip can be clocked has continued its inexorable progress. In 1997, chips with millions of transistors and clock speeds of hundreds of MHz are in routine production and there are no short-term barriers to these numbers continuing to double every three years into the foreseeable future [4]. However, off-chip and memory speeds have not increased to nearly the same extent [7]. This has led to a situation where system performance is constrained by the latency of memory fetches and off-chip interactions. One response to this has been to achieve additional performance through the use of parallelism.

While this parallelism can be extracted at multiple levels the trend of modern processor designers has been to extract Instruction Level Parallelism (ILP). This trend started with the design of processors containing multiple pipelined functional units that were capable of scheduling more than one instruction every clock cycle. The performance of these superscalar processors is dependent on scheduling instructions so that the utilisation of the functional units in the processor is maximised. Consequently, a large amount of effort has been invested in techniques to maximise the number of instructions scheduled for execution in each clock cycle.

Early RISC processors relied on compiler technology to generate a sequence of instructions that exhibited good run-time behaviour on the particular architecture. However the number of instructions that can be concurrently scheduled is determined by data and control dependencies. When scheduling in software it is only possible to use static information. As the available resources in a system has increased new hardware based scheduling techniques which make use of dynamic information have been developed to allow more instructions to be scheduled every clock cycle.

Microprocessors now require a window of instructions, which can potentially be scheduled in a clock cycle, to keep the pipeline full and the execution units busy. Using hardware support for data and control dependence analysis, instructions that can be executed are marked. The schedular then selects combinations of these instructions to maximise the use of the functional units. Increasing the size of this window increases the probability that functional units can be fully utilised.

Without out-of-order execution the instruction window is limited to instructions between the currently executing instruction and the first instruction with a data dependence on the result of an incomplete instruction. Tomasulo [20] showed that by allowing the instructions to be executed out of order the execution window can be expanded. Elaborations of Tomasulo's reservation stations are used in many contemporary production CPUs [9], [10], [15], [17], [22].

To safely execute out of order, instructions with a data dependency on an earlier instruction must wait for the earlier instruction to complete. Data dependencies can occur via registers or via the memory subsystem. To determine whether there are dependencies via the memory subsystem later memory operations must wait at least until the addresses of earlier stores are known. The Intel Pentium Pro implements a subset of the possible re-orderings of memory operations [10], allowing loads to

pass stores (that is a later load can be executed before an earlier store) and loads to pass loads. Given the ILP available in the Pentium Pro this seems to give good performance while more aggressive schemes do not give much further improvement.

Out-of-order execution by itself limits the instruction window to a single basic block, which averages only five instructions [8]. To maintain an instruction window large enough to keep a number of functional units busy, instructions from multiple basic blocks must be included. One way to do this is to speculate on the outcome of branches by executing both paths through a branch and then discarding the untaken branch. Instructions other than branch instructions can also be executed speculatively. Speculative execution necessitates some method of recovery from data dependence conflicts, usually by returning to an earlier state and re-executing.

Time Warp is a mature speculative algorithm that was originally described as a way to parallelise discrete event simulations [11]. However, it is a general technique that can be applied to parallelise any sequential computation. This paper introduces Time Warp as a way to further widen the execution window for ILP.

The next section surveys recent advances that allow the execution window to be further opened. This is followed by an introduction to Time Warp and a discussion of its similarities to current architectural terminology and techniques. Discussion of the major issues associated with developing an architecture based on Time Warp follows.

2. Speculation

While speculation increases the instruction window size it places additional requirements on the CPU. It must be possible to eventually retire or commit instructions when their preconditions are finally met and they are no longer speculative. Alternatively, if the speculation was incorrect then it must be possible to discard or quash the incorrect instructions.

Speculation is usually considered in the context of speculation on branch instructions. This can be done a number of ways. For example a prediction can be made as to the direction a branch will take and the CPU follows this direction. It has been shown that these mechanisms can achieve up to 93% prediction accuracy [16]. However, after only a few branches the probability that the correct path is being followed is too low to be useful. A more aggressive approach is to follow both paths through a branch but again this can achieve only a small amount of parallelism because of the exponential explosion in the number of possible paths.

Other forms of branch prediction include: loop buffers, multiple instruction streams, prefetch branch target, data fetch target, prepare to branch, delayed branch, look ahead resolution, branch folding and branch target buffers. They vary in the way in which they predict a branch and whether or not they hold branch target information. Also they vary in the location in the instruction pipeline in which they take effect. These forms of dynamic branch prediction are summarised in [13] and [16].

Speculation can be performed on both the data and on the control structure of a program. It has been shown in [14] that data value prediction, by making a guess as to the value to be returned by a load, would give performance gains of 4.5% – 23% if it were incorporated into a PowerPC 620. This form of speculation is still theoretical and has not been included in any delivered architecture.

A different style of speculation is possible by noticing that some blocks within a program are guaranteed to execute no matter what combination of branches precede them. That is, the execution stream is guaranteed to converge at a later point. None of today's production architectures make use of *code convergence*, but there is ongoing research into this area. Two such projects are the Wisconsin Multiscalar machine [18] and the Waikato WarpEngine [2]. Both use convergence information to optimistically execute code, but the amount of speculation and quashing of incorrect execution varies.

The Wisconsin Multiscalar has multiple processors and uses a conventional MIPS instruction set with a few extra instructions. These extra instructions enable code to be broken up into blocks of contiguous instructions (tasks) that may contain conditional branches within them, and may themselves be conditionally executed. A unidirectional communications ring links the processors together with a committed execution token. This *trailing task* token is passed onto the next processor in the ring when the execution has completed on the current processor.

Processors are loaded with tasks that are predicted to be sequentially contiguous. During execution the instructions are executed as in a standard MIPS CPU, with each task considered to be a separate "program". Speculative memory operations are performed by tasks that are ahead of the trailing task and are recorded in the Address Resolution Buffer (ARB). If, at a later time, a load from a location has a new value written to it by a sequentially earlier store then the task the load was initiated from is quashed and re-executed.

Tasks that are placed on processors ahead of the trailing task are speculative. If, during execution of a task it is determined that a succeeding task has been incorrectly predicted then that task and all tasks ahead of it are quashed. The appropriate task is then placed on the first quashed task's processor.

If task invocation prediction is good and the data that is being manipulated is independent between tasks

then substantial performance gains can be made. But if bad predictions are made or highly coupled data is being manipulated then the high cost of quashing entire tasks reduces the potential performance gain. With this coarse resolution fine-grained independent flows of control may be unnecessarily quashed and consequently repeated.

To reduce the effects of quashing finer grained state saving can be done more often. The WarpEngine performs its state saving at the individual instruction level. When the result of a load is re-issued the instruction at the destination of the load request is re-executed. This has a cascading effect on the instructions on paths that are direct descendants of the re-issued load result and the amount of repeated work is thus minimised.

To achieve this fine grain a mechanism must be in place that retains the original ordering of the instructions. Also as instruction windows get larger the need for more memory accesses to happen out of order increases. The HP-8000 [9] and the Multiscalar [18] use hardware as a solution by ordering instructions/blocks in buffers/processors. The sizes of these ordering buffers are limited, therefore restricting the size of executable instruction window. A more generalised technique for retaining order information is required. The Time Warp algorithm is one such mechanism.

3. Time Warp

Time Warp is an algorithm for distributing and parallelising discrete event simulations originally proposed by Jefferson in 1982 [5], [11], [12], [21]. Parallelising discrete event simulations is a difficult problem because of the need to synchronise the simulation times of different parts of the simulation without violating the principle of causality [6]. A simulation is decomposed into objects that communicate via messages. Each message has an associated timestamp and when an object receives the message an event is executed at the time of the message. An event may generate further messages, which only affect later instructions - this is the principle of causality.

A naive algorithm for distributing simulations would have all the processors move their current simulation time forward in lockstep. However, this extracts very little parallelism. The Time Warp algorithm solves the synchronisation problem by locally executing events speculatively. Sometimes this will result in an object receiving a message out of order. The object is then rolled back to the correct point, that is, all the incorrect speculative events are undone, and the out-of-order event is then executed in its correct order.

Time Warp is interesting because it is able to extract parallelism from simulations that are otherwise intractable [5]. This paper will show that it is possible to map Time Warp onto the sequential execution of CPU instructions and thereby extract parallelism that is otherwise unavailable.

The first step in such a mapping is to identify what, in a sequential execution, corresponds to objects, events and messages. An object is the fundamental unit in Time Warp that is rolled back. At one extreme this could correspond to a single machine instruction. This is the approach taken in the WarpEngine [2], however, in the Multiscalar architecture [18] the basic unit is a task.

As well as being units of parallel execution objects also carry all the state information in Time Warp. Thus, in a CPU it is natural to identify individual memory locations as objects. Given this identification of objects there are a number of possible types of messages. Branch, call and jump instructions are control messages between instruction groups that communicate conditional and unconditional transfer of control. Write instructions are messages from an instruction group to a memory location and read instructions correspond to two messages: a request from the instructions to the memory location; and a reply from the memory back to the instructions.

To rollback an object to a particular event two things must be accomplished: restore the state of the object to what it was just prior to the event; and undo the effect of any messages sent by events after the rollback point. A common technique for restoring state is to take a copy of an object's state each time it receives a message. Then when a rollback occurs, the appropriate previous state can be restored. Rollback of instruction groups is thus similar to quashing speculative branches where the saved state corresponds to the committed registers on the speculative path. The amount of execution that needs to be undone on a rollback depends on the granularity and precision of the particular implementation.

In the HP-8000 the incorrect execution is undone by flushing the instruction buffer from the point of violation onwards, re-fetching the instructions and re-commencing execution. In the Multiscalar machine it corresponds to a squash. In both of these cases the state of the entire system is rolled back to the lasted saved state before the causal violation occurred.

Time Warp uses *anti-messages* to undo the effect of messages sent by events which have been rolled back. Each message that is undone has an anti-message sent to the same location. When an anti-message is received, it causes the object to be rolled back to a prior state and the anti-message annihilates the message. This may cause further anti-messages to be sent. The idea of an anti-message has not been explicitly identified in existing speculative CPUs that quash the whole state of the system following a causal violation.

One side effect of Time Warp objects periodically saving their state is that the object builds up a queue of old state copies. Hence, it is necessary to eventually garbage collect these old state copies (this is referred to as

fossil collection in the Time Warp literature). This can only be done when it is known that a state copy will never be required for a rollback. This is done by periodically estimating a value called the GVT (Global Virtual Time) which is equal to the minimum time of any currently active object or message between objects. Causality implies that an object can only be rolled back by a message from an object at an earlier time. So no object can possibly be rolled back to a time prior to GVT. Hence it is possible to fossil collect all state copies with timestamps less than GVT. In the case of groups of instructions this means that the instructions can be retired or committed and the registers holding their state can be reused. GVT is not a notion that has been explicitly identified in speculative architectures. In the Multiscalar architecture it corresponds to the earliest task.

GVT is also used to *commit* irreversible interactions with the outside world. For example, a request to print or a report of a fault cannot be honoured until GVT passes the time that the request was made. Thus any I/O device must queue and delay all operations until they can be committed.

Much attention has been paid to algorithms to compute GVT in the Time Warp literature [1]. GVT is analogous to the necessary in order retirement of instructions executed out of order in production architectures.

4. Designing a Time Warp based CPU

In designing a Time Warp based CPU there are three main issues that need to be addressed: mapping sequential code onto a parallel architecture, saving the state of instruction execution and logging all memory references so that causality violations can be detected and corrected.

As a first step toward parallelising sequential code using Time Warp, it can be 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 many nodes generate more than one child, otherwise control dependencies force sequential execution. For example, consider the code sequence shown in Figure 1. Figure 2 shows a possible execution tree for this sequence in which block A, the if-then-else statement and block E have been scheduled in parallel. In scheduling the if-then-else statement it may be possible to schedule one or both of C and D speculatively and in parallel with B.

E provides an example of code convergence. It will always be executed, irrespective of whether C or D is executed. Hence, it can be started at the same time as A and B (and possibly C and D) and execute in parallel with them. Looping structures can also be mapped onto this tree structure by scheduling multiple iterations speculatively in parallel. For more details on generating execution trees refer to [3].

```
A;
IF (B) THEN
    C;
ELSE
    D;
E;
```

Figure 1 Code Sequence

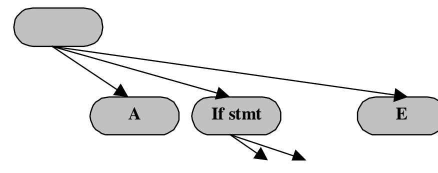

Figure 2 Possible Execution tree for code sequence in Figure 1

The second issue associated with the design of a Time Warp based CPU is state saving. The state of each task needs to be periodically logged so that execution can be rolled back (and re-started) should any causal violations be detected. The minimum amount of state saving that needs to be performed is the logging of preconditions for each task such that the task can be re-initiated if a violation is detected. This is the case in the Multiscalar machine. By allowing more frequent state saving it is possible to reduce the amount of execution that has to be re-satisfied as a result of a rollback. In the WarpEngine a task represents a basic block which is executed in a dataflow fashion. Instructions specify where their results will go (rather than where their arguments will come from). Each instruction has registers associated with it and when all its registers have values written into them the instruction can be executed. The results of executing an instruction are then stored in the registers of other instructions, which in turn cause them to be executed. When rollback occurs only the necessary instructions are re-executed.

The third issue that has to be addressed is the logging of memory references so that any causal violations that occur can be detected and corrected. The Multiscalar machine makes use of an Address Resolution Buffer (ARB) to store the locations of reads made by a given task. The addresses and values of writes from earlier tasks are passed to the ARB, where any violations are detected. This initiates a rollback of the task.

In the WarpEngine all memory references are logged in a time space cache [2], which sits on top of a conventional memory system. When a read request is received a value is returned and the transaction is logged. The returned value either comes from the time-space cache, or from the memory system below. Upon receiving a write request the time-space cache records the action and determines if reads need to be re-satisfied, in which case the new value is returned for those reads. Timestamps are used to maintain orderings within the time-space cache and are also used to determine which items can be flushed when fossil collection occurs.

5. Performance Issues

Results about the performance of Time Warp for discrete event simulation are surveyed in [6]. Good performance by Time Warp relies on a number of factors. First, the cascades of anti-messages induced by rollbacks must damp out quickly and not continue building. The overheads required to restore state on a rollback also cannot be too expensive. In practice, this means that the cost of recording state changes during forward execution must be low. Given all of this, Time Warp is capable of parallelising and speeding up problems that are otherwise intractable. Fujimoto [6] concludes that Time Warp is the only algorithm known which is capable of robustly speeding up a wide range of discrete event simulation problems, but that this is contingent on good implementations with low overheads.

If Time Warp principles are to be used in CPU design these performance issues still exist and need to be addressed. The first of these is the efficiency of state saving. For example, in the Multiscalar machine if a squash operation occurs, execution of the task has to be rolled back to the start of the task. To do this the Multiscalar machine saves the state of the CPU at the start of each task and all reads and writes during the task. In contrast the WarpEngine can rollback to any instruction and thus need only re-execute code directly dependent on the rolled back instruction. To do this it needs to save the state of every instruction. Thus, the cost of a rollback and re-execution is decreased, but more state needs to be saved than for the Multiscalar machine.

An example where parallelism can be gained by code convergence is inserting multiple items in a (balanced) binary tree. It is known that some number of items is to be inserted, but it is not known where they are to be inserted. When inserting one item around $m = \log_2 n$ branching decisions are made in travelling from the root to the leaves (where n is the number of items currently in the tree). To gain parallel speedup through speculative branch prediction would mean taking both targets of each branch and eventually undoing the incorrect one. To obtain a parallel speedup of P requires exploring 2^P different speculative branches with 2^P-1 of them being eventually undone [19]. This is an inefficient use of resources as the amount of possible parallelism increases only as the log of the size of the CPU hardware. A Time Warp based CPU can gain parallelism in this case without any need for branch speculation.

Each task, an individual insertion, is performed non-speculatively, that is, a branch in the search is done only when the values at a node are known. A Time Warp based processor can make use of code convergence to run each of the insertions in parallel. However , the tasks can each run in parallel. The parallelism will eventually be restricted by one insertion modifying memory locations read by later insertions that are thus rolled back. However, the amount of parallelism will increase with the size of the tree (there is less likelihood of a collision). Simulations of the dataflow parallelism for AVL and naive binary tree insertion on the WarpEngine are shown in Figure 3. As can be seen the amount of parallelism is high and increases with the size of the tree.

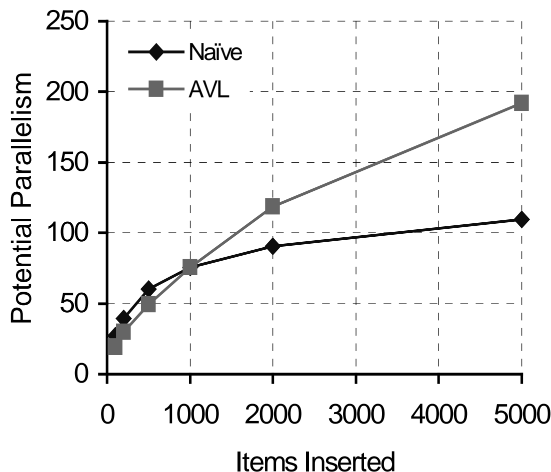

Figure 3 Potential Parallelism vs Problem Size for Binary Tree Insertion

It is worth noting that AVL tree insertion is difficult to parallelise using explicit synchronisation techniques. Any node in the tree can be modified by a rotation, so it is necessary to lock all nodes in the tree from the root to the leaf where insertion is done. Because of the lock on the root the execution is sequentialised preventing any significant parallel speed-up. Time Warp based CPUs, which do not require locks, allow the processor to extract the maximum amount of parallelism.

6. Conclusions

Time Warp is a mature algorithm that has shown great promise in parallelising otherwise intractable problems in discrete event simulation. This paper has

shown that Time Warp can be used to describe existing structures used in CPU design. In some cases concepts from Time Warp map directly to well known concepts in speculative and parallel CPU design. Other concepts such as GVT, rollback and anti-messages significantly generalise existing ideas in this area. At the very least Time Warp and its associated terminology should provide a way of talking about and comparing a wide range of possible schemes for parallelising instruction streams. However, we have shown some example programs where other parallelisation schemes such as control speculation or explicit locking are unable to extract significant parallelism but where Time Warp does provide good scalable parallelism. So at best a careful implementation of Time Warp may provide a route to obtaining significant additional instruction level parallelism without the need for explicit user level intervention.

7. References

[1] Bellenot, S., “Global Virtual Time Algorithms”, In D. Nicol, editor, Proceedings of the 1990 SCS multi-conference on Distributed Simulation, Vol. 22, No. 2, pp 122—130, San Diego, California, Jan. 1990

[2] Cleary, J. G., Pearson, M. and Kinawi, H., “The Architecture of an Optimistic CPU: The WarpEngine”, Proceedings of HICSS, vol 1 pp. 163—172., 1995

[3] Cleary, J. G., McWha, J. A. D. and Pearson, M., "Timestamp representations for Virtual Sequences", In Proceedings of Workshop on Parallel and Distributed Simulation, Jun. 1997 (In Press)

[4] Courtois, B., "CAD and Testing of ICs and Systems - Where are we going?", In Techniques of Informatics and Microelectronics for Computer Architecture, Grenoble, France, Nov. 1993

[5] Fujimoto, R. M., "Time Warp on a Shared Memory Multiprocessor", Transactions of the Society for Computer Simulation, Vol. 6, No. 3, pp 211—239, Jul. 1990

[6] Fujimoto, R. M., "Parallel Discrete Event Simulation", Communications of the ACM, Vol. 33, No. 10, pp 30—53, Oct. 1990

[7] Hennesey, J. L. and Joupii, N. P., "Computer Technology and Architecture: An Evolving Interaction", IEEE Computer, Vol 24, No. 9, pp 18—29, Sep. 1991

[8] Hennessy, J. L. and Patterson, D. A., "Computer Architecture A Quantitative Approach", Morgan Kuafmann Publishers, Inc, San Fansisco, second edition, 1996

[9] Hunt, D., "Advanced Performance Features of the 64-bit PA-8000", Compcon Digest of Papers, March 1995, pp. 123—128.

[10] Intel Corp. "Pentium Pro Processor at 150 MHz, 166 MHz, 180 MHz and 200 MHz", Datasheet, November 1995

[11] Jefferson, D. R., "Virtual Time", ACM Transactions on Programming Languages and Systems, Vol. 7, No. 3, pp 404—425, July 1985

[12] Jefferson, D. R., "Virtual Time II: Storage Management in Distributed Simulation", In Proceedings of the Annual Symposium on Principles of Distributed Computing, pp 75—89, Aug. 1989

[13] Lee, J. K. F., "Branch Prediction Strategies and Branch Target Buffer Design", IEEE Computer, Vol 17, No. 1, pp 6—22, Jan. 1984

[14] Lipasti, M. H. and Shen, J. P., "Exceeding the Dataflow Limit via Value Prediction", In Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture, Paris, France, Dec. 2—4, 1996

[15] MIPS Technologies Inc, "MIPS RISC Technology R10000 Microprocessor Technical Brief", October 1994

[16] Perleberg, C. H. and Smith, A. J., "Branch Target Buffer Design and Optimization", IEEE Transactions on Computers, Vol 42, No. 4, pp 396—412, Apr. 1993

[17] Smith, J. E. and Weiss, S., "PowerPC 601 and Alpha 21064: A Tale of Two RISCs", IEEE Computer, Vol. 27, No. 6, pp 46—58, June 1994

[18] Sohi G. S., S. Breach, S. and Vijaykumar, T. N., "Multiscalar Processors", 22th International Symposium on Computer Architecture, 1995

[19] Theobald, K. B., Gao, G. R. and Hendren, L. J., "Speculative Execution and Branch Prediction on Parallel Machines", In Proceedings of the 7th ACM International Conference on Supercomputing, Tokyo, Japan, July 20—22, pp 77—86, 1993

[20] Tomasulo, R.M., "An Efficient Algorithm for Exploiting Multiple Arithmetic Units" IBM Journal, Volume 11, January 1967, pp25—33

[21] Unger, B. W., Cleary, J., Dewar, A. and Xiao, Z., "A Multi-Lingual Optimistic Distributed Simulator", Transactions of the Society for Computer Simulation, Vol. 7, No. 2, pp 121—152, Jun. 1990

[22] Yeager, K., "The MIPS R10000 Superscalar Microprocessor", IEEE Micro, April 1996, pp. 28—40.