TimeWarp: a mechanism for extracting task-level parallelism

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

Department of Computer Science, University of Waikato, Hamilton, New Zealand
{jcleary,rhl,mpearson,jadm}@cs.waikato.ac.nz

The problem

A major problem facing computer architects over the next decade will be how to extract sufficient parallelism from existing code to sustain continuing increases in overall performance. Because of the huge installed base of code in languages with no explicit provision for parallelism it is necessary to do this from purely sequential programming languages. In recent years parallelism has been extracted at the micro-level where execution windows span a few tens of instructions and retirement rates up to four instructions per clock are achieved. Clearly to achieve one or two orders of magnitude increase in parallelism it will be necessary to spread execution over a much wider window of instructions.

Our solution

We are exploring three fundamental ideas to achieve this [2] based largely on the TimeWarp algorithm [1]. The first is that of using explicit timestamps on all memory reads and writes. These timestamps are ordered in the same way as the sequential code but allow more freedom in the order in which operations are actually executed. This is coupled with the second, the ability to roll-back and re-execute code when an out-of-order operation occurs. For example, a read may have to be re-executed if a prior write to the same address occurs late. The third idea is to use a tree structured control mechanism when issuing code. This allows adjacent sequential blocks or many iterations of a loop to be issued simultaneously. Such mechanisms allow parallelism to be extracted at the level of loops and other large blocks of code, a phenomenon sometimes referred to as task-level parallelism [3].

Such a system can, in principle, extract parallelism between loop iterations and blocks of code which have weak dataflow dependencies that cannot be determined statically. If the dependencies are strong then normal sequential code works better. If there are no dependencies and this can be determined statically then standard "parallel do" type constructs can be used to get perfect parallelism. That is, this work focuses on extracting parallelism in the intermediate cases between these two extremes.

Justification

We have been doing some simulations (on toy problems) and a number of interesting issues have arisen. The first is that care is needed in generating the explicit timestamps if they are not to quickly run out of precision. The second is that the ability to roll-back code requires the retention of information about previous execution states. The minimal information that must be retained is a record of all reads and writes. This state needs to be maintained in fast on-CPU storage. Unfortunately with some algorithms, as they scale up in size, the amount of parallelism that can be extracted reduces until, in the limit, only sequential execution is possible. The third issue surrounds the precision with which instructions are rolled-back and re-executed. In the Wisconsin Multiscalar machine [3] only task initialisation state is retained, forcing all execution following a roll-back point to be squashed. In our architecture, the *WarpEngine*, intermediate values are saved for each instruction which means that only instructions on the roll-back path are re-executed. For more information on the WarpEngine project see our web page: **http://www.cs.waikato.ac.nz/timewarp/wengine/index.html**

References

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

[2] Murray W. Pearson, Richard H. Littin, J. A. David McWha, and John G. Cleary. Applying Time Warp to CPU design. In *High Performance Computing Conference '97*, Bangalore, India, 1997. IEEE.

[3] G. S. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar processors. In *22nd International Symposium on Computer Architecture (ISCA-22)*, 1995.