

Scaling the Reorder Buffer to 10,000 Instructions

John Cleary, Richard Littin, David McWha and Murray Pearson
Department of Computer Science
University of Waikato
New Zealand
{jcleary,rhl,jadm,mpearson}@cs.waikato.ac.nz

Introduction

Much effort is being expended in current CPU design to increase the levels of instruction level parallelism (ILP). A key issue here is maximizing the size of the instruction window. Unfortunately, the centralized reorder buffer (ROB) of a modern superscalar CPU does not scale well. To overcome this limitation the WarpEngine architecture allows the large ROB's to be distributed [1]. The major features of the architecture are:

- The use of tree structured control to schedule large numbers of instructions;
- Support for speculation both on data values fetched from memory and on branches;
- The ability to re-execute individual instructions when speculation fails;
- The use of explicit time stamps to order reads and writes to memory (thus allowing memory operations to occur in any order); [2]

Other architectures which attempt to significantly increase instruction window size include the Multiscalar [3] and Trace [4] processors, although they are less aggressive in the number of in-flight instructions they consider.

WarpEngine Architecture

In order to issue large numbers of instructions in a short time the control is tree structured. For example, in the code sequence in Figure 1(a) the code for **A**, the conditional **B** and **E** would all be started in parallel as shown in Figure 1(b). This takes advantage of the fact that **E** will be executed no matter what the result is of the test in **B**.

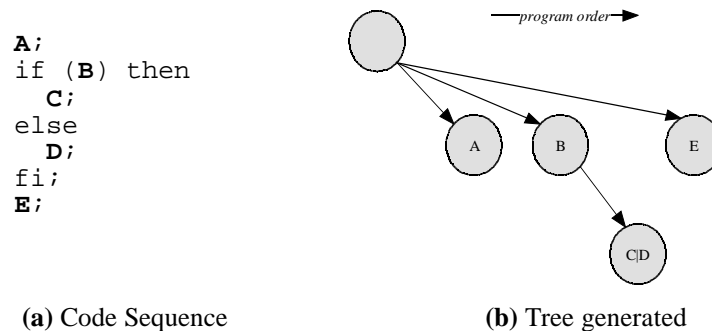


Figure 1 Tree-structured execution

The code executed by the WarpEngine is organized into fixed size blocks of instructions [5]. These are stored in hardware structures known as *frames*, which are made up of slots, each one containing an opcode, two source registers and the destination register locations. When the source registers contain valid data the instruction is scheduled for execution, much like a dataflow machine. The result is sent to the source registers of one or more other slots, and execution continues. There is no predetermined execution order within a frame, rather it is determined by data dependencies. To make the task of routing results to the next instruction tractable destination slots are limited to the current frame, or one initiated by the current frame. All other data must be routed through memory.

Speculative memory accesses are permitted in an unrestricted order, meaning there must be a mechanism for recovering from incorrect speculative loads. The memory system *rolls back* the load which received

the incorrect value and this is then cascaded down the data dependence chain formed by the result destinations and memory stores, until all the effects of the incorrect load have been revoked. The load is re-issued and the dependent chain of instructions re-execute with the correct values. To detect mis-speculated loads a record must be kept of all speculative memory accesses (both loads and stores), along with a way of determining their causal order. These records are stored in the *time-space cache*, which sits between the standard non-speculative memory hierarchy and the CPU. This consists of multiple versions of the value stored in each memory location, along with a time stamp indicating the region of the instruction window that value is valid for. When a store occurs a new value is added to this record and the record of loads from that address is checked for any dependent loads, which are re-issued with the new value.

Undoing the effects of memory mis-speculation on the instruction stream requires support from the frame. The frame effectively saves the state of the execution by allowing it to recover to the point prior to execution of any instruction. Rather than recording the state of the processor at a checkpoint, it allows the effect of each operation to be undone. This is done selectively at an instruction-level granularity to avoid rolling back correct processing, but requires a larger amount of storage than coarse granularity checkpoint methods. The ROB serves the same purpose in out-of-order superscalar architectures, albeit on a smaller scale.

The WarpEngine orders memory accesses by associating an explicit time stamp with each. The time stamps have the same order as a sequential execution of the instructions. Generating such time stamps has turned out to be challenging [6]. A naïve way to do this would be to make each time stamp a 32 bit binary string recording the path from the root of the control tree to the instruction. This runs out of precision as soon as the tree is 32 levels deep.

A number of solutions to this have been examined. For example, it is possible to force all frames to sit in hardware in the order they are generated. This allows simple disambiguation of memory accesses, but forces frequent copying of frames and can drastically reduce the available parallelism, leading to poor performance. At the other extreme every frame carries an explicit time stamp and can be scheduled into any available slot. In this case time stamps are hard to generate and memory disambiguation is expensive. Our current research is focussed on mixed strategies that lie between these extremes and which give better performance than either.

One solution to the problem of generating time stamps has been very fruitful. In this, the application code explicitly estimates the number of frames which are descendants of each frame. Consideration of actual code shows that frequently (but not always) tight upper bounds can be placed on these numbers. This allows time stamps to be allocated according to need and, in practice, dramatically reduces the number of cases where time stamp precision is exceeded.

The reason this approach is so fruitful is that it helps solve general problems of resource allocation. For example, when a frame is generated it is necessary to allocate hardware resources to it. By using the same explicit estimation process as for time stamps it is possible to schedule long sequences of frames adjacent to one another. This speeds both the scheduling of frames and, via the mixed strategy alluded to above, simplifies the structure of the time-space cache.

Evaluation

A high level simulator has been developed for the WarpEngine architecture and the results in this section were obtained using it [7]. Some architectural features have been idealized and then selectively constrained to examine the upper limit they place on execution. In the results that follow an unrestricted number of functional units are assumed, along with penalty-free rollback and unlimited instruction issue and retirement width. These are only constrained by the time taken for data to flow through the system when re-execution is initiated. In the simulation results that follow the size of the ROB is restricted in order to show its effect on the limit of parallelism that can be extracted.

Simulation results for parallel AVL tree insertion on the WarpEngine containing a ROB of 8,192 instructions are shown in Figure 2. The graph shows that an IPC of approximately 30 has been achieved. The lower two lines in the graph roughly correspond to ROB sizes in contemporary production

architectures. It is worth noting that AVL tree insertion is difficult to parallelize using explicit synchronization techniques as 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. As the WarpEngine does not require locks, it is able to extract the maximum amount of parallelism.

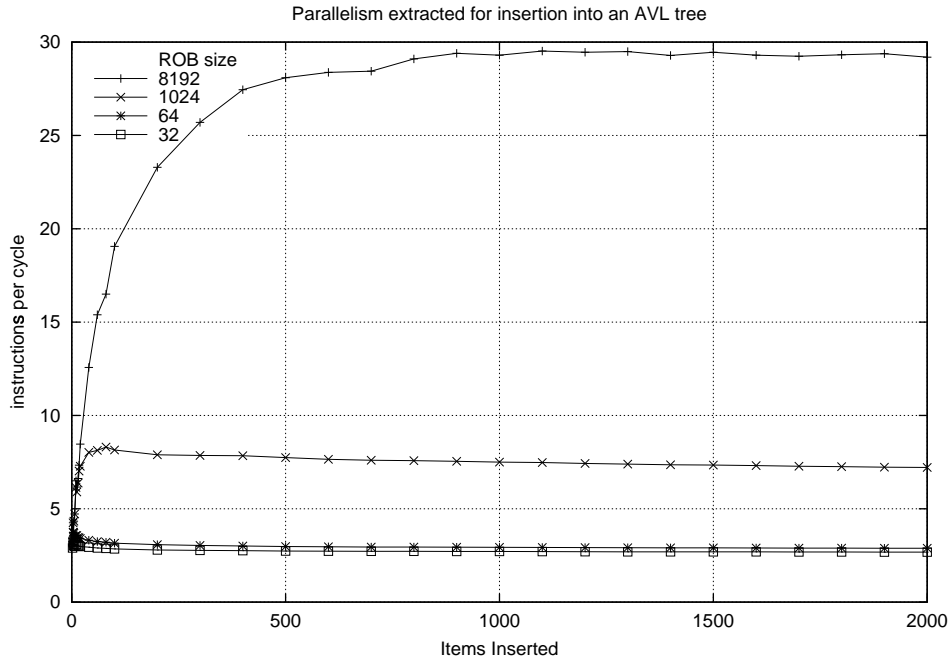


Figure 2 IPC for AVL tree insertion

Figure 3 shows the ROB usage throughout the execution of a program inserting 200 items into an AVL tree using an 8,192 instruction ROB. The 'valid' region shows slots occupied by instructions executing using correct data, the 'wasted' region shows slots occupied by instructions executing with incorrect data, which will ultimately be rolled back. The largest region, marked 'waiting' shows slots containing instructions which have completed execution correctly, but which cannot be retired because they are still speculative. This shows that a large proportion of the ROB is lying idle, restricting speculation and, ultimately, execution. The reason for this is that most code contains long chains of data dependencies that leave later instructions waiting for earlier ones to retire before they can be retired themselves. This phenomenon can lead to the paradoxical result that, for a fixed instruction window, increasing the size of a problem can lead to less parallelism extracted, and worse performance. As the size of the problem increases the inherent parallelism of the problem generally increases, but the ROB is used less efficiently, with a higher percentage of slots containing idle instructions. When a straggling instruction completes many instructions retire and the white area of Figure 3 shows slots which are empty while the ROB is being refilled.

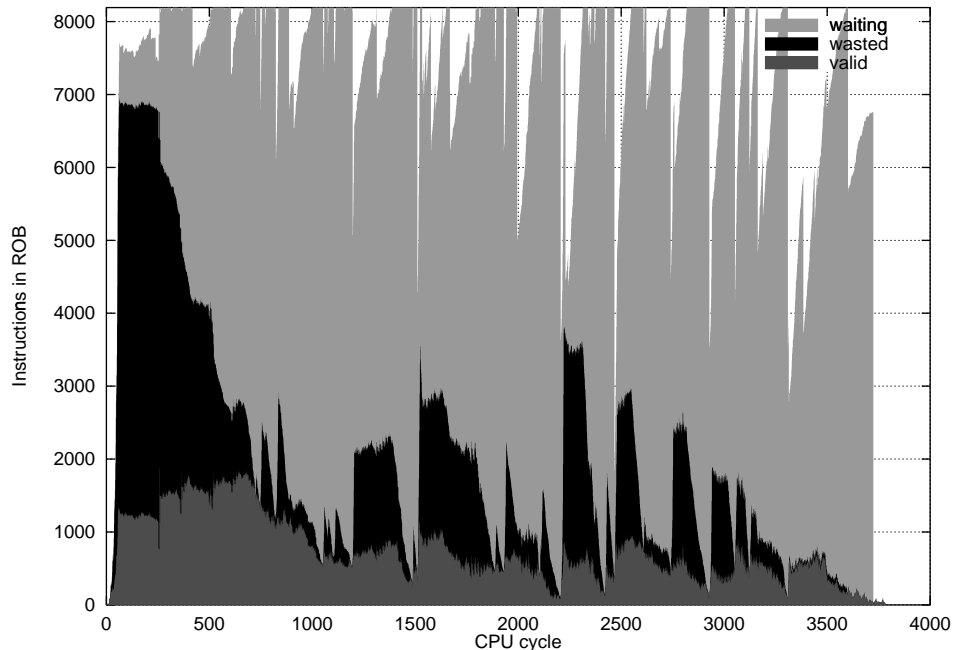


Figure 3 Reorder buffer usage for inserting 200 items into an AVL tree

Finding ways of retiring instructions out-of-order is one way to further improve performance. It is an open question how to do this. One approach is to take loops with independent iterations and retire them in parallel. Another is to allow user annotations to indicate blocks of code that are independent of each other (although this is counter to the philosophy of the WarpEngine which is to consider only ILP that can be automatically extracted either by a compiler or the CPU). Another possible way of making better use of state resources is to gradually retire instructions to storage that is slower, cheaper, and which contains less detail about the instructions. Thus an instruction which executed a long time ago might have the details of the instruction discarded and its parameters retired to RAM (of course there would be a high penalty for re-fetching the information should a speculation fault occur).

Conclusions

Simulation of the WarpEngine using ROB's of up to 10,000 instructions has taught us important lessons about how to design a CPU with very high levels of parallelism. The good news is that the parallelism is available in a range of programs including those that traverse deep and complex data structures that are not traditionally easy to parallelize. The bad news is that a number of issues need to be solved carefully and well to do this. To generate enough instructions for execution some form of tree structured code is essential. To be able to solve resource allocation problems, including generation of time stamps, it is necessary to estimate the size of sub-trees in the tree structured control graph. Space in the ROB to store the state of in-flight instructions is a limiting resource in many cases. More work is needed to characterize solutions to this problem, including: tradeoffs in the granularity and precision of instruction re-execution; techniques for out-of-order retirement; and compaction of old, but not retired, instructions.

The WarpEngine architecture includes an unusual block based design with instructions that say where their results are to be sent, rather than where operands are to be fetched from. In hindsight these are interesting ideas, but not critical to solving the problem of very large ROB's.

References

- [1] R. H. Littin, "The WarpEngine Project", <http://www.cs.waikato.ac.nz/timewarp/warpengine/> (current 18 February 2000).

- [2] M. W. Pearson, R. H. Littin, J. A. D. McWha and J. G. Cleary, "Applying Time Warp to CPU Design", High Performance Computing Conference 1997, Bangalore, India.
- [3] G. S. Sohi, S. Breach, and T. N. Vijaykumar, "Multiscalar Processors", the 22nd International Symposium on Computer Architecture, 1995, pp. 414–425.
- [4] E. Rotenberg, Q. Jacobsen, Y. Sazeides and J. Smith, "Trace Processors", the 30th International Symposium on Microarchitecture, December 1997.
- [5] L. Eeckhout, K. De Bosschere and H. Neefs, "On the Feasibility of Fixed–Length Block Structured Architectures", the 5th Australasian Computer Architecture Conference, January 2000, pp. 17–25.
- [6] J. G. Cleary and J. A. D. McWha and M. Pearson, "Timestamp Representations for Virtual Sequences", 11th Workshop on Parallel and Distributed Simulation, June 1997, pp. 98–105.
- [7] R. H. Littin, "Design and Evaluation of an Optimistic CPU: the WarpEngine", PhD thesis, University of Waikato, Hamilton, New Zealand, January 2000.