Designing Ultra-Large Instruction Issue Windows

Doug Burger

Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712 USA
Email : dburger@cs.utexas.edu

Abstract. To continue historical rates of improvement, future high-performance processors are likely to exploit more instruction-level parallelism. The best way to find much of that parallelism is by implementing an out-of-order issue core with an ultra-large issue window. However, there are serious challenges in building large issue windows that can hold hundreds or thousands of instructions, including how to build them, how to fill them, and how to empty them efficiently. In this paper, we describe some of the solutions proposed by other researchers that address the limitations currently constraining issue window sizes. We also describe the solutions being incorporated into the University of Texas TRIPS processor, which will contain a 1024-instruction window in each processor core.

1 Introduction

Commodity microprocessors have shown enormous performance gains over the past three decades, typically cited at 55% per year. Over the past 15 years, the bulk of those performance improvements have come from faster clock rates, improving at 40% per year, and even faster recently. This rate of growth is unsustainable, however, as pipelines are nearing their optimal depths [6]. Once clock rates reach that point in the next few years, the most promising source of continued performance improvements is increased parallelism, whether it be coarse-grained parallelism on a multiprocessor, or increased exploitation of instruction-level parallelism (ILP). Given the difficulties inherent in parallelizing irregular codes, and the lack of success in doing so over the past decades, we believe that striving for increased ILP is a more promising approach.

In this short paper, we discuss how out-of-order issue cores can exploit large windows of instructions to achieve higher ILP. These windows, which may eventually hold thousands of instructions, have enormous implementation challenges, particularly in the face of emerging technology constraints such as power ceilings and multi-cycle wire delays.

The major challenges associated with these kilo-windows of instructions (KWIs), are four-fold:

1. Implementing a kilo-instruction window
2. Filling a kilo-instruction window
3. Flushing a kilo-instruction window
4. Emptying a kilo-instruction window

Each of these challenges will require new techniques to solve effectively. We describe both the issues for conventional processors as well as the solutions being incorporated into the TRIPS processor prototype being designed at the University of Texas at Austin. The TRIPS processor will be the first example of a KWI; the each of the four processor cores on the prototype chip will issue instructions out of order from a window of at least one thousand instructions.

In the rest of this paper, we describe the challenges and current solutions to the four challenges described above. If solved, these challenges will provide significant performance increases, since past studies have shown that ILP is available out to thousands of instructions [2]. Recent studies have also shown that long-latency operations, such as cache misses to main memory, can be tolerated with ILP so long as the instruction window is sufficiently large and branch mispredictions do not invalidate a significant fraction of the window [8].

2 Implementing a KWI

Building large conventional, centralized issue windows is infeasible given four issues: growing on-chip wire delays [1], the quadratic growth of window complexity with issue width [20], long latencies for large windows, which must match broadcast register tags against every entry associatively, and the power limitations of building large associative structures.

Researchers have proposed techniques for building large but scalable windows. These approaches include building hierarchical windows, where a small, fast one backs up a larger, slower one [11, 5], clustered processors [9], or dependence-based queues [13, 10].

The TRIPS processor, conversely, implements a physically partitioned window [12, 17] that distributes an issue window among multiple execution units, treating them jointly as reservation stations, an issue window, operand buffers, and a distributed reservation station. These functionalities were originally merged into one structure in the Register Update Unit (RUU) [19]. The TRIPS window organization has two major differences from the RUU. First, the TRIPS window is a highly partitioned structure, with a partition at each execution unit (which number from 8 to 64, depending on the issue width of the processor core). Second, the mapping of instructions to slots in the window is performed partially by the compiler, since the TRIPS processor employs a Static-Placement, Dynamic Issue (SPDI) execution model.

This compiler mapping enables dependent instructions to be placed close together, and it also permits each partition of the window to be constructed with non-associative logic. Since the instruction set specifies that each instruction contain the physical locations of its consumers, an instruction can send its operand directly to the exact window slot where a consuming instruction is guaranteed to be buffered. Thus, while the TRIPS approach requires a change in instruction set and execution model, it supports out-of-order execution with a scalable issue window, which can grow linearly with the number of execution units. There are also power advantages to this approach, as the issues window is both partitioned and avoids the need for power-hungry, high-latency CAMs to implement associative lookups for waking up instructions.

3 Filling a KWI

The second major challenge for building feasible KWIs, aside from building scalable and practical physical structures, is filling them with useful instructions. The two challenges for filling KWIs is high-bandwidth fetch and effective prediction. We discuss each below.

3.1 High-Bandwidth Fetch

Much research has focused on increasing the bandwidth of the front end instruction fetch unit. There are several challenges to sustaining the levels of bandwidth required to keep large windows full at high rates of ILP, including fetching past multiple branches per cycle, renaming many instructions per cycle, and dispatching many instructions per cycle to the issue window.

Techniques to improve instruction fetch bandwidth include trace caches [16], fetch target buffers (FTB) [15], and many more. Trace caches are an example of a technique that deals with branches by crafting a linear sequence of instructions dynamically, whereas the fetch target buffer exploits idle time to run ahead of the actual front end fetch rate. By running ahead, the FTB prevents performance loss when the program enters brief periods when the front end cannot sustain enough bandwidth for the machine (e.g., too many predicted taken branches).

The solution that we employ in the TRIPS processor is to use large hyperblocks as the unit of fetch and map. Hyperblocks are predicated regions of code that have only one entry point, but which may have multiple exits. We couple these hyperblocks with an exit predictor that chooses the first taken exit branch in a hyperblock [14]. By making only one prediction per hyperblock (and implicitly predicting all the branches before the predicted exit in that hyperblock), a large number of instructions–80 on average–can be fetched with each prediction.

Due to the SPDI execution model, in which the compiler places instructions in a fixed-format block, the instruction caches can be distributed to rows of ALUs, columns of ALUs, or individual ALUs. When a prediction is made, the global controller looks up the address produced by the branch target buffer in the instruction cache tags. If an I-cache hit occurs, the controller broadcasts the correct index to all distributed I-cache banks, which proceed to fetch their portion of the statically mapped block in parallel. The exit predictor, coupled with a BTB and the distributed I-cache banks, can run ahead with its predictions, similar to an FTB, but with many more instructions per prediction and a much higher sustainable bandwidth from the distributed I-cache array.

3.2 Prediction

For most irregular codes without regular, predictable loops, mispredictions will result in a small fraction of a KWI being utilized. Currently, integer codes (such as SPECINT2000) demonstrate a rate of two to ten Mispredictions per Thousand (kilo) Instructions (MPKI), with an average of roughly 5 [18]. By dividing 1000 by the average MPKI for a benchmark, the average number of useful instructions fetched before a misprediction can be obtained. Even the most accurate predictors currently proposed in the literature, such as the perceptron predictor [7], cannot achieve under 1 MPKI for most benchmarks, indicating that if straight branch prediction is to be used, considerably more accurate predictors will need to be developed. Predication of branches has been proposed to reduce the rate of branches that must be predicted, but it does not typically improve the predicatility. when an unpredictable branch is removed, the removal often pushes the poor predictability onto other branches [3].

Simulation results show that the TRIPS processor currently loses 33-50% of its potential performance to branch mispredictions. The approach that we are taking is to be more aggressive with if-conversion and loop unrolling, forming larger hyperblock regions for fetching that contain multiple paths, thus trading useless instruction overhead for better predictability. Predictability is improved, however, only when basic blocks that reside on multiple control paths are added to a hyperblock until they re-join in the control flow graph. If-converting to control flow merges allows the processor to exploit control independence in a clean manner, since the successor block is predictable. The challenges to this approach are (1) providing enough buffering and execution resources that the overhead (non-taken path) instructions do not impede performance, and (2) ensuring that the non-taken paths included in the mapped blocks do not have critical paths significantly longer than the taken paths. The balancing of these path lengths and the decisions about which paths to include is made at compile time, and is an active area of research.

4 Flushing a KWI

When a control misprediction occurs in a large window, an enormous number of in-flight instructions may be invalid. Future systems will benefit from keeping the flushing and recovery costs as low as possible. This problem is fairly simple to solve in a conventional processor, which typically defers handling of the misprediction (or synchronous exception) until the faulting instruction reaches the head of the reorder buffer, at which point the entire pipeline is flushed.

That solution is considerably less attractive in a distributed, large-window microarchitecture with high communication delays, since it may take many cycles for the faulting instruction to become the oldest instruction. We are investigating two techniques to reduce the performance losses due to flushes, the first of which reduces the overhead of the flush, and the second of which reduces the frequency of flushes.

T ag-based Flushing: An alternative to waiting for the faulting instruction to complete is to actively squash only the mis-speculated instructions in flight. Explicit squashing of all in-flight, mis-speculated instructions is particularly difficult in a distributed microarchitecture. The approach that we are exploring involves tagging each block that is mapped to the execution substrate and updating and broadcasting a tag that indicates that all operations past a mapped block are now invalid. This approach is similar to how conventional microarchitectures handle mis-speculated loads that return from memory after a pipeline has been flushed and re-filled with correct work. Even more efficient would be simply injecting new blocks with updated tags when a misprediction was detected, requiring little waiting time at all. This scheme adds both the complexity of tag management and a verification challenge–since old and new operations may be in flight together–but permits lower-latency flushes.

D istributed Selective Re-Execution: The other approach that we are currently exploring is to minimize the frequency of complete flushes. The pipeline is flushed on control mispredictions, but for other kinds of speculative violations, such as a load/store ordering–or any violation that involves the right instructions computing with the wrong data–we perform selective re-execution, re-firing only the instructions that depended on the faulting instruction. With this approach, when the right kind of misprediction occurs, the pipeline does not need to be flushed, no useful work is thrown out, and no instructions need to be re-fetched. We describe a protocol for achieving selective re-execution in a distributed microarchitecture elsewhere [4].

5 Emptying a KWI

In conventional architectures that commit one instruction per cycle, on average, draining the instruction window of completed instructions is relatively straightforward. The Alpha 21264 [9] is able to commit up to 11 instructions per cycle, but can only commit one branch per cycle.

In a distributed microarchitecture, however, commit is significantly more difficult. Determining the correct order to remove the instructions from the partitions is both necessary and challenging, particularly in a multiprocessor where ordering must be maintained to satisfy memory consistency models. Stores are typically written back to the memory system at commit, and register values to the architectural register file.

A centralized structure to track orderings of written stores, pipelining permissions across multicycle communication delays, is one feasible approach. One drawback of this approach is that a cache miss on a store could quickly make the commit stage a bottleneck. The store issue argues for write-back, write-noallocate level-one caches. It is not clear that commit will be a bottleneck for KIW machines, but it is certainly possible that new approaches will need to be devised to permit these machines to get instructions *out* of the pipeline sufficiently fast.

6 Summary

Future performance gains for uniprocessors, above and beyond those afforded by faster transistors, will have to come from either instruction or thread-level parallelism. Many (if not most) workloads still do not lend themselves to easy parallelization or multithreading. Consequently, instruction-level parallelism will likely grow significantly in importance in the coming decade.

However, to maintain greater ILP, it is likely that the research community will need to develop wide-issue out-of-order cores that can sift through many more instructions than today to find enough ready-to-issue instructions each cycle. These large windows have a number of daunting implementation challenges, including filling the window, designing a practical window, deallocating instructions for commit, and efficient flushing of the window upon a misprediction.

The TRIPS processor prototype being designed at the University of Texas will have 1024-entry, distributed instruction issue windows in each processor core (with four processors on each chip). We have solved several of the challenges of designing a practical kilo-window, and have shown how to fill it quickly with a high-bandwidth front end. We are working on the compiler technology to permit predication to control-flow merge points in an attempt to improve the predictability of the instruction stream and thus fill the window for irregular benchmarks. If this effort fails, alternative methods to improve the predictability must be found. Currently, we are still in the process of designing our commit and flush logic, which may employ some of the principles described in this paper. We hope to have a working prototype by the end of 2005, successfully demonstrating solutions to the problems enumerated in this paper.

References

1. V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 248–259, June 2000.
2. T. M. Austin and G. S. Sohi. Dynamic dependency analysis of ordinary programs. In *Nineteenth International Symposium on Computer Architecture*, pages 342–351, Gold Coast, Australia, 1992. ACM and IEEE Computer Society.
3. Y. Choi, A. Knies, L. Gerke, and T.-F. Ngai. The impact of if-conversion and branch prediction on program execution on the intel itanium processor. In *Proceedings of the 34th International Symposium on Microarchitecture*, pages 182–191, Dec. 2001.
4. R. Desikan, S. Sethumadhavan, R. NAgarajan, D. Burger, and S. L. keckler. Lightweight distributed selective re-execution and its implications for value speculation. In *Proceedings of the First Value Speculation Workshop, Associated with ISCA-30*, June 2003.
5. D. Ernst and T. Austin. Efficient dynamic scheduling through tag elimination. In *Proceedings of the 29th International Symposium on Computer Architecture*, May 2002.

6. M. Hrishikesh, N. Jouppi, K. Farkas, D. Burger, S. Keckler, and P. Shivakumar. The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 14–24, May 2002.
7. D. Jimenez and C. Lin. Dynamic branch prediction with perceptrons. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, pages 197–206, Jan. 2001.
8. T. Karkhanis and J. Smith. A day in the life of a cache miss. In *Proceedings of the 2nd Annual Workshop on Memory Performance Issues (WMPI02)*, 2002.
9. R. Kessler. The alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.
10. H. Kim and J. Smith. An instruction set and microarchitecture for instruction level distributed processing. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 71–81, 2002.
11. A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A large, fast instruction window for tolerating cache misses. In *Proceedings of the 29th International Symposium on Computer Architecture*, May 2002.
12. R. Nagarajan, K. Sankaralingam, D. Burger, and S. W. Keckler. A design space evaluation of grid processor architectures. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, pages 40–51, December 2001. Submited by: Ramadass Nagarajan.
13. S. E. Raasch, N. L. Binkert, and S. K. Reinhardt. A scalable instruction queue design using dependence chains. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 318–329, May 2002.
14. N. Ranganathan, R. Nagarajan, D. Burger, and S. Keckler. Combining hyperblocks and exit prediction to increase front-end bandwidth and performance. Technical Report TR2002-41, Department of Computer Sciences, The University of Texas at Austin, Austin, TX, September 2002.
15. G. Reinman, B. Calder, and T. Austin. A scalable front-end architecture for fast instruction delivery. In *Proceedings of the 26th International Symposium on Computer Architecture*, June 1999.
16. E. Rotenberg, S. Bennett, and J. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*. ACM, 1996. Submited by: simha.
17. K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. Keckler, and C. Moore. Exploiting ilp,tlp, and dlp with the polymorphous trips architecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, May 2003.
18. A. Seznec, S. Felix, V. Krishnan, and Y. Sazeides. Design tradeoffs for the alpha ev8 conditional branch predictor. In *Proceedings of the 29th International Symposium on Computer Architecture*, June 2002.
19. G. S. Sohi. Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers. *IEEE Trans. Comput.*, 39(3):349–359, March 1990.
20. J. S. Subbarao Palacharla, Norman Jouppi. Complexity-effective superscalar processors. In *Proceedings of the 24th International Symposium on Computer Architecture*, 1997. Submited by: Hrishi.