Tolerating Cache-Miss Latency with Multipass Pipelines

MULTIPASS PIPELINING USES PERSISTENT ADVANCE EXECUTION TO ACHIEVE MEMORY-LATENCY TOLERANCE WHILE MAINTAINING THE SIMPLICITY OF AN IN-ORDER DESIGN.

Ronald D. Barnes
George Mason University

Shane Ryoo
Wen-mei W. Hwu
University of Illinois, Urbana-Champaign

Microprocessors exploit instruction-level parallelism and tolerate memory-access latencies to achieve high-performance. Out-of-order microprocessors do this by dynamically scheduling instruction execution, but require power-hungry hardware structures. Other general-purpose microprocessors, such as the Sun UltraSparc[1] and Intel Itanium,[2] and most embedded microprocessors, avoid such costs, relying instead on the compile-time scheduling of instruction execution.

Appropriate instruction set architecture features—large register files, explicit speculation, and predication—help compilers successfully exploit instruction-level parallelism. In the absence of runtime delays, such as cache-miss-induced stalls, the compiler-arranged schedules can effectively use execution resources, overlap execution latencies, and work around execution constraints.[3] For example, if we discount runtime stall cycles, the Intel reference compiler achieves an average throughput of 2.5 instructions per cycle (IPC) across SPECint2000 benchmarks for a 1.0-GHz Itanium 2 processor.

However, the compile-time scheduling approach has had less success in tolerating memory access latencies. Memory access latencies can't be determined reliably at compile time and can vary by orders of magnitude, according to the accessed memory data's runtime position in the multilevel cache/dynamic RAM (DRAM) hierarchy. Data cache misses significantly prolong compile-generated schedules during runtime, causing data cache stalls to disproportionately affect the performance of in-order processors. Clearly, in-order microprocessors need to better tolerate memory access latencies without resorting to power-hungry solutions.

This article describes *multipass pipelining*, a microarchitectural model that provides an alternative to out-of-order execution for tolerating memory access latencies. We call our approach "flea-flicker" multipass pipelining because it uses two (or more) passes of preexecution or execution to achieve performance efficacy. In American football, a flea-flicker offense tries to catch the defense off guard by adding a forward pass to a lateral-pass play. Defenders covering the ball carrier thus miss the tackle and, hopefully for the offense, the ensuing play.

Multipass pipelining assumes compile-time scheduling for lower-power and lower-complexity exploitation of instruction-level parallelism. Unlike traditional in-order processors, multipass allows persistent advance execution of instructions beyond those that are interlocked on a cache miss. In a multipass

Published by the IEEE Computer Society

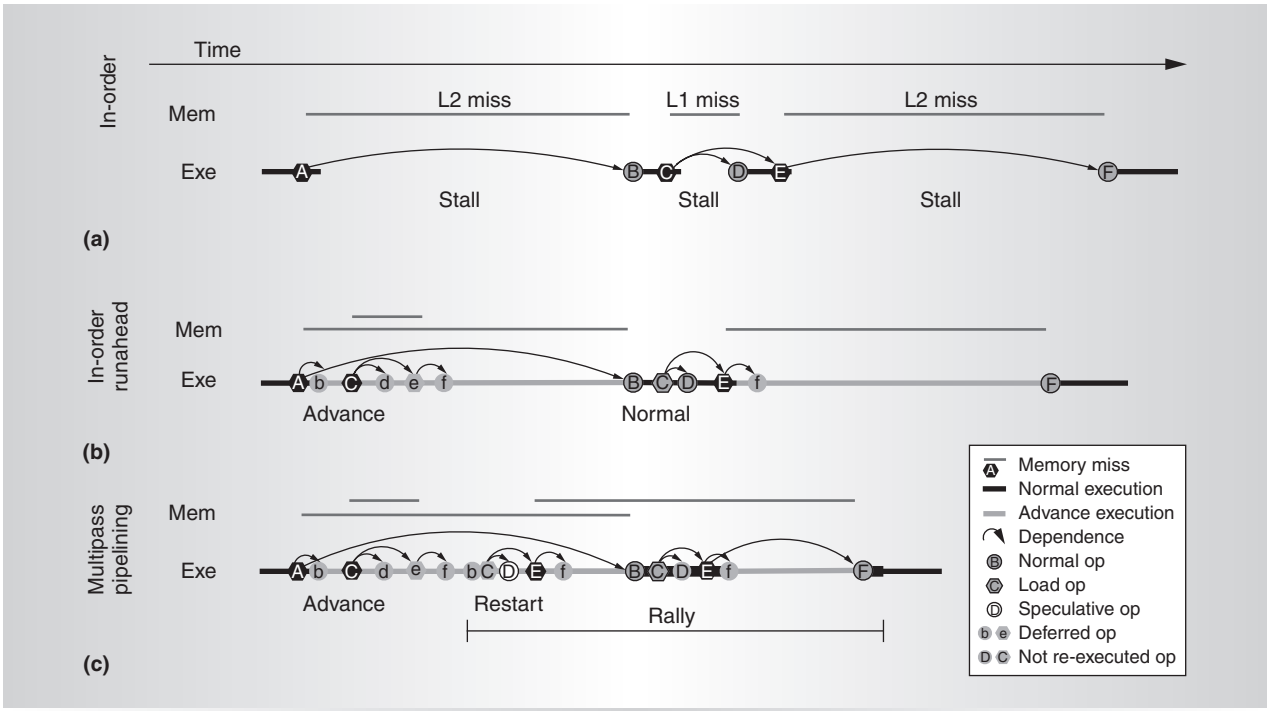

Figure 1. Execution and memory-access timeline for three in-order execution models: basic (a), in-order runahead (b), and multipass pipelining (c).

pipeline, instructions continue to execute after an instruction that needs the result of a cache miss. The pipeline skips over instructions that aren't ready and persistently executes those that are ready. We achieve this through multiple, carefully controlled in-order execution passes over instructions following the interlock. A low-complexity result buffer preserves the valid results of independent instructions. Because persistent advance execution requires additional hardware components, we examine its benefit relative to the more traditional in-order execution models.

In-order operation models

Figure 1 shows an example timeline repeated for three different execution models. For each model, we divided the execution activity into actual instruction execution (Exe) and the handling of data-cache misses caused by executing load instructions (Mem). In each example, the Exe line represents many executing instructions. Arrows represent data dependences between instructions. Figure 1 shows two types of misses: relatively long level-2 cache misses (L2 miss) and relatively short level-1 cache misses (L1 miss).

Figure 1a demonstrates the most hindersome problem accompanying in-order processors: Instructions can artificially stall behind consumers of load instructions that missed in the cache. In the example, load instruction A misses in the data cache, and a stall on use occurs when execution reaches instruction B, A's first consumer. Because the processor relies on the compiler's instruction ordering, it stalls for the remaining duration of A's miss (marked as *stall*). D and F trigger similar execution gaps.

Figure 1b illustrates the in-order runahead approach, first introduced by Dundas and Mudge.[4] In-order runahead execution reduces execution gaps by increasing the overlap between cache-miss handling of independent memory loads. In this example, when B attempts to use A's result before the cache-miss handling completes, the runahead approach lets execution continue speculatively. We show this speculative execution—or *advance execution*—as a faint continuation of the Exe line, marked as *advance* in the timeline beyond deferred instruction b.

A replicated register file preserves a checkpoint of the processor's register state when speculative advance execution begins. During

advance execution, instructions such as b don't have operands required to compute a valid result. The pipeline suppresses b's execution, and b writes a specially marked nonresult to its consumers and destination. The pipeline will suppress any instructions that consume b's result, delivering specially marked nonresults to their consumers and destinations. In the example timeline, execution continues past b and reaches independent instruction C, which can begin its memory access. C's access will overlap **C** with that of A. In Figure 1b, the overlapping bold lines in the timeline's Mem component demonstrates the main source of performance improvement from the in-order runahead approach.

When A's access is complete, the processor returns to its checkpointed state. Instruction execution returns to B, which now has the operand needed for its execution. Because C's miss was already initiated during advance execution, instruction D no longer causes the pipeline to stall after normal execution resumes. Runahead execution is a relatively inexpensive addition to an in-order processor: Replicated processor state to support checkpointing, propagation of special invalid operand markings from instruction to instruction, and suppression of instructions whose operands are unavailable during advance execution.

Multipass pipelining

Multipass pipelining enhances the in-order runahead approach. As Figure 1c shows, multipass pipelining differs from in-order runahead in two important ways.

First, during advance execution (at the point labeled *restart*), advance execution restarts at original consumer instruction b. Although b isn't ready for execution, advance execution restarts in the hope of finding additional instructions in the queue that might be newly ready for execution. In the example timeline, during the second pass, the short cache-miss handling for C has completed, and *E* can trigger its cache-miss handling. Advance restart is particularly useful when the speculative processor becomes so polluted with nonresults that continued advance execution is fruitless. Advance execution restart returns to previously ignored instructions rather than wasting advance execution effort further down the instruction stream.

The additional overlap of *E*'s cache-miss handling reflects the fact that multipass pipelining addresses a limitation of in-order runahead: When the in-order runahead pipeline suppresses an instruction during advance execution, it won't reconsider that instruction until normal execution begins again. Advance execution restart in multipass pipelining lets the suppressed instructions whose operands become available during a long cache miss have another chance to trigger their cache-miss handling.

Figure 2 illustrates an implementation of multipass pipelining on a base processor pipeline similar to that of the Itanium 2. Multipass pipelining requires additional components, indicated by the shaded boxes. We use a speculative register file (SRF), advance bits (A-bits), and invalid bits (I-bits) to separate the register state for advanced execution from the normal architectural execution state and to implement the suppression of instructions whose operands are unavailable during advance execution. These components are similar to those that support in-order runahead execution; they are shut off during normal execution.

During advance execution, instructions aren't allowed to write their results to the architectural register file (ARF). Instead, the multipass pipeline redirects their results to the SRF, which stores the speculative processor state for the current advance execution pass. When the pipeline enters a new advance execution pass, the SRF contains no valid information; advance instructions must initially access ARF for their input operands. As advance instructions write into SRF, A-bits redirect consumers of the instructions' results to SRF for their operands. Each A-bit indicates that future accesses to its associated register entry should go to SRF.

Each SRF entry contains an I-bit that marks nonresults written by suppressed advance instructions. Advance instructions reading a register with a set I-bit are suppressed. Bypass logic will forward the I-bit value in addition to register values. When normal execution resumes, ARF will contain the checkpoint register state. A-bits, I-bits, and SRF are no longer necessary and can be shut off for energy efficiency.

We added the shaded latches in Figure 2 to

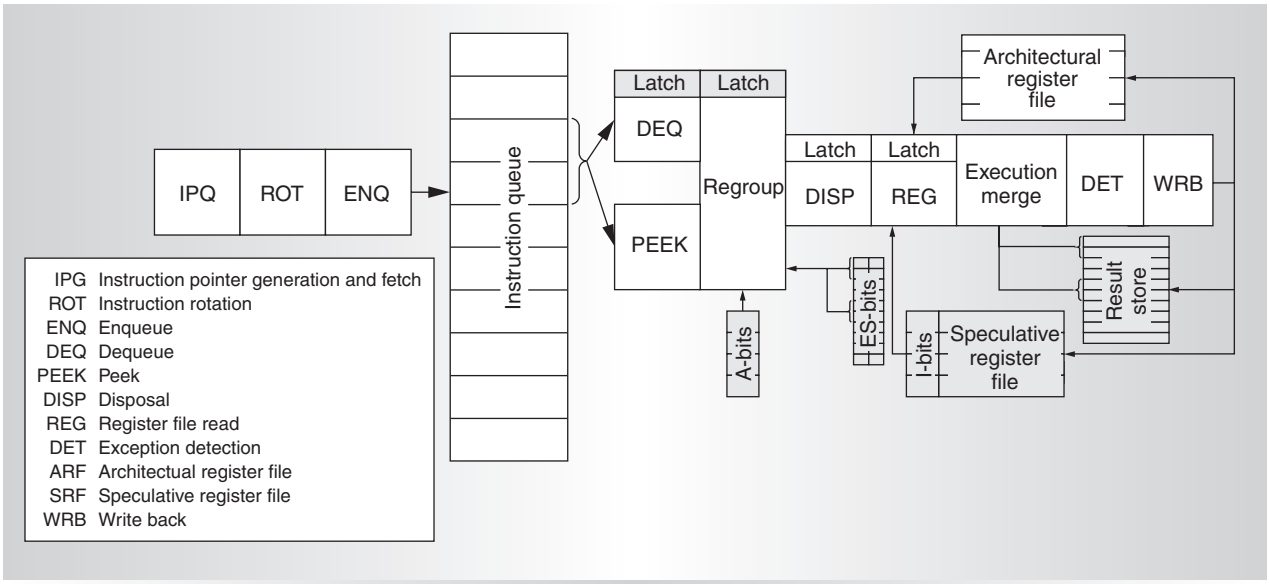

Figure 2. Integer multipass pipeline.

facilitate fast advance execution restart and fast normal execution resumption. The shaded latches preserve the instructions immediately following the instruction that triggered advance execution. The instruction queue in Figure 2 is an enlarged version of the Itanium 2 queue. With a large instruction queue and the additional latches, the pipeline incurs no start-up latency cycles at the beginning of each pass of advance execution or the resumption of normal execution.

Our initial experiments used the compiler to control advance restart. Advance restart is likely desirable if a deferred instruction will defer most subsequent preexecution. For the results we present, the compiler explicitly inserts restart instructions to check whether critical operands are ready, and, if not, cause advance restart to occur. A hardware mechanism could likely dynamically control advance restart even more effectively.

The second important improvement of multipass pipelining over in-order runahead execution is that during advance execution, multipass pipelining preserves valid execution results, using them to reduce power consumption and speed up execution during subsequent passes. As the pipeline in Figure 2 shows, we added a result store to the base pipeline to store advance instruction results. In the period labeled *rally* after advance restart occurs, the processor doesn't have to execute advance instructions such as D again, because the result queue already contains their correct results.

Preserving advance results also allows the second advance execution pass in Figure 1c reach *E* more rapidly, because the instructions with preserved execution results no longer have dataflow dependence on other instructions. The empty bits (E-bits) and the *regroup* stage in Figure 2 make this accelerated execution possible. The regroup stage checks dependences on an instruction-by-instruction basis (as would a wide-issue in-order processor's dependency check stage[1]). Each E-bit indicates whether its corresponding result queue entry contains a valid execution result. If the instruction's associated E-bit is cleared, then the instruction has previously executed with success. It no longer requires execution in the current pass and can be treated as an instruction that simply moves a literal value into a register file. This process, *instruction regrouping*, achieves a dynamic schedule compaction beyond what was possible at compile time. When the processor returns to normal execution, the processor can again use the preserved results for C and D to speed up the instructions' processing while saving energy.

The complete state of memory contents can't be checkpointed the same way that the register state is preserved. Other work describes mechanisms for maintaining memory state during advance execution and supporting store

to load memory dependences, even in the presence of suppressed memory stores.[5]

Multipass benefits

Multipass pipelining lets in-order processors perform useful processing instead of stalling (potentially for hundreds of cycles) while an instruction waits on a data cache miss. Multipass advance execution first targets the initiation of positionally blocked long-latency data cache misses by overlapping them with the advance-mode-causing miss. It also accelerates in-order execution following the handling of the miss by reusing advance results.

Advance execution restart

As advance execution proceeds, the processor often reaches a point at which it can perform little fruitful forward advance execution because most subsequent instructions depend on cache-missing loads or deferred instructions. At the same time, instructions that have previously been deferred because of an unready operand can now represent an opportunity for productive advance execution. The general wakeup mechanisms of out-of-order execution let such instructions execute as soon as their operands are ready. Multipass pipelining achieves the same benefit by relying on advance execution's systematic restart.

Instruction regrouping

Because of the persistent execution performed during advance mode, much of rally mode execution consists of merely merging precomputed instruction results into the processor state. The multipass pipeline doesn't recompute the results of precomputed instructions, and can consider such instructions to no longer depend on their source operands' original producers. This elimination of input dependences permits instruction regrouping. The *regroup* stage of the multipass pipeline can form new issue groups without changing the compiler-specified instruction order, and can thus schedule formerly dependent instructions in the same cycle.

Performance highlights

We evaluated the performance improvement of adding multipass pipelining to an in-order microarchitecture based roughly on the Itanium 2.[2] We used 12 C language benchmarks from SPEC CPU2000 representing a wide variety of application types (we excluded the remaining C SPECint2000 benchmarks because of compilation issues). We compiled and aggressively optimized each application using the OpenImpact compiler (http://gelato.uiuc.edu). We modeled a processor with a six-instruction issue width; a 16-Kbyte, four-way, one-cycle L1 data cache; a 256-Kbyte, eight-way, five-cycle L2 data cache; a 3-Mbyte, 12-way, 12-cycle L3 data cache; and a 145-cycle main-memory latency.

We chose these moderate cache latencies to reflect a near-term design. Multipass pipelining's cache tolerance benefits increase with longer memory latencies. We examine more forward-looking cache parameters elsewhere.[5]

We modeled multipass pipelining as an addition to this baseline machine with a 256-entry instruction queue and result store. For comparison, we also modeled an aggressive out-of-order implementation with a 128-entry scheduling table and a 256-entry instruction window.

Figure 3 gives a performance breakdown of the multipass concept and a comparison with the aggressive out-of-order design using the speedup over the baseline in-order model. The first component is the performance improvement achieved by Dundas and Mudge's in-order runahead-execution scheme. On average, the prefetching effects of this simple runahead approach achieves a 1.1× speedup. The next component is the additional speedup achieved by a multipass-pipeline processor, which preserves advance execution results and uses those results to perform instruction regrouping. By eliminating many dependences to preexecuted instructions, instruction regrouping's benefit is twice that achieved by runahead execution alone. Adding advance execution restart to multipass pipelining provides a more modest 1.06× additional speedup. This feature is particularly important for mcf, where, by tolerating data cache misses whose accesses aren't ready on the first advance execution pass, it raises the overall speedup by 1.67×. For several other benchmarks, advance restart provided no benefit. This is because these benchmarks had fewer important cache misses depending on other (relatively short) cache misses, and they didn't exhibit the behavior targeted with the

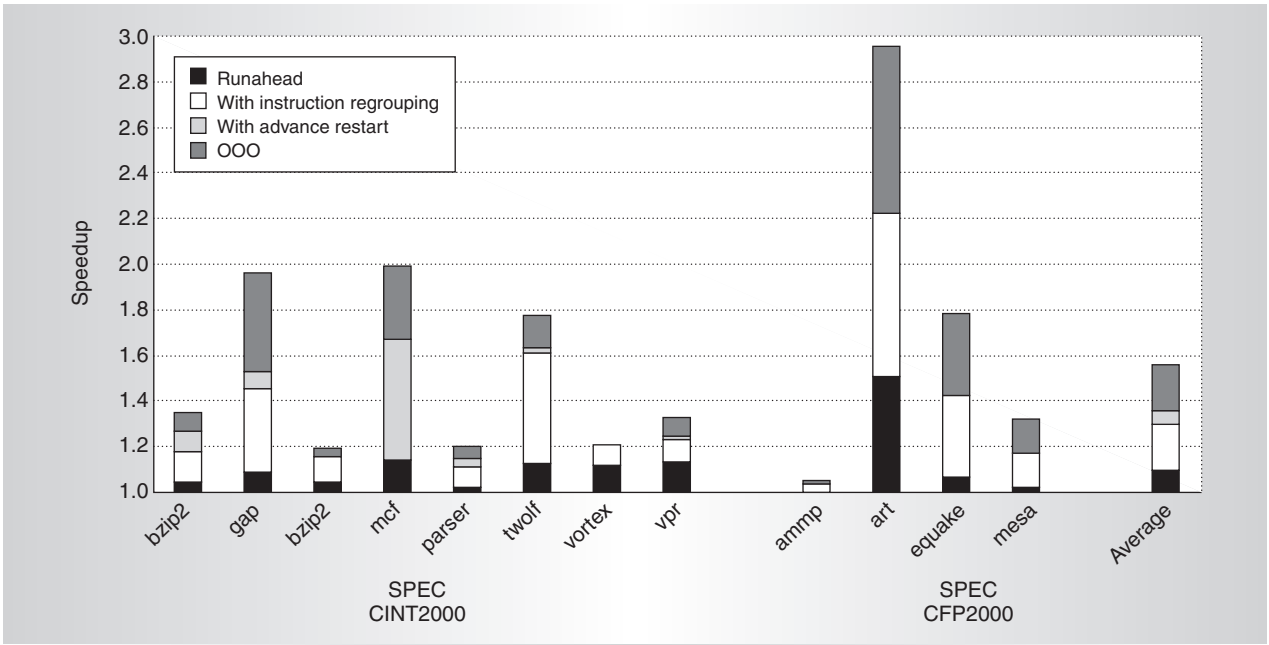

Figure 3. Speedups over in-order for in-order runahead, multipass pipelining with instruction regrouping, multipass pipelining with instruction regrouping and advance restart, and out-of-order execution.

compiler identification of advance restart locations. On average, multipass pipelining achieves a 1.36× speedup.

Overall, the somewhat idealized out-of-order execution that we modeled only achieves an additional 1.14× speedup over a multipass-pipeline system. These benefits come from its ability to find instruction-level parallelism by reordering instruction executions and its more general tolerance of runtime latency. However, the parameters chosen for this model are quite aggressive to match the aggressive 256-instruction multipass implementation. In other experiments, multipass pipelining performed better than moderate implementations of out-of-order execution while using simpler and lower-power structures.[5]

Cost and complexities

Multipass pipelining's appeal over more traditional dynamic memory-latency-tolerant designs (which use out-of-order execution) is the relative power cost of its supporting structures. The additional structures required to implement out-of-order execution create significant power overhead. For example, in the Alpha 21264 processor, support for dynamic scheduling consumes as much at 18 percent of total power—almost as much as all of its integer and floating-point execution units combined.[6]

We used power models adapted from the Wattch framework[7] to compare the power consumption of associated structures in the out-of-order and multipass models used in our simulations, utilizing technology parameters similar to those of contemporary high-performance microprocessors. Table 1 shows the study's results as power ratios for three categories of corresponding structures:

- preservation of computed results,
- instruction issue, and
- load/store ordering.

A ratio greater than 1 indicates higher out-of-order power. The peak power ratio assumes maximum switching activity. We based the average power ratio on simulated results and Wattch's linear clock-gating model. To measure the average-power ratio, we incorporated the relevant Wattch component models into the cycle-by-cycle simulator used for performance results.

The Wattch component power models consist primarily of array components: decoders, word lines, bit lines, and sense amplifiers. For these structures, we expect power to scale

Table 1. Power ratios (for circuits in a 100-nm process) for out-of-order to multipass structures.

Function	Out-of-order structures	Multipass structures	Peak-power ratio	Average-power ratio
Result storage	Combined architectural and rename register file, and register alias table	Architectural and speculative register files, result score	0.99*	1.20
Instruction storage	Instruction wakeup and issue	Instruction queue	10.28	7.15
Load/store ordering	Load and store buffers	Speculative memory address queue and advance store cache	3.21	9.79

*The 0.99 power ratio requires double the register file accesses that the architecture can actually incur; it's an intentionally conservative estimate. Under architectural constraints, the peak power would be 1.92.

nearly linearly with the number of ports, with an additional effect due to cell size increase because of the additional word lines and bit lines needed for access. Table 1 only illustrates the degree of disparity between out-of-order and multipass structures; it doesn't represent the power consumption of any physical implementation.

The large disparity in power ratios is primarily due to the inherently in-order nature of multipass execution. Sequential reads and writes to both the instruction queue and the result store enable simple array structures with a small number of ports each with shared bit lines. Array structures supporting out-of-order designs allow general instruction reordering, but require structures with many randomly accessible ports (minimally, enough ports to handle the pipeline's issue width).

Our modeling of array-based dynamic scheduling for out-of-order is a bit more energy efficient than most actual implementations. Often, dynamic scheduling uses content-addressable memories, which read and match their entire contents, requiring far more power than indexed arrays. Finally, multipass structures only consume power during advance and rally modes, which causes the average power consumption ratio to be higher than the peak ratio for results storage and load/store ordering.

The importance of preserving advance execution results for reuse and instruction regrouping invites comparison with other runahead techniques that discard all preexecuted work (other than preinitiated data cache accesses). Previous examination of runahead execution as a way to expand otherwise modest instruction windows in out-of-order execution systems[8] has found little benefit from preserving runahead execution results.[9] The out-of-order runahead approach's use of dynamic scheduling logic and multipass pipelining's avoidance of dynamic scheduling cause these seemingly contradictory results.

Instruction regrouping in multipass systems uses advance execution results to eliminate breaks in the issue of instructions caused by the input dependences of preexecuted instructions. This compacts the still in-order *schedule height*, the optimal number of cycles required to issue instructions while meeting their dependences. In out-of-order processors, dynamic schedules are already more compact than the static instruction arrangement, and reducing input dependences due to result reuse appears to have a smaller additional benefit.

A prominent issue for incorporating multipass pipelining into existing architectures is using a restart mechanism that doesn't require instruction set changes or executable recompilation. To support existing binaries, we could use hardware mechanisms to detect the opportunity for advance restart. Such a mechanism could be triggered on the required condition for useful advance restart, when a load other than advance-mode initiator returns its value after a miss. We're currently experimenting with configurations to determine effective methods of balancing restart opportunities with further advance execution, and preliminary investigation has shown cases in which microarchitectural techniques achieve greater speedups from advance restart than the compiler technique used here.

We present multipass pipelining largely in

the context of an explicitly parallel architecture. However, increasingly restrictive power budgets will require processor families that are currently implemented with traditional out-of-order microarchitectures to decrease their reliance on expensive dynamic structures. MICRO

Acknowledgments

We acknowledge the past and present members of the Illinois Microarchitecture Project utilizing Advanced Compiler Technology (IMPACT) research group for their feedback and assistance. The Gigascale Systems Research Center, US National Science Foundation ITR Grant 86096, and generous gift funds from Intel Corp. partially supported this work.

References

1. R. Heald et al., "A Third-Generation SPARC V9 64-b Microprocessor," *IEEE J. Solid-State Circuits*, vol. 35, no. 11, Nov. 2000, pp. 1526-1538.
2. C. McNairy and D. Soltis, "Itanium 2 Processor Microarchitecture," *IEEE Micro*, vol. 23, no. 2, Mar. 2003, pp. 44-55.
3. J.W. Sias et al., "Field-Testing Impact EPIC Research Results in Itanium 2," *Proc. 31th Ann. Int'l Symp. Computer Architecture* (ISCA 04), IEEE CS Press, 2004, pp. 26-37.
4. J. Dundas and T. Mudge, "Improving Data Cache Performance by Preexecuting Instructions Under a Cache Miss," *Proc. 11th Ann. Int'l Conf. Supercomputing* (SC 97), IEEE CS Press, 1997, pp. 66-75.
5. R.D. Barnes, S. Ryoo, and W.W. Hwu, "Flea-Flicker Multipass Pipelining: An Alternative to the High-Power Out-of-Order Offense," *Proc. 38th Ann. Int'l Symp. Microarchitecture* (Micro-38), IEEE CS Press, 2005, pp. 319-330.
6. M.K. Gowan, L.L. Biro, and D.B. Jackson, "Power Considerations in the Design of the Alpha 21264 Microprocessor," *Proc. 35th Design Automation Conf.* (DAC 98), ACM Press, 1998, pp. 726-731.
7. D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A Framework for Architectural-level Power Analysis and Optimizations," *Proc. 27th Ann. Int'l Symp. Computer Architecture* (ISCA 00), ACM Press, 2000, pp. 83-94.
8. O. Mutlu et al., "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors," *Proc. 9th Int'l Symp. High-Performance Computer Architecture* (HPCA 03), IEEE CS Press, 2003, pp. 129-140.
9. O. Mutlu et al., "On Reusing the Results of Preexecuted Instructions in a Runahead Execution Processor," *Computer Architecture Letters*, vol. 4, Jan. 2005; http://www.cs.virginia.edu/~tcca.

Ronald D. Barnes is an assistant professor at George Mason University. His research interests include complexity-effective microarchitectures, program-phase detection, and binary and runtime optimization. Barnes has a PhD in electrical engineering from the University of Illinois, Urbana-Champaign. He is a member of the IEEE and the ACM.

Shane Ryoo is a graduate student in electrical engineering at the University of Illinois, Urbana-Champaign. His research interests include interprocedural analysis techniques for identifying coarse-grained parallelism and efficient algorithms for computer architectures and compilers. Ryoo has an MS in electrical engineering from the University of Illinois, Urbana-Champaign. He is a student member of the IEEE.

Wen-mei W. Hwu is the Sanders-AMD Endowed Chair Professor in the Department of Electrical and Computer Engineering, University of Illinois, Urbana-Champaign. He is the director of the Illinois Microarchitecture Project utilizing Advanced Compiler Technology (Impact) lab, which delivers new compiler and computer architecture technologies to the computer industry. His research interests include architecture, implementation, and software for high-performance computer systems. Hwu has a BSEE from the National Taiwan University and a PhD in computer science from the University of California, Berkeley. He is a Fellow of the IEEE and the ACM.