Runahead Execution: An Effective Alternative to Large Instruction Windows

AN INSTRUCTION WINDOW THAT CAN TOLERATE LATENCIES TO DRAM MEMORY IS PROHIBITIVELY COMPLEX AND POWER HUNGRY. TO AVOID HAVING TO BUILD SUCH LARGE WINDOWS, RUNAHEAD EXECUTION USES OTHERWISE-IDLE CLOCK CYCLES TO ACHIEVE AN AVERAGE 22 PERCENT PERFORMANCE IMPROVEMENT FOR PROCESSORS WITH INSTRUCTION WINDOWS OF CONTEMPORARY SIZES. THIS TECHNIQUE INCURS ONLY A SMALL HARDWARE COST AND DOES NOT SIGNIFICANTLY INCREASE THE PROCESSOR'S COMPLEXITY.

Onur Mutlu
The University of Texas at Austin

Jared Stark
Chris Wilkerson
Intel Microarchitecture Research Lab

Yale N. Patt
The University of Texas at Austin

High-performance processors execute instructions out of program order to tolerate long latencies and extract instruction-level parallelism. However, these processors retire instructions in program order to support precise exceptions.[1] If the execution of a long-latency instruction is not complete, the processor cannot retire that instruction and the instructions following it in the sequential instruction stream. If the window is not large enough, this delay in retirement causes incoming instructions to fill the instruction window. Once the window becomes full, the processor cannot place new instructions into the window, and it stalls. This resulting stall is called a *full-window stall*, and prevents the processor from finding independent instructions to execute to tolerate the long latency. The straightforward solution to this problem is to increase the size of the instruction window. However, doing so is challenging because of design complexity, verification difficulty, and the increased power consumption of a large instruction window.

Unfortunately, main memory latencies are so long that out-of-order processors require large instruction windows to tolerate them. A cache miss to main memory costs about 128 cycles on an Alpha 21264[2] and 330 cycles on a Pentium-4-like processor.[3] Figure 1 shows that a Pentium-4-like processor with a 128-entry instruction window and a 512-Kbyte level-two (L2) cache (processor 1) spends 68 percent of its execution cycles in full-window stalls. If the L2 cache is perfect—that is, processor memory accesses never miss in this cache—the processor (processor 2) wastes only 30 percent of its cycles in full-window stalls, indicating that long-latency L2 misses cause the most full-window stalls in processor 1. However, processor 3, which has a 2,048-entry instruction window and a 512-Kbyte L2 cache, spends only 33 percent of its cycles in full-window stalls. So a processor

Published by the IEEE Computer Society

with a large instruction window tolerates the main-memory latency much better than a processor with a small instruction window.

We propose a simple alternative to large, complex instruction windows. This alternative, *runahead execution*,[4,5] uses a simple algorithm that increases the tolerance of a processor to long-latency memory operations and provides instruction and data prefetching benefits. This algorithm is easily implementable in current out-of-order processors and increases the instructions per cycle (IPC) performance of an aggressive processor model by 22 percent. Compared to large instruction windows, a processor with runahead execution performs 3 percent better than a processor with twice the instruction window size and almost as well as a processor with three times the instruction window size.

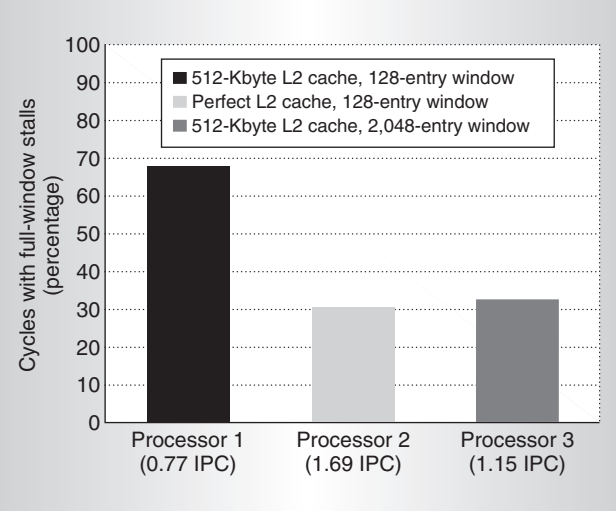

Figure 1. Percentage of execution cycles with full-window stalls. The figure also lists instructions per cycle (IPC). Later, we describe the processor models and benchmarks.

Runahead execution operation

The mechanism we propose avoids stalling the processor when a long-latency L2 cache miss blocks the placement of new instructions into the instruction window. When the processor detects that the oldest instruction being serviced is a long-latency cache miss, it checkpoints the architectural register state, the branch history register, and the return address stack; records the program counter of the long-latency instruction; and enters a speculative processing mode called *runahead mode*. The processor then removes this long-latency instruction from the instruction window.

While in runahead mode, the processor continues to execute instructions without updating the architectural state and without blocking retirement due to long-latency cache misses and the instructions dependent on them. The processor identifies the results of long-latency cache misses and their dependents as bogus, removing instructions that generate or source bogus results from the instruction window so that they do not prevent the placement of independent instructions into the window. Runahead mode allows the processor to execute more instructions than the instruction window normally permits.

Some of the instructions in runahead mode—those that are independent of long-latency cache misses—miss in the instruction, data, or unified caches. Their miss latencies overlap with the latency of the runahead-causing cache miss. When the runahead-causing cache miss completes, the processor exits runahead mode, restores the checkpointed state, and resumes normal instruction fetch and execution starting with the runahead-causing instruction. Once the processor returns to normal mode, it can make faster progress without stalling because, during runahead mode, the processor has already prefetched some of the data and instructions for normal mode into the caches.

Benefits

Runahead execution lets the processor do useful processing instead of stalling for hundreds of cycles while it services a long-latency data cache miss. The processing during runahead execution targets the discovery and initiation of long-latency data and instruction accesses to DRAM and services them in parallel with the runahead-causing miss. Besides prefetching these long-latency accesses, runahead execution prefetches data and instructions between levels of the cache hierarchy; trains the hardware data and instruction prefetchers with future access information; and trains the branch prediction structures.

Cost and complexity

In previous work, we detail the runahead execution mechanism and its implementation on a high-performance processor.[5] We also

described the additional hardware necessary to implement runahead execution. This hardware consists of the checkpointed architectural registers, branch history register, and return address stack; a single invalid (INV) bit associated with every physical register and store buffer entry; and a small, 512-byte store buffer (runahead cache) used to forward data from stores to loads during runahead mode.[5] None of these added structures are complex or on the processor's critical path.

The mechanism for checkpointing the architectural registers depends on the microarchitecture. Microarchitectures that store the architectural state in the physical register file can avoid checkpointing the entire architectural register state by checkpointing only the register map that points to the architectural state. It is possible to checkpoint the return address stack without significant hardware cost.[6]

The INV bit identifies the entry associated with it as bogus, that is, dependent on a long-latency cache miss. The mechanism that communicates INV bits between dependent instructions is already present in an out-of-order processor, which communicates data values between dependent instructions.

The runahead store buffer is perhaps the most significant area cost of runahead execution. However, it is small compared to the level-one (L1) data cache. Our simulations show that this buffer is very latency-tolerant and that the processor does not need to access it in parallel with the data cache. So this buffer is not on the processor's critical path.

Adding runahead execution to an out-of-order processor does not significantly increase processor complexity. However, as we will show in a later section, a processor with runahead execution attains the performance of those with larger instruction windows, which are power hungry, complex, and on the critical path.[7] So runahead execution offers a cost- and complexity-effective alternative to large windows.

Performance

We evaluated the performance improvement of adding runahead execution to an aggressive Pentium-4-like processor.[8] We use an execution-driven x86 simulator and 80 memory-intensive benchmarks from a variety of suites: SPEC (SPEC95, FP00, Int00), Internet (Web), multimedia (MM), productivity (Prod), server (Server), and workstation (WS). In previous work, we describe our simulation methodology and benchmark sets.[5] The processor we model is three micro-ops wide and has a 128-entry instruction window (in terms of micro-ops); 29-stage pipeline; 32-Kbyte, eight-way, three-cycle L1 data cache; 512-Kbyte, eight-way, 16-cycle L2 unified cache; and a 12K-micro-op, eight-way trace cache. Main memory latency is 495 cycles. The processor uses an aggressive streaming hardware data prefetcher[8] and a streaming instruction prefetcher. We modeled bandwidth and contention at all levels of the memory hierarchy. Our earlier work described other parameters of the baseline processor.[5] All IPC numbers are in terms of micro-ops per cycle.

Runahead execution versus large windows

Figure 2 shows the IPCs of five different processors for each benchmark suite. From left, the first bar shows the IPC of the baseline processor. The next bar shows the IPC of the baseline with runahead execution. The other three bars show the IPCs of processors without runahead; instead, these processors had larger instruction windows with 256, 384, and 512 entries. Where applicable, the percentages show the IPC improvement of adding runahead execution to the baseline.

On average (shown in the rightmost set of bars), adding runahead execution to the baseline processor improves IPC by 22 percent. The baseline processor with runahead execution outperforms the processor with a 256-entry window by 3 percent. Also, the baseline processor with runahead execution has an IPC within 1 percent of that of the processor with a 384-entry window. So, runahead execution on a 128-entry window processor attains almost the same IPC as a processor with three times the window size.

Runahead execution on future processors

Based on experiments that we present in another paper,[5] we believe that runahead execution will become more important and effective in future-generation microprocessors. As processor and system designers continue to push for shorter cycle times and larger memory modules, and memory designers contin-

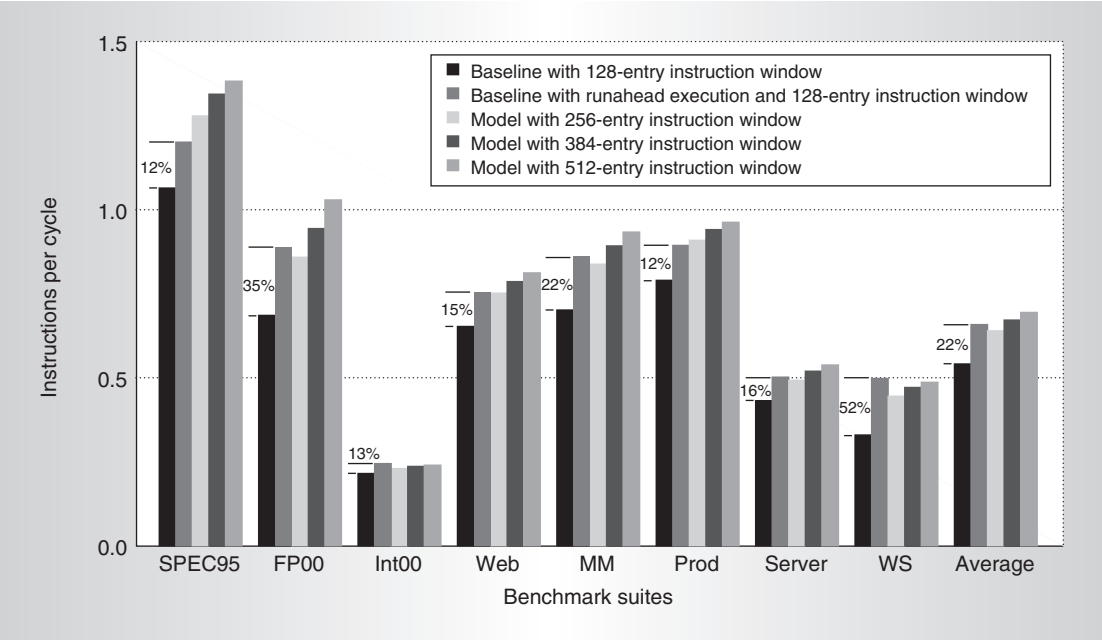

Figure 2. Performance of runahead versus large windows.

ue to push for higher bandwidth and density, main-memory latencies will continue to increase in terms of processor cycles.[2] These increased latencies should make runahead execution more effective for future processors.

To support this hypothesis, we examined the performance of runahead execution on a future processor model,[5] which is six micro-ops wide with a 58-stage pipeline, 512-entry window, 1-Mbyte L2 cache, and 1,008-cycle main-memory latency. Runahead execution on this future model is just as effective and improves IPC by 23 percent on average.

The effectiveness of the processor's instruction supply mechanism (branch prediction and instruction fetch unit) bounds the performance improvement of runahead execution, especially on the wider, deeper, and larger future processor model. Our previous paper described how a processor with a better instruction supply mechanism benefits more from runahead execution. As architects continue to improve branch prediction and instruction fetch units, the performance improvement provided by runahead execution will increase.

Future-generation processors will also have larger L2 caches. Figure 3 shows that implementing runahead execution on a processor with a 1-Mbyte L2 cache improves the IPC by 17 percent. With a 4-Mbyte L2 cache, IPC improves by 16 percent. So, runahead execution remains effective for large L2 caches. For the Int00 suite, runahead execution becomes more effective as L2 cache size increases, because a larger L2 cache is more tolerant to the pollution generated by inaccurate prefetches in runahead mode.

Runahead execution benefit analysis

Runahead execution provides performance improvements in two main areas: instruction and data prefetching. The instruction-prefetching improvement comes from prefetching runahead instructions into the L2 cache and the trace (or instruction) cache, and training the branch predictors during runahead mode. Data-prefetching improvement comes from prefetching runahead load requests into the L2 cache and L1 data cache, and training the hardware data prefetchers' buffers during runahead mode.

We find that, on average, 88 percent of the IPC improvement comes from data prefetching. All benchmark suites, except the server suite, owe more than 70 percent of the performance improvement to data prefetching. Because server applications are branch intensive and have large instruction footprints, 45 percent of the performance improvement for

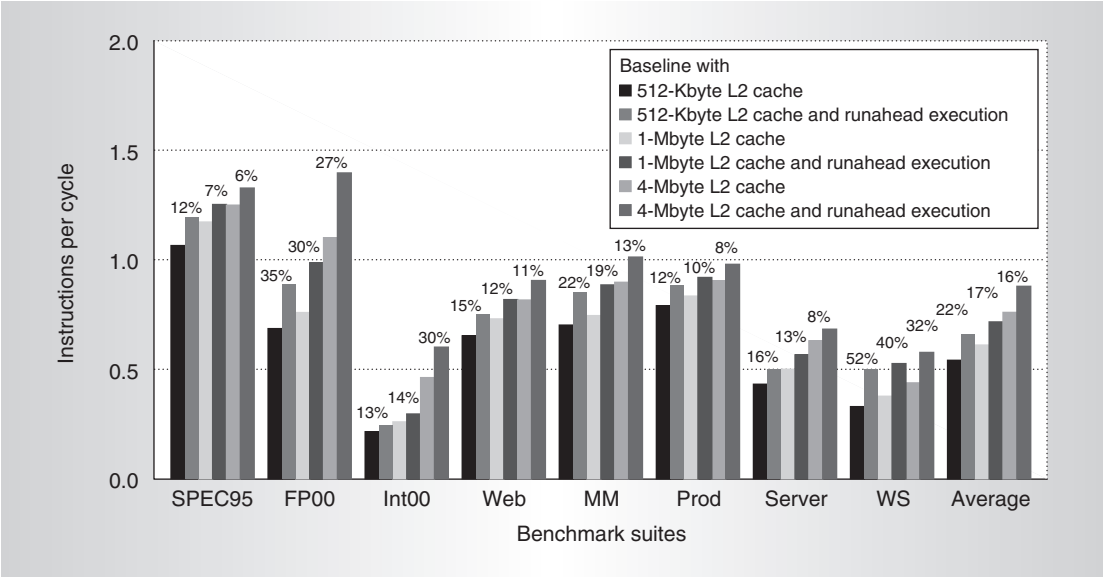

Figure 3. IPC improvement of runahead execution for 512-Kbyte, 1-Mbyte, and 4-Mbyte L2 caches.

the server suite comes from instruction prefetching.

Runahead execution improves performance because the processing during runahead mode fully or partially eliminates the cache misses incurred during normal mode. On average, the baseline processor incurs 13.7 L1 data cache misses and 4.3 L2 data misses per 1,000 instructions. If we add runahead execution to the baseline processor, the L1 data cache miss rate during normal mode decreases by 18 percent. With runahead execution, 15 percent of the baseline processor's L2 data misses do not occur during normal mode. Another 18 percent of the L2 data misses in the baseline processor begin during runahead mode but are not fully complete by the time instructions in normal mode need them. We find that the performance improvement of runahead execution correlates well with the reduction in normal-mode L2 data misses, indicating that the main benefit of runahead execution comes from prefetching data from main memory to the L2 cache.

Overall, the L2 data prefetch accuracy of runahead execution is 94 percent, and its L2 instruction prefetch accuracy is 98 percent. This shows that runahead execution is an accurate prefetching technique, as we expected, because it follows the path that the instruction stream will follow in the future.

Runahead execution has two immediate advantages: It permits small-instruction-window processors to attain the same performance as processors with much larger instruction windows, and it does so with a simple, cost-effective implementation that enables seamless integration into today's high-performance processors. It provides these advantages using a simple algorithm that increases the tolerance of an out-of-order processor to long-latency memory operations. As these long latencies increase in future processor generations, runahead execution could become more important and effective. As architects continue to improve the branch prediction algorithms and instruction fetch units, the effectiveness of runahead execution will continue to increase. MICRO

References

1. J.E. Smith and A.R. Pleszkun, "Implementation of Precise Interrupts in Pipelined Processors," *Proc. 12th Ann. Int'l Symp. Computer Architecture* (ISCA 85), ACM Press, 1985, pp. 36-44.
2. M.V. Wilkes, "The Memory Gap and the Future of High-Performance Memories," *ACM Computer Architecture News*, vol. 29, no. 1, Mar. 2001, pp. 2-7.
3. E. Sprangle and D. Carmean, "Increasing Processor Performance by Implementing

Deeper Pipelines," *Proc. 29th Ann. Int'l Symp. Computer Architecture* (ISCA 02), IEEE CS Press, 2002, pp. 25-34.
4. J. Dundas and T. Mudge, "Improving Data Cache Performance by Pre-Executing Instructions Under a Cache Miss," *Proc. 1997 Int'l Conf. Supercomputing* (ICS 97), ACM Press, 1997, pp. 68-75.
5. O. Mutlu et al., "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors," *Proc. 9th IEEE Int'l Symp. High-Performance Computer Architecture* (HPCA 03), IEEE CS Press, 2003, pp. 129-140.
6. S. Jourdan et al., "The Effects of Mispredicted-Path Execution on Branch Prediction Structures," *Proc. 1996 ACM/IEEE Conf. Parallel Architectures and Compilation Techniques* (PACT 96), IEEE Press, 1996, pp. 58-67.
7. S. Palacharla et al., "Complexity Effective Superscalar Processors," *Proc. 24th Ann. Int'l Symp. Computer Architecture* (ISCA 97), ACM Press, 1997, pp. 206-218.
8. G. Hinton et al., "The Microarchitecture of the Pentium 4 Processor," *Intel Technology J.*, Feb. 2001.

Onur Mutlu is a doctoral student at The University of Texas at Austin. His research interests include high-performance processor microarchitecture, with a focus on multithreading, data prefetching, and memory subsystem design. Mutlu has a BSE in computer engineering and a BS in psychology from the University of Michigan, and an MS in computer engineering from the University of Texas at Austin.

Jared Stark is a research scientist at Intel's Hillsboro Microarchitecture Research Lab. His research interests include branch prediction, dynamic instruction scheduling, and aggressive speculation. Stark has a BS in electrical engineering and an MS and a PhD in computer engineering, all from the University of Michigan. He is a member of the IEEE.

Chris Wilkerson is a research scientist at Intel's Hillsboro Microarchitecture Research Lab. His research interests include low-power, high-performance, highly scalable processor designs; and high-performance processor designs that target specific application classes such as graphics, media, database, and managed runtime systems. Wilkerson has an MS in electrical and computer engineering from Carnegie Mellon University. He is a member of the AAA.

Yale N. Patt is the Ernest Cockrell Jr. Centennial Chair in Engineering at The University of Texas at Austin, where he directs the research of 13 PhD students on problems in high-performance microarchitecture. Patt has a BS from Northeastern University, and an MS and a PhD from Stanford University, all in electrical engineering. He is the coauthor, with Sanjay Patel, of *Introduction to Computer Systems: From Bits and Gates to C and Beyond,* 2nd ed. (McGraw-Hill, 2004). An IEEE and an ACM Fellow, Patt is a recipient of the IEEE/ACM Eckert-Mauchly Award (1996), the IEEE Piore Medal (1995), and the ACM Karl V. Karlstrom Outstanding Educator Award (2000).

Direct questions and comments about this article to Onur Mutlu, The University of Texas at Austin, Electrical and Computer Engineering, 1 University Station Stop C0803, Austin, TX 78712; onur@ece.utexas.edu.