Efficient Runahead Execution: Power-Efficient Memory Latency Tolerance

SEVERAL SIMPLE TECHNIQUES CAN MAKE RUNAHEAD EXECUTION MORE EFFICIENT BY REDUCING THE NUMBER OF INSTRUCTIONS EXECUTED AND THEREBY REDUCING THE ADDITIONAL ENERGY CONSUMPTION TYPICALLY ASSOCIATED WITH RUNAHEAD EXECUTION.

Onur Mutlu
Hyesoon Kim
Yale N. Patt
University of Texas at Austin

Today's high-performance processors face main-memory latencies on the order of hundreds of processor clock cycles. As a result, even the most aggressive processors spend a significant portion of their execution time stalling and waiting for main-memory accesses to return data to the execution core. Previous research has shown that runahead execution significantly increases a high-performance processor's ability to tolerate long main-memory latencies.[1, 2] Runahead execution improves a processor's performance by speculatively preexecuting the application program while the processor services a long-latency (L2) data cache miss, instead of stalling the processor for the duration of the L2 miss. Thus, runahead execution lets a processor execute instructions that it otherwise couldn't execute under an L2 cache miss. These preexecuted instructions generate prefetches that the application program will use later, improving performance.

Runahead execution is a promising way to tolerate long main-memory latencies because it has modest hardware cost and doesn't significantly increase processor complexity.[3] However, runahead execution significantly increases a processor's dynamic energy consumption by increasing the number of speculatively processed (executed) instructions, sometimes without enhancing performance.

For runahead execution to be efficiently implemented in current or future high-performance processors which will be energy-constrained, processor designers must develop techniques to reduce these extra instructions. Our solution to this problem includes both hardware and software mechanisms that are simple, implementable, and effective.

Background on runahead execution

Conventional out-of-order execution processors use instruction windows to buffer instructions so they can tolerate long latencies. Because a cache miss to main memory takes hundreds of processor cycles to service, a processor needs to buffer an unreasonably large number of instructions to tolerate such a long latency. Runahead execution[1] provides the memory-level parallelism (MLP) benefits of a large instruction window without requiring the large, complex, slow, and power-hungry structures—such as large schedulers, register files, load/store buffers, and reorder buffers—associated with a large instruction window.

The execution timelines in Figure 1 illustrate the differences between the operation of a con-

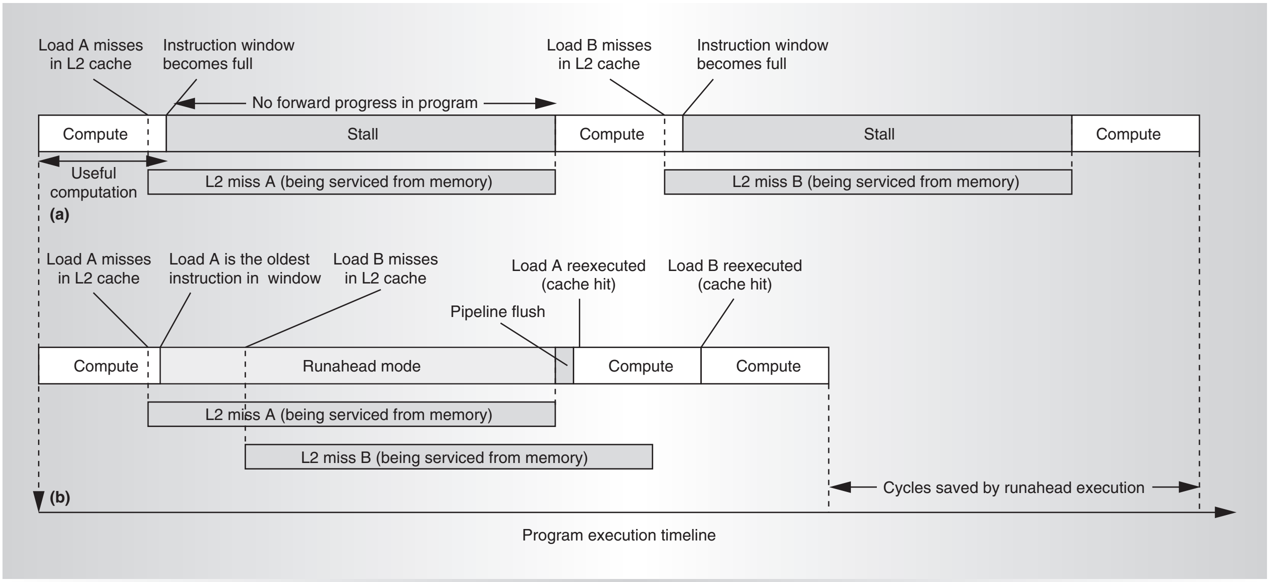
Load A misses in L2 cache
Instruction window becomes full
No forward progress in program
Load B misses in L2 cache
Instruction window becomes full
Compute
Stall
Compute
Stall
Compute
Useful computation
L2 miss A (being serviced from memory)
L2 miss B (being serviced from memory)
(a)
Load A misses in L2 cache
Load A is the oldest instruction in window
Load B misses in L2 cache
Load A reexecuted (cache hit)
Load B reexecuted (cache hit)
Pipeline flush
Compute
Runahead mode
Compute
Compute
L2 miss A (being serviced from memory)
L2 miss B (being serviced from memory)
Cycles saved by runahead execution
(b)
Program execution timeline

Figure 1. Execution timelines showing a high-level overview of the concept of runahead execution: conventional out-of-order execution processor (a) and runahead execution processor (b).

ventional out-of-order execution processor (Figure 1a) and a runahead execution processor (Figure 1b). A conventional processor's instruction window becomes full soon after a load instruction incurs an L2 cache miss. Once the instruction window is full, the processor can't decode and process any new instructions and stalls until it has serviced the L2 cache miss.

While the processor is stalled, it makes no forward progress on the running application. Therefore, a memory-intensive application's execution timeline on a conventional processor consists of useful compute periods interleaved with long useless stall periods due to L2 cache misses, as Figure 1a shows. With increasing memory latencies, stall periods start dominating the compute periods, leaving the processor idle for most of its execution time and thus reducing performance.

Runahead execution avoids stalling the processor when an L2 cache miss occurs, as Figure 1b shows. When the processor detects that the oldest instruction is waiting for an L2 cache miss that is still being serviced, it checkpoints the architectural register state, the branch history register, and the return address stack, and enters a speculative processing mode—the runahead mode. The processor then removes this L2-miss instruction from the instruction window. While in runahead mode, the processor continues to execute instructions without updating the architectural state. It identifies the results of L2 cache misses and their dependents as bogus or invalid (INV) and removes instructions that source INV results (INV instructions) from the instruction window so they don't prevent the processor from placing independent instructions into the window. Pseudoretirement is the program-order removal of instructions from the processor during runahead mode. Some of the instructions executed in runahead mode that are independent of L2 cache misses might miss in the instruction, data, or unified caches (for example, Load B in Figure 1b). The memory system overlaps their miss latencies with the latency of the runahead-causing cache miss. When the runahead-causing cache miss completes, the processor exits runahead mode by flushing the instructions in its pipeline. It restores the checkpointed state and resumes normal instruction fetch and execution starting with the runahead-causing instruction (Load A in Figure 1b).

When the processor returns to normal mode, it can make faster progress without stalling because it has already prefetched into the caches during runahead mode some of the data and instructions needed during normal mode. For example, in Figure 1b, the processor doesn't need to stall for Load B because it discovered the L2 miss caused by Load B in runahead mode and serviced it in

Related work on runahead execution

As a promising technique for increasing tolerance to main-memory latency, runahead execution has recently inspired and attracted research from many other computer architects in both industry[1-3] and academia.[4-6] For example, architects at Sun Microsystems are implementing a version of runahead execution in their next-generation microprocessor.[3] To our knowledge, none of the previous work addressed the runahead execution efficiency problem. We hereby provide a brief overview of related work on runahead execution.

Dundas and Mudge first proposed runahead execution as a means to improve the performance of an in-order scalar processor.[7] In other work (see the main article), we proposed runahead execution to increase the main-memory latency tolerance of more aggressive out-of-order superscalar processors. Chou and colleagues demonstrated that runahead execution effectively improves memory-level parallelism in large-scale database benchmarks because it prevents the instruction and scheduling windows, along with serializing instructions, from being performance bottlenecks.[1] Three recent articles[1,4,6] combined runahead execution with value prediction, and Zhou[5] proposed using an idle processor core to perform runahead execution in a chip multiprocessor. Applying the efficiency mechanisms we propose to these variants of runahead execution can improve their power efficiency.

References

1. Y. Chou, B. Fahs, and S. Abraham, "Microarchitecture Optimizations for Exploiting Memory-Level Parallelism," *Proc. 31st Int'l Symp. Computer Architecture* (ISCA 04), IEEE CS Press, 2004, pp. 76-87.
2. S. Iacobovici et al., "Effective Stream-Based and Execution-Based Data Prefetching," *Proc. 18th Int'l Conf. Supercomputing*, ACM Press, 2004, pp. 1-11.
3. S. Chaudhry et al., "High-Performance Throughput Computing," *IEEE Micro*, vol. 25, no. 3, May/June 2005, pp. 32-45.
4. L. Ceze et al., "CAVA: Hiding L2 Misses with Checkpoint-Assisted Value Prediction," *Computer Architecture Letters*, vol. 3, Dec. 2004, http://www.cs.virginia.edu/~tcca/2004/ceze_dec04.pdf.
5. H. Zhou, "Dual-Core Execution: Building a Highly Scalable Single-Thread Instruction Window," *Proc. 14th Int'l Conf. Parallel Architectures and Compilation Techniques* (PACT 05), IEEE CS Press, 2005, pp. 231-242.
6. N. Kirman et al., "Checkpointed Early Load Retirement," *Proc. 11th Int'l Symp. High-Performance Computer Architecture* (HPCA-11), IEEE CS Press, 2005, pp. 16-27.
7. J. Dundas and T. Mudge, "Improving Data Cache Performance by Pre-executing Instructions Under a Cache Miss," *Proc. 1997 Int'l Conf. Supercomputing*, IEEE Press, 1997, pp. 68-75.

parallel with the L2 miss caused by Load A. Hence, runahead execution uses otherwise-idle clock cycles caused by L2 misses to pre-execute instructions in the program to generate accurate prefetch requests. Previous research has shown that runahead execution increases processor performance mainly because it parallelizes independent L2 cache misses[3] (see also the "Related Work" sidebar). Furthermore, the memory latency tolerance provided by runahead execution comes at a small hardware cost, as we've shown in previous work.[1,3]

Efficiency of runahead execution

A runahead processor executes some instructions in the instruction stream more than once because it speculatively executes instructions in runahead mode. Because each executed instruction consumes dynamic energy, a runahead processor consumes more dynamic energy than a conventional processor. Reducing the number of instructions executed in runahead mode reduces the energy consumed by a runahead processor. Unfortunately, reducing the number of instructions can significantly reduce runahead execution's performance improvement because runahead execution relies on the execution of instructions in runahead mode to discover L2 cache misses further down in the instruction stream. Our goal is to increase a runahead processor's efficiency without significantly decreasing its instructions per cycle (IPC) performance improvement.

We define efficiency as

Efficiency = Percent increase in IPC performance/Percent increase in executed instructions

where percent increase in IPC performance is the IPC increase after adding runahead execution to a conventional baseline processor, and percent increase in executed instructions is the increase in the number of executed instructions after adding runahead execution.

We can increase a runahead processor's efficiency in two ways:

- We can reduce the number of executed instructions (the denominator) without affecting the increase in IPC (the numerator) by eliminating the causes of inefficiency.
- We can increase the IPC improvement without increasing the number of executed instructions. To do this, we increase the usefulness of each runahead execution period by extracting more useful prefetches from the executed instructions.

Our techniques increase efficiency in both ways.

Figure 2 shows by how much runahead execution increases IPC and the number of exe-

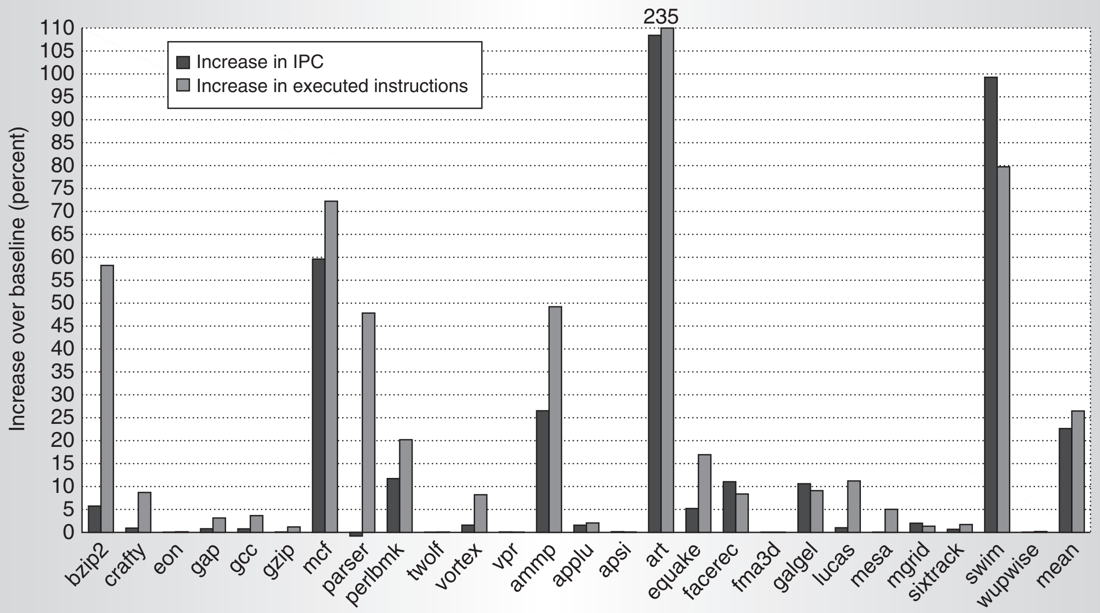

Figure 2. Increase in instructions per cycle (IPC) performance and executed instructions due to runahead execution.

cuted instructions compared to an aggressive conventional out-of-order processor. Our baseline processor model includes an effective stream-based prefetcher, a 1-Mbyte L2 cache, and a detailed model of a main-memory system with a 500-cycle latency. Detailed information on our experimental methodology is available elsewhere.[4]

On average, for the SPEC CPU2000 benchmarks, runahead execution increases IPC by 22.6 percent at a cost of increasing the number of executed instructions by 26.5 percent. Unfortunately, runahead execution in some benchmarks results in a large increase in the number of executed instructions without yielding a correspondingly large IPC improvement. For example, in parser, runahead execution increases the number of executed instructions by 47.8 percent while decreasing the IPC by 0.8 percent. In art, the IPC increase is impressive at 108.4 percent, but is overshadowed by a 235.4 percent increase in the number of executed instructions.

Eliminating the causes of inefficiency

The three major causes of inefficiency in runahead execution processors are short, overlapping, and useless runahead periods. Runahead execution episodes with these properties rarely provide performance benefit but result in unnecessary speculative instruction execution.

Because exiting from runahead mode has a performance cost (it requires a full pipeline flush), such runahead periods can actually decrease performance. We propose some simple techniques to eliminate such periods.

Short runahead periods

In a short runahead period, the processor stays in runahead mode for tens of cycles instead of hundreds. A short runahead period occurs because the processor can enter runahead mode in response to an already outstanding L2 cache miss that was initiated—but not yet completed—by the hardware or software prefetcher, a wrong-path instruction, or a previous runahead period.

Figure 3a shows a short runahead period caused by an incomplete prefetch generated by a previous runahead period. Load B generates an L2 miss when it is speculatively executed in runahead period A. When the processor executes Load B again in normal mode, the associated L2 miss (L2 miss B) is still in progress. Therefore, Load B causes the processor to enter runahead mode again. Shortly afterward, the memory system completely services L2 miss B, and the processor exits runahead mode. Hence, the runahead period caused by Load B is short. Short

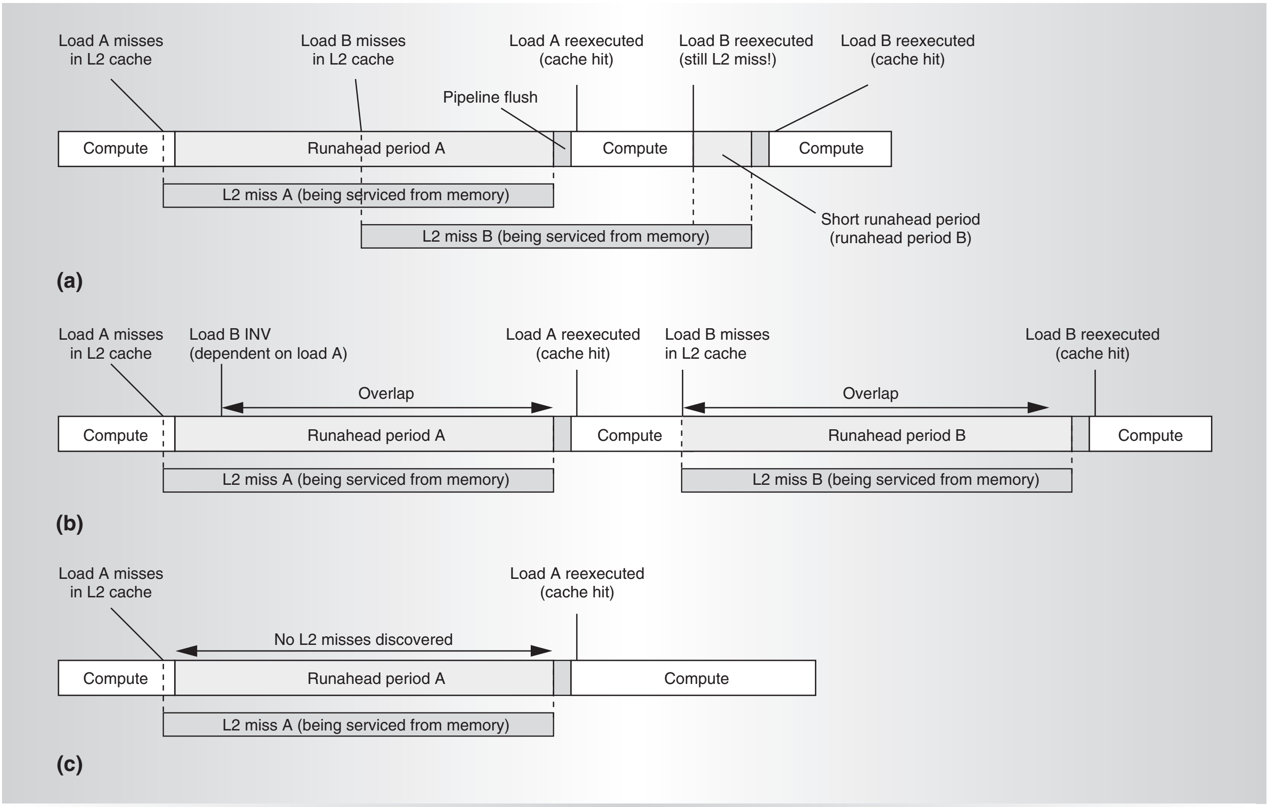
Load A misses in L2 cache
Load B misses in L2 cache
Load A reexecuted (cache hit)
Load B reexecuted (still L2 miss!)
Load B reexecuted (cache hit)
Pipeline flush
Compute
Runahead period A
Compute
Compute
L2 miss A (being serviced from memory)
L2 miss B (being serviced from memory)
Short runahead period (runahead period B)
(a)
Load A misses in L2 cache
Load B INV (dependent on load A)
Load A reexecuted (cache hit)
Load B misses in L2 cache
Load B reexecuted (cache hit)
Overlap
Overlap
Compute
Runahead period A
Compute
Runahead period B
Compute
L2 miss A (being serviced from memory)
L2 miss B (being serviced from memory)
(b)
Load A misses in L2 cache
Load A reexecuted (cache hit)
No L2 misses discovered
Compute
Runahead period A
Compute
L2 miss A (being serviced from memory)
(c)

Figure 3. Example execution timelines illustrating the causes of inefficiency in runahead execution and how they can occur: short runahead period (a), overlapping runahead period (b), and useless runahead period (c).

runahead periods are undesirable because the processor is unlikely to preexecute enough instructions far ahead into the instruction stream and hence unlikely to uncover any useful L2 cache misses during runahead mode.

We eliminate short runahead periods by associating a timer with each outstanding L2 miss. If the L2 miss has been outstanding for more than *N* cycles, where *N* is determined statically or dynamically, the processor predicts that the miss will return from memory soon and doesn't enter runahead mode on that miss. We found that a static threshold of 400 cycles for a processor with a 500-cycle minimum main-memory latency eliminates almost all short runahead periods and reduces the extra instructions from 26.5 to 15.3 percent with negligible impact on performance (Performance improvement decreases slightly from 22.6 to 21.5 percent).

Overlapping runahead periods

Two runahead periods are overlapping if some of the instructions executed in the two periods are the same dynamic instructions. These periods can be caused by independent L2 misses that have significantly different latencies or by dependent L2 misses (for example, L2 misses due to pointer-chasing loads). In Figure 3b, runahead periods A and B are overlapping because of dependent L2 misses. During period A, the processor executes Load B and finds that it is dependent on the miss caused by Load A. Because the processor hasn't serviced L2 miss A yet, Load B can't calculate its address and the processor marks Load B as INV. The processor executes and pseudoretires *N* instructions after Load B and exits period A. In normal mode, the processor reexecutes Load B and finds it to be an L2 miss, which causes runahead period B. The first *N* instructions executed during period B are the same dynamic instructions that were executed at the end of period A. Hence, period B repeats the work done by period A.

Overlapping runahead periods can benefit performance because the completion of Load A can provide data values for more instructions in runahead period B, which can result in the generation of useful L2 misses that the processor couldn't have generated in runahead period A. However, in the benchmark set we examined, overlapping runahead periods rarely benefited performance. In any case, overlapping runahead periods can be a major cause of inefficiency because they result in the execution of the same instructions multiple times in runahead mode, especially if many L2 misses are clustered together in the program.

Our solution to reducing the inefficiency due to overlapping periods involves not entering a runahead period if the processor predicts it to be overlapping with a previous runahead period. During a runahead period, the processor counts the number of pseudoretired instructions. During normal mode, the processor counts the number of instructions fetched since the exit from the last runahead period. If the number of instructions fetched after runahead mode is less than the number of instructions pseudoretired in the previous runahead period, the processor doesn't enter runahead mode. This technique, implemented with two simple counters and a comparator, reduces the extra instructions resulting from runahead execution from 26.5 to 11.8 percent while reducing the performance benefit only slightly, from 22.6 to 21.2 percent.

Useless runahead periods

Useless runahead periods are those in which the processor generates no useful L2 misses that are needed by normal mode execution, as Figure 3c shows. These periods exist because of the lack of MLP[5] in the application program—that is, because the application lacks independent cache misses under the shadow of an L2 miss. Useless periods are inefficient because they increase the number of executed instructions without benefiting performance. To eliminate a useless runahead period, we propose four simple mechanisms for predicting whether a runahead period will be useful (that is, whether it will generate an L2 cache miss).

In the first technique, the processor records the usefulness of past runahead periods caused by static load instructions in the Runahead Cause Status Table (RCST), a small table of two-bit counters.[4] If recent runahead periods initiated by the same load were useful, the processor initiates runahead execution if that load misses in the L2 cache. Otherwise, the processor doesn't enter runahead mode on an L2 miss due to the static load instruction. The insight behind this technique is that the processor can usually predict the usefulness of future runahead periods from the recent past behavior of runahead periods caused by the same static load.

The second technique predicts the available MLP during the ongoing runahead period. If the fraction of INV (that is, L2-miss dependent) load instructions encountered during the ongoing runahead mode is greater than a statically determined threshold, the processor predicts that there isn't enough MLP for runahead execution to exploit and exits runahead mode.

The third technique uses sampling to predict runahead execution's usefulness in a more coarse-grained fashion. This technique aims to turn off runahead execution in program phases with low MLP. To do so, the processor periodically monitors the total number of L2 misses generated during *N* consecutive runahead periods. If this number is less than a static threshold *T*, the processor doesn't enter runahead mode for the next *M* L2 misses. We found that even with untuned values of *N*, *M*, and *T* (100, 1,000, and 25, respectively, in our experiments), sampling can significantly reduce the extra instructions resulting from runahead execution.

The fourth uselessness prediction technique leverages compile-time profiling. The compiler profiles the application and identifies load instructions that consistently cause useless runahead periods. The compiler marks such load instructions as nonrunahead loads. When the hardware encounters a nonrunahead load instruction that is an L2 cache miss, it doesn't initiate runahead execution on that load.

Combining the four uselessness prediction techniques reduces the extra instructions from 26.5 to 14.9 percent while reducing the performance benefit only slightly, from 22.6 to 20.8 percent. Experiments analyzing each technique's effectiveness are available elsewhere.[4]

Increasing the usefulness of runahead periods

Because runahead execution's performance improvement is mainly a result of the useful L2 misses prefetched during runahead mode,[3] discovering more L2 misses during runahead mode can increase the benefit. We propose two optimizations that increase efficiency by increasing runahead periods' usefulness.

Eliminating useless instructions

Because runahead execution aims to generate L2 cache misses, instructions that don't contribute to the generation of L2 cache misses are essentially useless for its purposes. Therefore, eliminating these instructions during runahead mode can increase a runahead period's usefulness.

Floating-point (FP) operate instructions, which don't contribute to the address computation of load instructions, are an example of such useless instructions. We turn off the FP unit during runahead mode and drop FP operate instructions after they are decoded. This optimization spares the processor resources for more useful instructions that lead to the generation of load/store addresses, which increases the likelihood of generating an L2 miss during a runahead period. Furthermore, by not executing the energy-intensive FP instructions and powering down the FP unit during runahead mode, the processor can save significant dynamic and static energy.

However, turning off the FP unit during runahead mode can reduce performance. If a processor mispredicts a control-flow instruction that depends on an FP instruction's result during runahead mode, it has no way of recovering from that misprediction if the FP unit is turned off because the branch's source operand wouldn't be computed. Nevertheless, our simulations show that turning off the FP unit is a valuable optimization that both increases runahead execution's performance improvement (from 22.6 to 24.0 percent) and reduces the extra instructions (from 26.5 to 25.5 percent).

Optimizing runahead execution and hardware prefetcher interaction

A potential benefit of runahead execution is that the processor can update the hardware data prefetcher during runahead mode. If the updates are accurate, the prefetcher can generate prefetches earlier than it would in the baseline processor. This can improve the timeliness of the accurate prefetches. On the other hand, if the prefetches generated by updates during runahead mode are inaccurate, they'll waste memory bandwidth and can cause cache pollution. Moreover, inaccurate hardware prefetcher requests can cause resource contention for the more accurate runahead memory requests during runahead mode and thus reduce runahead execution's effectiveness.

Runahead execution and hardware data prefetching have synergistic behavior[1] (see also Reference 2 in the "Related Work" sidebar). We propose optimizing the prefetcher update policy in runahead mode to increase the synergy between these two prefetching mechanisms.

Our analysis shows that creating new hardware prefetch streams is sometimes harmful in runahead mode because these streams contend with more accurate runahead requests. Thus, not creating prefetch streams in runahead mode increases the usefulness of runahead periods. This optimization increases runahead execution's IPC improvement (from 22.6 to 25.0 percent) and also reduces the extra instructions (from 26.5 to 24.7 percent).

Putting it all together

Figure 4 shows the increase in executed instructions and IPC resulting from runahead execution when we incorporate our proposed techniques into a runahead processor. We examine the effect of profiling-based useless period elimination separately because it requires modifying the instruction set architecture (ISA).

Applying all of our proposed techniques significantly reduces the average increase in executed instructions in a runahead processor from 26.5 to only 6.7 percent (6.2 percent with profiling). Using the proposed techniques reduces the average IPC increase of runahead execution slightly, from 22.6 to 22.0 percent (22.1 percent with profiling). Hence, a runahead processor using the proposed techniques is much more efficient than a traditional runahead processor but it increases performance almost as much.

Figure 5 shows that the proposed techniques are effective for a wide range of memory latencies. As memory latency increases, both the IPC improvement and extra instructions resulting from runahead execution increase. Hence, runahead execution is more effective with longer memory latencies. For almost all memory latencies, using the proposed efficiency techniques increases the average IPC improvement on the FP benchmarks while only slightly reducing the IPC

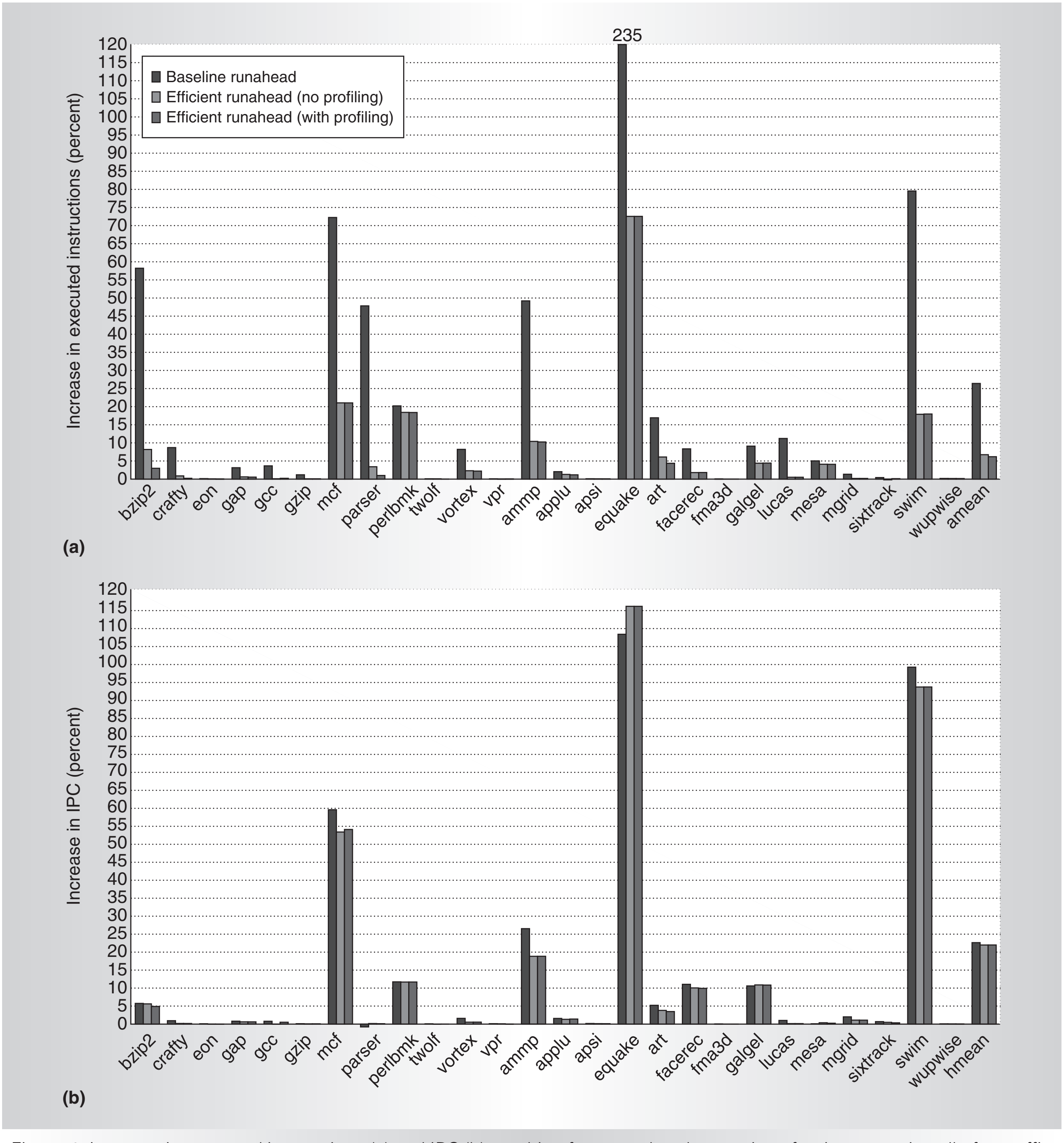

Figure 4. Increase in executed instructions (a) and IPC (b) resulting from runahead execution after incorporating all of our efficiency techniques.

improvement on the integer (INT) benchmarks. For all memory latencies, using the proposed dynamic techniques significantly reduces the extra instructions.

Efficient runahead execution has two major advantages:

- It doesn't require large, complex, and power-hungry structures in the processor core. Instead, it utilizes the already-existing processing structures to improve memory latency tolerance.
- With the simple efficiency techniques described in this article, it requires only

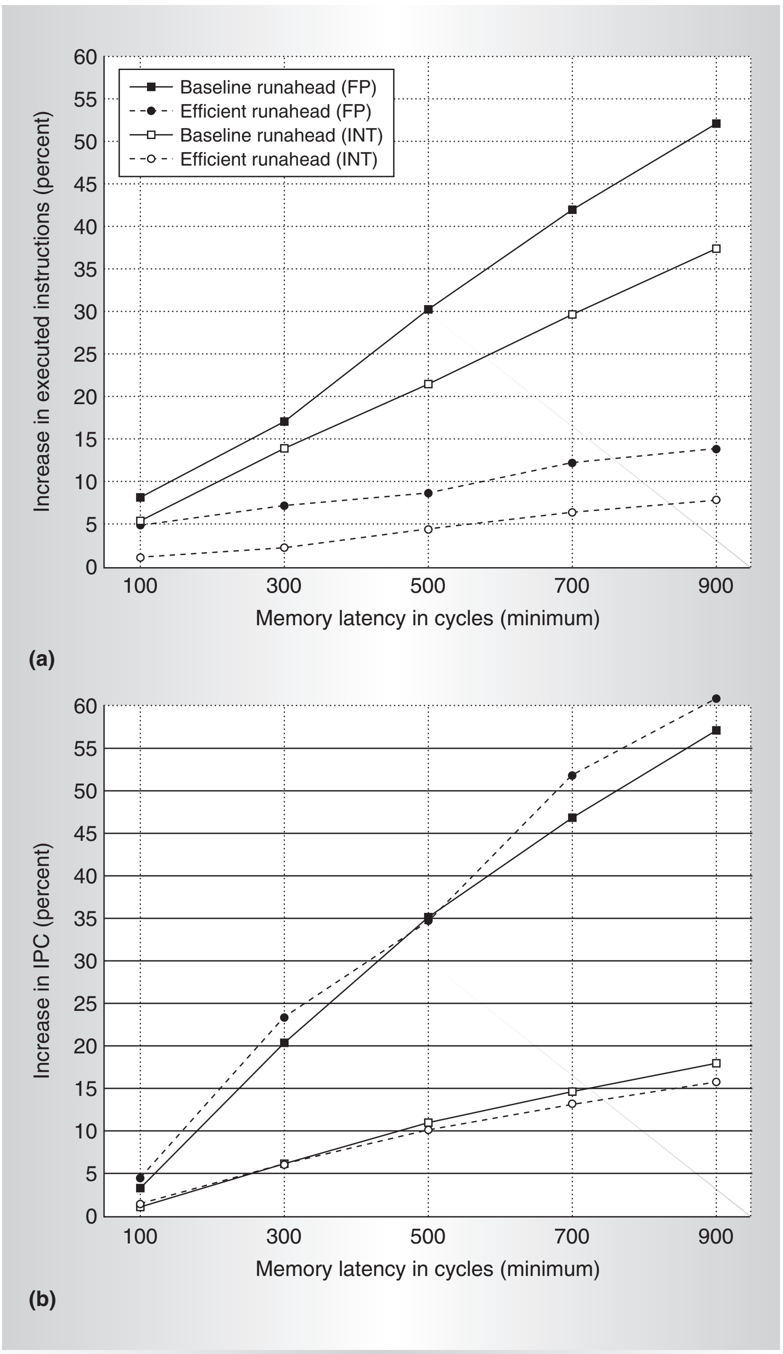

Figure 5. Increase in executed instructions (a) and IPC (b) with and without the efficiency techniques for five different memory latencies. Data shown is averaged separately over integer (INT) and floating-point (FP) benchmarks.

a small number of extra instructions to provide significant performance improvements.

Hence, efficient runahead execution provides a simple, energy-efficient, and complexity-effective solution to the pressing memory latency problem in high-performance processors.

Orthogonal approaches can be developed to solve the inefficiency problem in runahead processors, which we believe is an important research area in runahead execution and other memory-latency tolerance techniques. In particular, solutions to two important problems in computer architecture can significantly increase runahead execution's efficiency: branch mispredictions and dependent cache misses.

Because processors rely on correct branch predictions to stay on the correct program path during runahead mode, the development of more accurate branch predictors will increase runahead execution's efficiency and performance benefits. Irresolvable branch mispredictions that depend on L2 cache misses cause the processor to stay on the wrong path, which might not always provide useful prefetching benefits, until the runahead period ends. Reducing such branch mispredictions with novel techniques is a promising area of future work.

Dependent L2 cache misses reduce a runahead period's usefulness because they can't be parallelized using runahead execution. Therefore, runahead execution is inefficient, and sometimes ineffective, for pointer-chasing workloads in which dependent load instructions are common. In previous work, we've shown that a simple value-prediction technique for pointer-load instructions—address-value delta prediction—significantly increases runahead execution's efficiency and performance by parallelizing dependent L2 cache misses.[6] Enabling the parallelization of dependent cache misses is another promising area of future research in runahead execution.

Our future research will also focus on refining the methods for increasing the usefulness of runahead execution periods. Combined compiler-microarchitecture mechanisms can be instrumental in eliminating useless runahead instructions. Through simple modifications to the ISA, the compiler can convey to the hardware which instructions are important to execute or not execute during runahead mode. Furthermore, the compiler might be able to increase runahead periods' usefulness by trying to arrange code such that independent L2 cache misses are clustered close together during program execution.

Eliminating the reexecution of instructions executed in runahead mode via result reuse[7] or value prediction[8] can potentially increase runahead execution's efficiency. However, even an ideal reuse mechanism doesn't significantly improve performance[7] and likely has significant hardware cost and complexity, which can offset the energy reduction resulting from improved efficiency. Value prediction might not significantly improve efficiency because of its low accuracy.[8] Nevertheless, further research on eliminating the unnecessary reexecution of instructions might yield low-cost mechanisms that can significantly improve runahead efficiency.

Finally, the scope of our efficient processing techniques isn't limited to runahead execution. In general, the proposed runahead uselessness predictors are techniques for predicting the available MLP at a given point in a program. They are therefore applicable to other mechanisms that are designed to exploit MLP. Other methods of preexecution that are targeted for prefetching, such as helper threads,[9,10] can use our techniques to eliminate inefficient threads and useless speculative execution. MICRO

Acknowledgments

We thank Mike Butler, Nhon Quach, Jared Stark, Santhosh Srinath, and other members of the HPS research group for their helpful comments on earlier drafts of this article. We gratefully acknowledge the commitment of the Cockrell Foundation, Intel Corporation, and the Advanced Technology Program of the Texas Higher Education Coordinating Board.

References

1. 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-9), IEEE Press, 2003, pp. 129–140.
2. J. Dundas and T. Mudge, "Improving Data Cache Performance by Pre-executing Instructions Under a Cache Miss," *Proc. 1997 Int'l Conf. Supercomputing*, IEEE Press, 1997, pp. 68-75.
3. O. Mutlu et al., "Runahead Execution: An Effective Alternative to Large Instruction Windows," *IEEE Micro*, vol. 23, no. 6, Nov./Dec. 2003, pp. 20–25.
4. O. Mutlu, H. Kim, and Y.N. Patt, "Techniques for Efficient Processing in Runahead Execution Engines," *Proc. 32nd Int'l Symp. Computer Architecture* (ISCA 05), IEEE CS Press, 2005, pp. 370–381.
5. A. Glew, "MLP Yes! ILP No!" *Architectural Support for Programming Languages and Operating Systems* (ASPLOS 98) Wild and Crazy Idea Session, Oct. 1998; http://www.cs.berkeley.edu/~kubitron/asplos98/slides/andrew_glew.pdf.
6. O. Mutlu, H. Kim, and Y.N. Patt, "Address Value Delta (AVD) Prediction: Increasing the Effectiveness of Runahead Execution by Exploiting Regular Memory Allocation Patterns," *Proc. 38th Int'l Symp. Microarchitecture* (Micro-38), IEEE CS Press, 2005, pp. 233–244.
7. O. Mutlu et al., "On Reusing the Results of Pre-executed Instructions in a Runahead Execution Processor," *Computer Architecture Letters*, vol. 4, Jan. 2005, http://www.cs.virginia.edu/~tcca/2005/mutlu_jan05.pdf.
8. N. Kirman et al., "Checkpointed Early Load Retirement," *Proc. 11th Int'l Symp. High-Performance Computer Architecture* (HPCA-11), IEEE CS Press, 2005, pp. 16–27.
9. R.S. Chappell et al., "Simultaneous Subordinate Microthreading (SSMT)," *Proc. 26th Int'l Symp. Computer Architecture* (ISCA 99), IEEE CS Press, 1999, pp. 186–195.
10. J.D. Collins et al., "Dynamic Speculative Precomputation," *Proc. 34th Int'l Symp. Microarchitecture* (Micro-34), IEEE CS Press, 2001, pp. 306–317.

Onur Mutlu is a PhD candidate in computer engineering at the University of Texas at Austin. His research interests include computer architectures, with a focus on high-performance energy-efficient microarchitectures, data prefetching, runahead execution, and novel latency-tolerance techniques. Mutlu has an MS in computer engineering from UT Austin and BS degrees in psychology and computer engineering from the University of Michigan. He is a student member of the IEEE and the ACM.

Hyesoon Kim is a PhD candidate in electrical and computer engineering at the University of Texas at Austin. Her research interests include high-performance energy-efficient microarchitectures and compiler-microarchitecture interaction. Kim has master's degrees in mechanical engineering from Seoul National University and in computer engineering from UT Austin. She is a student member of the IEEE and the ACM.

Yale N. Patt is the Ernest Cockrell Jr. Centennial Chair in Engineering at the University of Texas at Austin. His research interests include harnessing the expected fruits of future process technology into more effective microarchitectures for future microprocessors. Patt has a PhD in electrical engineering from Stanford University. He is co-author of Introduction to Computing Systems: From Bits and Gates to C and Beyond, (McGraw-Hill, 2nd edition, 2004). His honors include the 1996 IEEE/ACM Eckert-Mauchly Award and the 2000 ACM Karl V. Karlstrom Award. He is a Fellow of both the IEEE and the ACM.

Direct questions and comments about this article to Onur Mutlu, 2501 Lake Austin Blvd., Apt. N204, Austin, TX 78703; onur@ece.utexas.edu.

For further information on this or any other computing topic, visit our Digital Library at http://www.computer.org/publications/dlib.