Efficient Runahead Execution: Power-efficient Memory Latency Tolerance

Onur Mutlu Hyesoon Kim Yale N. Patt

Department of Electrical and Computer Engineering
University of Texas at Austin
{onur,hyesoon,patt}@ece.utexas.edu

Abstract

Runahead execution improves memory latency tolerance without significantly increasing processor complexity. Unfortunately, a runahead execution processor executes significantly more instructions than a conventional processor, sometimes without providing any performance benefit, which makes it inefficient. In this article, we identify the causes of inefficiency in runahead execution and propose simple -yet effective- techniques to make a runahead processor more efficient, thereby reducing its energy consumption. The proposed efficiency techniques reduce the extra instructions executed in a runahead processor from 26.5% to 6.2% without significantly affecting the 22% performance improvement provided by runahead execution.

1. Introduction

Today's high-performance processors are facing main memory latencies in the order of hundreds of processor clock cycles. As a result, even the most aggressive state-of-the-art processors end up spending a significant portion of their execution time stalling and waiting for main memory accesses to return data into the execution core. Previous research has shown that "runahead execution" is a technique that significantly increases the ability of a high-performance processor to tolerate long main memory latencies [1, 2, 3]. Runahead execution improves the performance of a processor by speculatively pre-executing the application program while a long-latency (L2) data cache miss is being serviced, instead of stalling the processor for the duration of the long-latency miss. Thus, runahead execution allows the execution of instructions that cannot be executed by a state-of-the-art processor under a long-latency cache miss. These pre-executed instructions generate prefetches that will later be used by the application program, which results in performance improvement.

Runahead execution is a promising way to tolerate long main memory latencies, since it has very modest hardware cost and it does not significantly increase processor complexity [4]. However, runahead execution significantly increases the dynamic energy consumption of a processor by increasing the number of speculatively processed (executed) instructions, sometimes without providing any performance benefit.

To be efficiently implementable in current or future high-performance processors, which will be energy-constrained, the extra instructions executed due to runahead execution need to be reduced. Our article provides novel solutions to this problem with both hardware and software mechanisms that are simple, implementable, and effective. We first present a brief overview of runahead execution and then describe the techniques that increase its power-efficiency.

2. Background on Runahead Execution

Conventional out-of-order execution processors employ instruction windows to buffer instructions in order to tolerate long latencies. Since a cache miss to main memory takes hundreds of processor cycles to service, the number of instructions that need to be buffered to tolerate such a long latency is unreasonably large. Runahead execution [2] is a technique that 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.

Figure 1 shows an example execution timeline illustrating the differences between the operation of a conventional out-of-order execution processor and a runahead execution processor. The instruction window of a conventional processor becomes full soon after a load instruction incurs a long-latency (L2) cache miss. Once the instruction window is full, the processor cannot decode and process any new instructions and stalls until the L2 cache miss is serviced. While the processor is stalled, no forward progress is made in the running application. Therefore, the execution timeline of a memory-intensive application on a conventional processor consists of useful computation (COMPUTE) periods interleaved with long useless STALL periods due to L2 cache misses, as shown in Figure 1(a). 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 shown in Figure 1(b). 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, which is called the "runahead mode." The processor removes this long-latency instruction from the instruction window. While in runahead mode, the processor continues

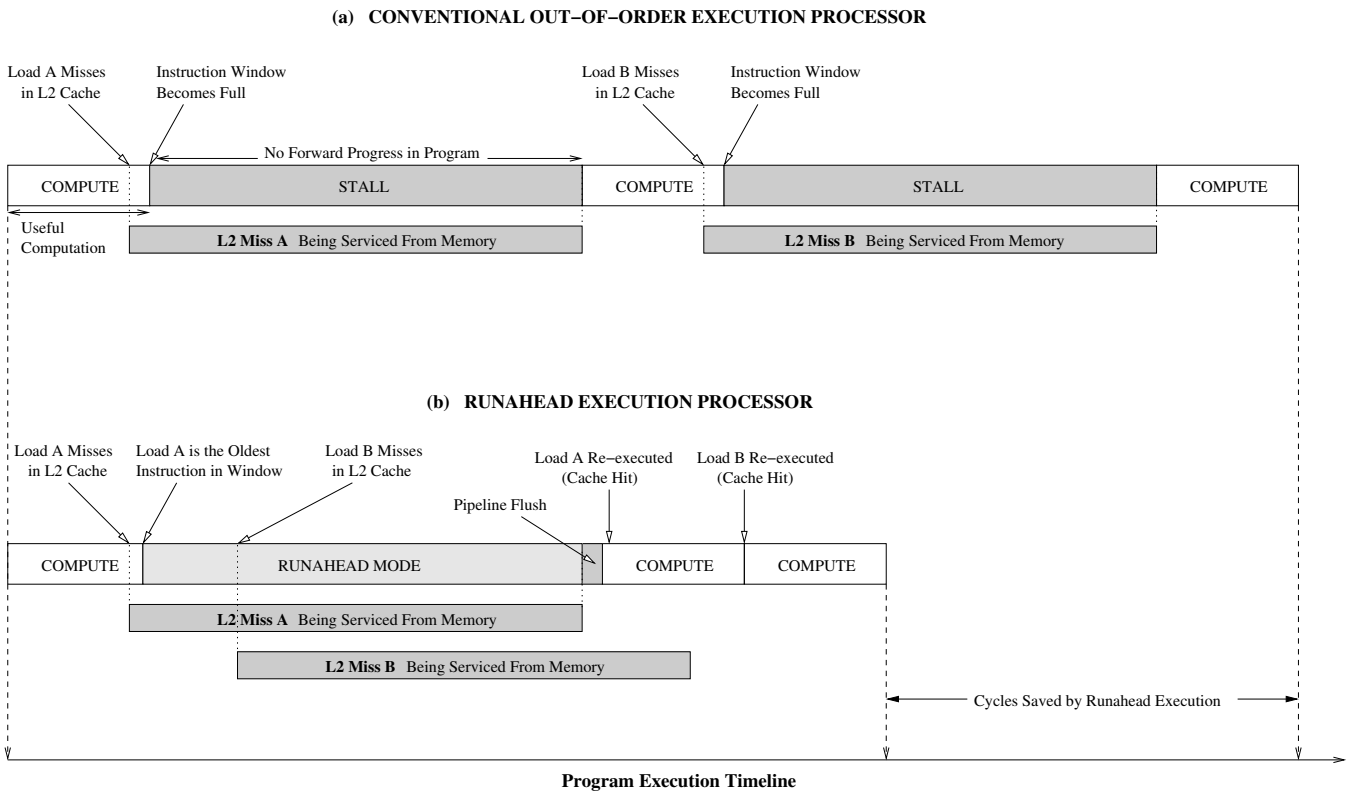

Figure 1. Execution timeline showing a high-level overview of the concept of runahead execution. A runahead processor pre-executes the running application during cycles in which a conventional processor would be stalled due to an L2 cache miss. The purpose of this pre-execution is to discover and service in parallel additional L2 cache misses. In this example, runahead execution discovers L2 Miss B and services it in parallel with L2 Miss A, thereby eliminating a stall that would be caused by Load B in a conventional processor.

to execute instructions without updating the architectural state. The results of L2 cache misses and their dependents are identified as bogus (INV). Instructions that source INV results (INV instructions) are removed from the instruction window so that they do not prevent independent instructions from being placed into the window. The removal of instructions from the processor during runahead mode is accomplished in program order and it is called "pseudo-retirement." Some of the instructions executed in runahead mode that are independent of L2 cache misses may miss in the instruction, data, or unified caches (e.g., Load B in Figure 1(b)). Their miss latencies are overlapped 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 1(b)).

Once the processor returns to "normal mode," it is able to make faster progress without stalling because some of the data and instructions needed during normal mode have already been prefetched into the caches during runahead mode. For example, in Figure 1(b), the processor does not need to stall for Load B because the L2 miss caused by Load B was discovered in runahead mode and serviced in parallel with the L2 miss caused by Load A. Hence, runahead execution uses otherwise-idle clock cycles due to L2 misses to pre-execute the application in order to generate accurate prefetch requests. Previous research has shown that runahead execution increases processor performance mainly because it parallelizes independent long-latency L2 cache misses [4, 3]. Furthermore, the memory latency tolerance provided by runahead execution comes at a small hardware cost, as shown in our previous papers [2, 4].

3. Efficiency of Runahead Execution

A runahead processor executes some instructions in the instruction stream more than once because it speculatively pre-executes instructions in runahead mode. As each execution of an instruction consumes dynamic energy, a runahead processor consumes more dynamic energy than a processor that does not implement runahead execution. To reduce the energy consumed by a runahead processor, it is desirable to reduce the number of instructions executed in runahead mode. Unfortunately, reducing the number of instructions executed during runahead mode may significantly re-

duce the performance improvement of runahead execution, since 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 the *efficiency* of a runahead processor without significantly decreasing its IPC (Instructions Per Cycle) performance improvement. We define efficiency as follows:

$$Efficiency = \frac{Percent\ Increase\ In\ IPC\ Performance}{Percent\ Increase\ In\ Executed\ Instructions}$$

Percent Increase In IPC Performance is the IPC increase after the addition of runahead execution to a conventional baseline processor. *Percent Increase In Executed Instructions* is the increase in the number of executed instructions after the addition of runahead execution. Efficiency of a runahead processor can be increased in two ways. First, the number of executed instructions (the denominator) can be reduced without affecting the increase in IPC (the nominator). This can be accomplished by eliminating the causes of inefficiency. Second, the IPC improvement can be increased without increasing the number of executed instructions. This can be accomplished by increasing the usefulness of each runahead execution period by extracting more useful prefetches from the executed instructions. We propose techniques to increase efficiency in both ways.

Figure 2 shows the increase in IPC and increase in the number of executed instructions due to the addition of runahead execution to an aggressive conventional out-of-order processor. Our baseline processor model includes a very effective stream-based prefetcher, a 1 MB L2 cache, and a detailed model of a main memory system with a 500-cycle latency. Detailed information on our experimental methodology is provided in our ISCA-32 paper [5]. On average, for the SPEC CPU2000 benchmarks, runahead execution increases the IPC by 22.6% at a cost of increasing the number of executed instructions by 26.5%. 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 increases the number of executed instructions by 47.8% while decreasing the IPC by 0.8%. In art, there is an impressive 108.4% IPC increase, only to be overshadowed by a 235.4% increase in the number of executed instructions.

4. Eliminating the Causes of Inefficiency

We have identified three major causes of inefficiency in a runahead execution processor: *short*, *overlapping*, and *useless* runahead periods. Runahead execution episodes with these properties usually do not provide performance benefit but result in unnecessary speculative execution of instructions. As exit from runahead execution is costly in terms of performance (it requires a full pipeline flush), such runahead periods can actually be detrimental to performance.

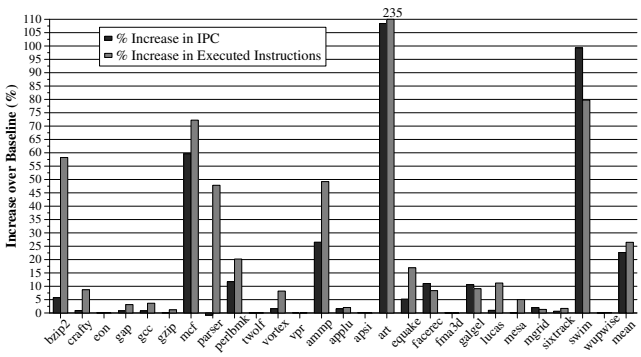

Figure 2. Increase in IPC performance and executed instructions due to runahead execution.

This section proposes simple techniques to eliminate such periods.

4.1. Short Runahead Periods

In a *short* runahead period, the processor stays in runahead mode for tens, instead of hundreds of cycles. These periods occur because the processor may start runahead execution due to an already outstanding L2 cache miss, which was prefetched by the hardware or software prefetcher, a wrong-path instruction, or a previous runahead period, but has not yet completed. Figure 3(a) shows a short runahead period that occurs due to an incomplete prefetch generated by a previous runahead period. Load B generates an L2 miss when it is speculatively pre-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 after, L2 Miss B is completely serviced by the memory system and the processor exits runahead mode. Hence, the runahead period caused by Load B is short. Short runahead periods are not desirable because the processor is unlikely to pre-execute enough instructions far ahead into the instruction stream and hence unlikely to uncover any useful L2 cache misses during runahead mode.

We propose to 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 does not 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% with negligible impact on performance (Performance improvement is reduced slightly from 22.6% to 21.5%).

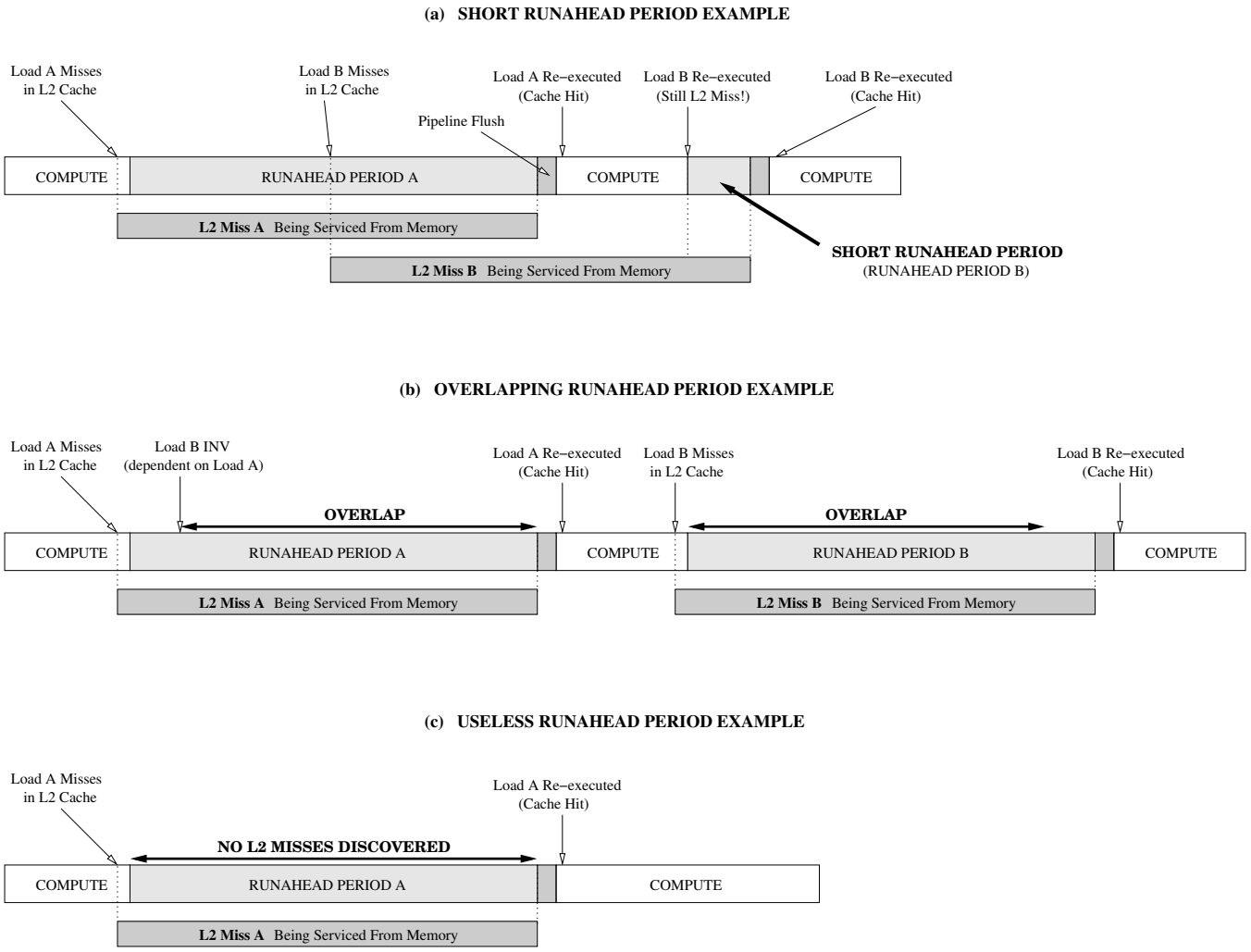

Figure 3. Example execution timelines illustrating the causes of inefficiency in runahead execution (short, overlapping, useless runahead periods) and how they may occur.

4.2. Overlapping Runahead Periods

Two runahead periods are *overlapping* if some of the instructions the processor executes in both periods are the same dynamic instructions. These periods can occur due to dependent L2 misses (e.g., due to pointer-chasing loads) or independent L2 misses that have significantly different latencies. Figure 3(b) shows an example where runahead periods A and B are overlapping due to dependent L2 misses. During period A, Load B is processed and found to be dependent on the miss caused by Load A. Since L2 Miss A has not been serviced yet, Load B cannot calculate its address and is marked as INV. The processor executes and pseudo-retires N instructions after load B and exits period A. In normal mode, Load B is re-executed and found 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 same work done by period A.

Overlapping runahead periods may be beneficial for performance, because the completion of Load A may result in the availability of data values for more instructions in runahead period B, which may result in the generation of useful L2 misses that could not have been generated in runahead period A. However, in the benchmark set we examined, we found that overlapping runahead periods are usually not beneficial for 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 pseudo-retired 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 pseudo-retired in the previous runahead period, the processor does not enter runahead mode. With two simple counters and a comparator needed to implement this technique, extra instructions due to runahead execution are reduced from 26.5% to 11.8% while the performance benefit is reduced only slightly from 22.6% to 21.2%.

4.3. Useless Runahead Periods

Useless runahead periods are runahead periods where no useful L2 misses that are needed by normal mode execution are generated, as shown in Figure 3(c). These periods exist due to the lack of memory-level parallelism [6, 3] in the application program, i.e. due to the lack of independent cache misses under the shadow of an L2 miss. Useless periods are inefficient because they increase the number of executed instructions without providing any performance benefit. To eliminate a useless runahead period, we propose four simple, novel prediction mechanisms that predict whether or not a runahead period will be useful (i.e., generate an L2 cache miss).

In the first technique, the processor records the usefulness of past runahead periods caused by static load instructions in a small table of two-bit counters, which is called the Runahead Cause Status Table (RCST) [5]. If the runahead periods initiated by the same load were useful in the recent past, the processor initiates runahead execution if that load misses in the L2 cache. Otherwise, the processor does not initiate entry into runahead mode on an L2 miss due to the static load instruction. The insight behind this technique is that the usefulness of runahead periods caused by the same static load tend to be predictable based on recent past behavior.

The second technique predicts the available memory-level parallelism during the ongoing runahead period. If the fraction of INV (i.e., L2-miss dependent) load instructions encountered during the ongoing runahead mode is greater than a statically-determined threshold, the processor predicts that there is not enough memory-level parallelism runahead execution can exploit and exits runahead mode.

The third technique predicts the usefulness of runahead execution in a more coarse-grain fashion using *sampling*. The purpose of this technique is to turn off runahead execution in program phases where memory-level parallelism is low. 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 does not enter runahead mode for the next M L2 misses. We found that even with un-tuned values of N, M, and T (100, 1000, and 25, respectively, in our experiments), sampling can significantly reduce the extra instructions due to runahead execution.

The fourth uselessness prediction technique leverages compile-time profiling. The compiler profiles the application and identifies load instructions which consistently cause useless runahead periods. Such load instructions are marked as *non-runahead load*s by the compiler. When the hardware encounters a non-runahead load instruction that is an L2 cache miss, it does not initiate runahead execution on that load.

Combining the four uselessness prediction techniques reduces the extra instructions from 26.5% to 14.9% while the performance benefit is reduced only slightly from 22.6% to 20.8%. Experiments analyzing the effectiveness of each individual technique can be found in a previous paper [5].

5. Increasing the Usefulness of Runahead Periods

As the performance improvement of runahead execution is mainly due to the useful L2 misses prefetched during runahead mode [4, 3], it can be increased with optimizations that lead to the discovery of more L2 misses during runahead mode. This section proposes two optimizations that increase efficiency by increasing the usefulness of runahead periods.

5.1. Eliminating Useless Instructions

Since the goal of runahead execution is to generate L2 cache misses, instructions that do not contribute to the generation of L2 cache misses are essentially "useless" for the purposes of runahead execution. Therefore, the usefulness of a runahead period can be increased by eliminating instructions that do not lead into the generation of L2 cache misses during runahead mode.

One example of such useless instructions are floating-point (FP) operate instructions, which do not contribute to the address computation of load instructions. Thus, we propose that the FP unit be turned off during runahead mode and FP operate instructions be dropped after being decoded. With this optimization, the processor resources are spared for more useful instructions that lead into the generation of load/store addresses, which increases the likelihood of generating an L2 miss during a runahead period. Furthermore, significant dynamic and static energy savings can be enabled by not executing the energy-intensive FP instructions and powering down the FP unit during runahead mode. On the other hand, turning off the FP unit during runahead mode has one disadvantage that can reduce performance. If a control-flow instruction that depends on the result of an FP instruction is mispredicted during runahead mode, the processor would have no way of recovering from that misprediction if the FP unit is turned off, since the source operand of the branch would not be computed. Nevertheless, our simulations show that turning off the FP unit is a valuable optimization that both increases the performance

improvement of runahead execution (from 22.6% to 24.0%) *and* reduces the extra instructions (from 26.5% to 25.5%).

5.2. Optimizing the Interaction Between Runahead Execution and the Hardware Prefetcher

One of the potential benefits of runahead execution is that the hardware data prefetcher can be updated 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 not accurate, they will waste memory bandwidth and may 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 the effectiveness of runahead execution.

Runahead execution and hardware data prefetching were shown to have synergistic behavior [2, 7]. 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, since 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 the IPC improvement of runahead execution (from 22.6% to 25.0%) and also reduces the extra instructions (from 26.5% to 24.7%).

6. Putting It All Together

Figure 4 shows the increase in executed instructions and IPC due to runahead execution when all the techniques proposed in the previous two sections are incorporated into a runahead processor. The effect of profiling-based useless period elimination is examined separately, since it requires modifications to the instruction set architecture (ISA).

Applying all the proposed techniques significantly reduces the average increase in executed instructions in a runahead processor, from 26.5% to only 6.7% (6.2% with profiling). The average IPC increase of a runahead processor which uses the proposed techniques is reduced slightly from 22.6% to 22.0% (22.1% with profiling). Hence, a runahead processor employing the proposed techniques is much more efficient than a traditional runahead processor, but it still increases performance almost as much as a traditional runahead processor does.

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 due to runahead execution increase. Hence, runahead execution is more effective at longer memory latencies. For almost all memory latencies, employing the proposed efficiency techniques increases the average IPC improvement on the floating-point (FP) benchmarks while only slightly reducing the IPC improvement on the integer (INT) benchmarks. For all memory latencies, employing the proposed dynamic techniques significantly reduces the extra instructions.

7. Future Research Directions

We have described the major causes of inefficiency in runahead execution and proposed simple and effective ways of increasing the efficiency of a runahead processor. Orthogonal approaches can be developed to solve the same problem, which, we believe, is an important research area in runahead execution and memory latency tolerance techniques in general. In particular, solutions to two important problems in computer architecture can significantly increase the efficiency of runahead execution: branch mispredictions and dependent cache misses.

Since the processor relies on correct branch predictions to stay on the correct program path during runahead mode, the development of more accurate branch predictors will increase both the efficiency and the performance benefits of runahead execution. Irresolvable branch mispredictions that are dependent on long-latency cache misses cause the processor to stay on the wrong-path, which may not always provide useful prefetching benefits, until the end of the runahead period. Reducing such branch mispredictions with novel techniques is a promising area of future work.

Dependent long-latency cache misses reduce the usefulness of a runahead period because they cannot be parallelized using runahead execution. Therefore, runahead execution is inefficient, and sometimes ineffective, for pointer-chasing workloads where dependent load instructions are common. In a recent paper [8], we showed that a simple novel value prediction technique for pointer load instructions, called "address-value delta prediction," significantly increases the efficiency and performance of runahead execution by parallelizing dependent L2 cache misses. We believe techniques that enable 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 very 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 may be able to increase the usefulness of runahead periods by trying to arrange code such that independent L2 cache misses are clustered close together during program execution.

The efficiency of runahead execution can potentially be increased by eliminating the re-execution of instructions executed in runahead mode via result reuse [9] or value prediction [10]. However, even an ideal reuse mechanism does not

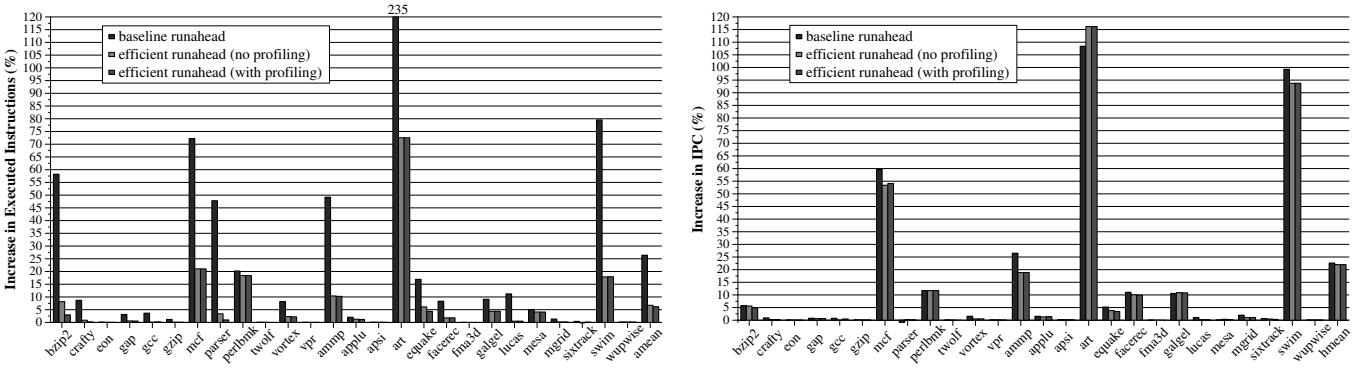

Figure 4. Increase in executed instructions and IPC due to runahead execution after incorporating all the efficiency techniques.

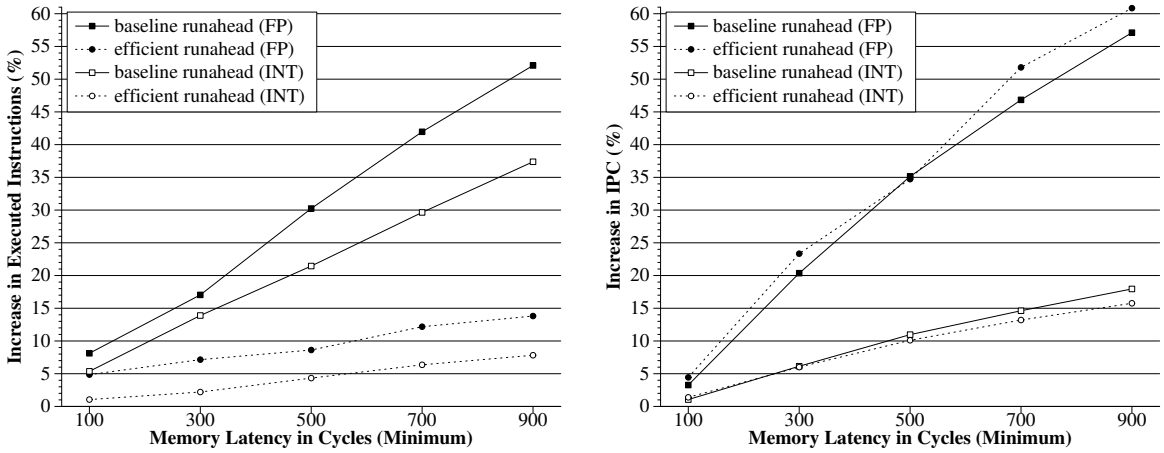

Figure 5. Extra instructions and IPC increase with and without the efficiency techniques for five different memory latencies. Data shown are averaged separately over integer (INT) and floating-point (FP) benchmarks.

significantly improve performance [9] and it likely has significant hardware cost and complexity, which may offset the energy reduction due to improved efficiency. Value prediction may not significantly improve efficiency due to its low accuracy [10]. Nevertheless, further research on eliminating the unnecessary re-execution of instructions may yield low-cost mechanisms that can significantly improve runahead efficiency.

Finally, the scope of the efficient processing techniques is not limited to only runahead execution. In general, the proposed runahead uselessness prediction techniques are techniques for predicting the available MLP at a given point in a program. Therefore, the techniques are applicable to other mechanisms that are designed to exploit MLP. We believe other methods of pre-execution that are targeted for prefetching, such as helper threads [11, 12], can benefit from the proposed efficiency techniques to eliminate inefficient threads and useless speculative execution.

8. Related Work on Runahead Execution

As a promising technique to increase tolerance to main memory latency, runahead execution has recently inspired and attracted research from many other computer architects from both the industry [3, 7] and the academia [10, 13, 14]. Architects from Sun Microsystems described that they are implementing a version of runahead execution in their next-generation microprocessor [15]. To our knowledge, none of the previous work addressed the efficiency problem in runahead execution. We hereby provide a very brief overview of related work in runahead execution.

Dundas and Mudge [1] first proposed runahead execution as a means to improve the performance of an in-order scalar processor. We [2] proposed runahead execution to increase the main memory latency tolerance of more aggressive out-of-order superscalar processors. Chou et al. [3] demonstrated that runahead execution is very effective in improving memory-level parallelism in large-scale database benchmarks because it prevents the instruction and scheduling windows, along with serializing instructions from being performance bottlenecks. Three recent papers [3, 10, 13] proposed combining runahead execution with value prediction and Zhou [14] proposed using an idle processor core to perform runahead execution in a chip multiprocessor. The efficiency mechanisms we propose in this article can be ap-

plied to these variants of runahead execution to improve their power-efficiency.

9. Conclusion

In today's power-constrained processor design environment, memory latency tolerance techniques need to be energy efficient. Inefficient techniques that provide good latency tolerance can be unimplementable or very costly to implement because they may require significant hardware cost/complexity and energy cost. "Efficient runahead execution" has two major advantages. First, it does not require large, complex, and power-hungry structures to be implemented in the processor core. Instead, it utilizes the already-existing processing structures to improve memory latency tolerance. Second, with the simple efficiency techniques described in this article, it requires only a small number of extra instructions to be speculatively executed in order 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.

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 paper. We gratefully acknowledge the generous support of the Cockrell Foundation and Intel Corporation.

References

[1] James Dundas and Trevor Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the 1997 International Conference on Supercomputing*, pages 68–75, 1997.

[2] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of the 9th International Symposium on High Performance Computer Architecture*, pages 129–140, 2003.

[3] Yuan Chou, Brian Fahs, and Santosh Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. In *Proceedings of the 31st International Symposium on Computer Architecture*, pages 76–87, 2004.

[4] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt. Runahead execution: An effective alternative to large instruction windows. *IEEE Micro*, 23(6):20–25, Nov./Dec. 2003.

[5] Onur Mutlu, Hyesoon Kim, and Yale N. Patt. Techniques for efficient processing in runahead execution engines. In *Proceedings of the 32nd International Symposium on Computer Architecture*, pages 370–381, 2005.

[6] Andy Glew. MLP yes! ILP no! In *ASPLOS Wild and Crazy Idea Session '98*, October 1998.

[7] Sorin Iacobovici, Lawrence Spracklen, Sudarshan Kadambi, Yuan Chou, and Santosh G. Abraham. Effective stream-based and execution-based data prefetching. In *Proceedings of the 18th International Conference on Supercomputing*, pages 1–11, 2004.

[8] Onur Mutlu, Hyesoon Kim, and Yale N. Patt. Address-value delta (AVD) prediction: Increasing the effectiveness of runahead execution by exploiting regular memory allocation patterns. In *Proceedings of the 38th International Symposium on Microarchitecture*, 2005.

[9] Onur Mutlu, Hyesoon Kim, Jared Stark, and Yale N. Patt. On reusing the results of pre-executed instructions in a runahead execution processor. *Computer Architecture Letters*, 4, January 2005.

[10] Nevin Kırman, Meyrem Kırman, Mainak Chaudhuri, and José F. Martínez. Checkpointed early load retirement. In *Proceedings of the 11th International Symposium on High Performance Computer Architecture*, pages 16–27, 2005.

[11] Robert S. Chappell, Jared Stark, Sangwook P. Kim, Steven K. Reinhardt, and Yale N. Patt. Simultaneous subordinate microthreading (SSMT). In *Proceedings of the 26th International Symposium on Computer Architecture*, pages 186–195, 1999.

[12] Jamison D. Collins, Dean M. Tullsen, Hong Wang, and John P. Shen. Dynamic speculative precomputation. In *Proceedings of the 34th International Symposium on Microarchitecture*, 2001.

[13] Luis Ceze, Karin Strauss, James Tuck, Jose Renau, and Josep Torrellas. CAVA: Hiding L2 misses with checkpoint-assisted value prediction. *Computer Architecture Letters*, 3, December 2004.

[14] Huiyang Zhou. Dual-core execution: Building a highly scalable single-thread instruction window. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 231–242, 2005.

[15] Shailender Chaudhry, Paul Caprioli, Sherman Yip, and Marc Tremblay. High-performance throughput computing. *IEEE Micro*, 25(3):32–45, May 2005.