Techniques for Efficient Processing in Runahead Execution Engines

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 is a technique that improves processor performance by pre-executing the running application instead of stalling the processor when a long-latency cache miss occurs. Previous research has shown that this technique significantly improves processor performance. However, the efficiency of runahead execution, which directly affects the dynamic energy consumed by a runahead processor, has not been explored. A runahead processor executes significantly more instructions than a traditional out-of-order processor, sometimes without providing any performance benefit, which makes it inefficient. In this paper, we describe the causes of inefficiency in runahead execution and propose techniques to make a runahead processor more efficient, thereby reducing its energy consumption and possibly increasing its performance.

Our analyses and results provide two major insights: (1) the efficiency of runahead execution can be greatly improved with simple techniques that reduce the number of short, overlapping, and useless runahead periods, which we identify as the three major causes of inefficiency, (2) simple optimizations targeting the increase of useful prefetches generated in runahead mode can increase both the performance and efficiency of a runahead processor. The techniques we propose reduce the increase in the number of instructions executed due to runahead execution from 26.5% to 6.2%, on average, without significantly affecting the 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 the long main memory latencies [5, 13, 2]. Runahead execution improves the performance of a processor by speculatively pre-executing the application program while a long-latency 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 data cache miss. These pre-executed instructions generate prefetches that will later be used by the application program, which results in performance improvement.

A runahead processor executes significantly more instructions than a traditional out-of-order processor, sometimes without providing any performance benefit. This makes runahead execution inefficient and results in higher dynamic energy consumption than a traditional processor. To our knowledge, previous research has not explored the efficiency problems in a runahead processor. In this paper, we examine the causes of inefficiency in runahead execution and propose techniques to make a runahead processor more efficient. By making runahead execution more efficient, our goal is to reduce the dynamic energy consumption of a runahead processor and possibly increase its performance. The questions we answer in this paper to achieve this goal are:

1. As a runahead processor speculatively pre-executes portions of the instruction stream, it executes more instructions than a traditional high-performance processor, resulting in higher dynamic energy consumption. How can the processor designer decrease the number of instructions executed in a runahead processor, while still preserving most of the performance improvement provided by runahead execution? In other words, how can the processor designer increase the *efficiency* of a runahead processor? (Section 5)
2. As the pre-execution of instructions targets the generation of useful prefetches, instruction processing during runahead mode should be optimized for maximizing the number of useful prefetches generated during runahead execution. What kind of techniques increase the probability of the generation of useful prefetches and hence increase the performance of a runahead processor, while reducing or not significantly increasing the number of instructions executed? (Section 6)

2. Background on Runahead Execution

To provide the terminology used in this paper, we give a brief overview of the operation of runahead execution. For a thorough description, we refer the reader to [13].

Runahead execution avoids stalling the processor when a long-latency L2 cache miss blocks instruction retirement, preventing new instructions from being placed into the instruction window. When the processor detects that the oldest instruction is a long-latency cache miss that is still being serviced, it checkpoints the architectural register state, the branch history register, and the return address stack, records the program counter of the blocking long-latency instruction 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 to execute instructions without updating the architectural state and without blocking retirement due to long-latency cache misses and instructions dependent on them. The results of the long-latency 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 window during runahead mode is accomplished in program order and it is called "pseudo-retirement." Some of the instructions in runahead mode that are independent of long-latency cache misses may miss in the instruction, data, or unified caches. Their miss latencies are overlapped with the latency of the runahead-causing cache miss. When the runahead-causing cache miss completes, the processor exits the speculative 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. 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. Previous research showed that runahead execution increases performance mainly because it parallelizes independent long-latency L2 cache misses [14, 2].

3. Methodology

We first describe our baseline processor and experimental evaluation methodology. Analyses and results presented in later sections are based on the baseline processor described in this section.

Our baseline processor is an aggressive superscalar, out-of-order processor that implements the Alpha ISA. Machine parameters are summarized in Table 1. As the performance impact of runahead execution is highly dependent on the accuracy of the memory model, we use a detailed memory model, which models bandwidth, port contention, bank conflicts, and queueing delays at every level in the memory system. We use a large, 1 MB L2 cache. An aggressive state-of-the-art stream-based prefetcher similar to the one described in [15] is also employed in the baseline processor. All experiments are performed using the SPEC 2000 Integer (INT) and Floating Point (FP) benchmarks. INT benchmarks are run to completion with a reduced input set [9]. FP benchmarks are simulated using the reference input set. The initialization portion is skipped for each FP benchmark and simulation is done for the next 250 million instructions.

Table 1. Baseline processor configuration.

Front End	64KB, 4-way I-cache; 8-wide fetch, decode, rename; 64K-entry gshare/PAs hybrid branch pred.; min. 20-cycle mispred. penalty; 4K-entry, 4-way BTB; 64-entry RAS; 64K-entry indirect target cache
Execution Core	128-entry reorder buffer; 8 functional units; 128-entry st buffer; store misses do not block retirement unless store buffer is full
Caches	64KB, 4-way, 2-cycle L1 D-cache, 128 L1 MSHRs, 4 load ports; 1MB, 32-way, 10-cycle unified L2, 1 ld/st port; 128 L2 MSHRs; all caches have LRU replacement and 64B line size; 1-cycle AGEN
Memory	500-cycle min. latency; 32 banks; 32B-wide, split-trans. core-to-mem. bus at 4:1 freq. ratio; conflicts, bandwidth, and queueing modeled
Prefetcher	Stream-based [15]; 32 stream buffers; can stay 64 cache lines ahead

Figure 1 shows, for reference, the IPC (retired Instructions Per Cycle) performance of four processors for each benchmark: from left to right, a processor with no prefetcher, the baseline processor, the baseline processor with runahead, and the baseline processor with a perfect (100% hit rate) L2 cache. All IPC averages are calculated as the harmonic average. The baseline prefetcher is quite effective and improves the average IPC by 90.4%.

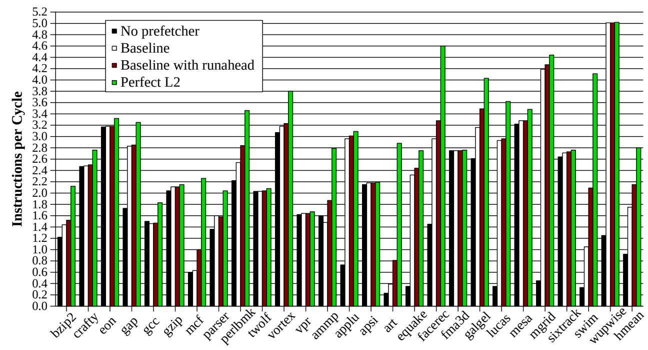

Figure 1. Baseline IPC performance.

4. The Problem: Inefficiency of Runahead Execution

Runahead execution increases processor performance by pre-executing the instruction stream while an L2 cache miss is in progress. At the end of a runahead execution period, the processor restarts its pipeline beginning with the instruction that caused entry into runahead mode. Hence, a runahead processor executes some instructions in the instruction stream more than once. 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 during runahead mode. Unfortunately, reducing the number of instructions executed during runahead mode may significantly reduce the performance improvement of runahead execution, since runahead execution relies on the execution of instructions during runahead mode to discover useful prefetches. Our goal is to increase the *efficiency* of a runahead processor without significantly decreasing its IPC performance improvement. We define efficiency as follows:

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

Percent Increase In IPC is the percentage IPC increase after the addition of runahead execution to the baseline processor. *Percent Increase In Executed Instructions* is the percentage increase in the number of executed instructions after the addition of runahead execution.[1] We use this definition, because it is congruent with the $\Delta Performance/\Delta Power$ metric used in power-aware design to decide whether or not a new microarchitectural feature is power aware [7]. We examine techniques that increase efficiency by reducing the *Percent Increase In Executed Instructions* in Section 5. Techniques that increase efficiency by increasing the *Percent Increase In IPC* are examined in Section 6.

Note that efficiency *by itself* is not a very meaningful metric. Observing the increase in the executed instructions *and* the increase in IPC *together* gives a better view of both efficiency and performance, especially because our goal is to increase efficiency without significantly reducing performance. Therefore, we always report changes in these two metrics. Efficiency values can be easily computed using these two metrics.

Figure 2 shows the increase in IPC and increase in the number of executed instructions due to the addition of runahead execution to our baseline processor. All instructions executed in the processor core, INV or valid, are counted to obtain the number of executed instructions. On average, 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% (efficiency $= -0.8/47.8 = -0.02$). 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 (efficiency $= 108.4/235.4 = 0.46$).

5. Techniques for Improving Efficiency

We have identified three major causes of inefficiency in a runahead processor: short, overlapping, and useless runahead periods. This section describes these causes and proposes techniques to eliminate them. For the purposes of these studies, we only consider those benchmarks with an IPC increase of more than 5% *or* with an executed instruction increase of more than 5%.

[1] There are other ways to define efficiency. We also examined a definition based on *Percent Increase In Fetched Instructions*. This definition gave similar results to the definition provided in this paper, since the increase in the number of fetched instructions due to runahead execution is very similar to the increase in the number of executed instructions.

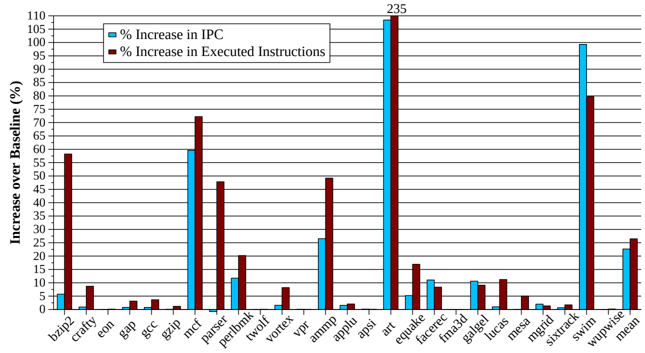

Figure 2. Increase in IPC and executed instructions due to runahead execution.

5.1. Eliminating Short Runahead Periods

One cause of inefficiency in runahead execution is short runahead periods, where the processor stays in runahead mode for tens, instead of hundreds, of cycles. A short runahead period can occur because the processor may enter runahead mode due to an L2 miss that was already prefetched by the prefetcher, a wrong-path instruction, or a previous runahead period, but that has not completed yet. Short runahead periods are not desirable, because the processor may not be able to pre-execute enough instructions far ahead into the instruction stream and hence may not be able to generate any useful prefetches during runahead mode. As exit from runahead execution is costly (it requires a full pipeline flush), short runahead periods can actually be detrimental to performance.

Ideally, we would like to know when an L2 cache miss is going to return back from main memory. If the L2 miss is going to return soon enough, the processor can decide not to enter runahead mode on that L2 miss. Unfortunately, in a realistic memory system, latencies to main memory are variable and are not known beforehand due to bank conflicts, queueing delays, and contention in the memory system. To eliminate the occurrence of short runahead periods, we propose a simple heuristic to predict that an L2 miss is going to return back from main memory soon.

In our mechanism, the processor keeps track of the number of cycles each L2 miss has spent after missing in the L2 cache. Each L2 Miss Status Holding Register (MSHR) [10] contains a counter to accomplish this. When the request for a cache line misses in the L2 cache, the counter in the MSHR associated with the cache line is reset to zero. This counter is incremented periodically until the L2 miss for the cache line is complete. When a load instruction at the head of the instruction window is an L2 miss, the counter value in the associated L2 MSHR is compared to a threshold value T. If the counter value in the MSHR is greater than T, the processor does not initiate entry into runahead mode, predicting that the L2 miss will return back soon from main memory. We considered both statically and dynamically determined thresholds. A static threshold is fixed for a processor and

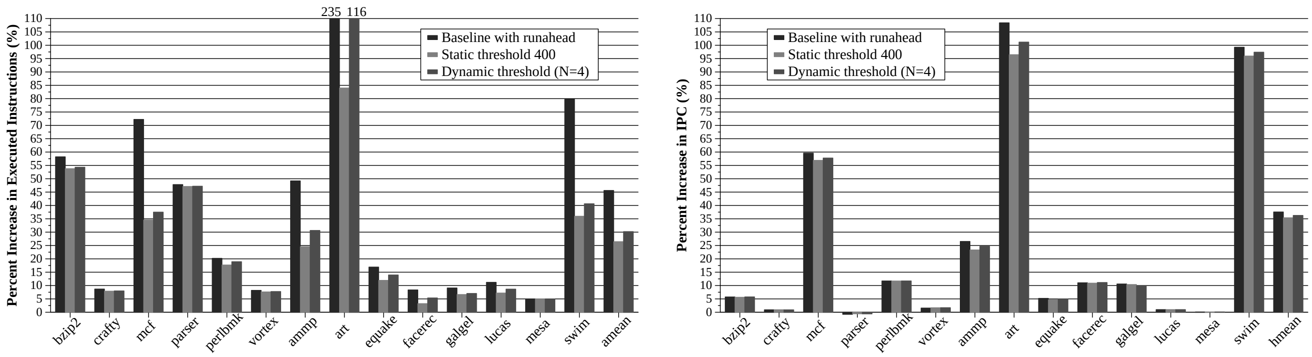

Figure 3. Increase in executed instructions and IPC after eliminating short runahead periods using thresholding.

can be set based on design-time estimations of main memory latency. As the memory latency in our baseline processor is 500 cycles, we examined thresholds between 250 to 500 cycles. A dynamic threshold can be set by computing the average latency of the last N L2 misses and not entering runahead execution if the current L2 miss has covered more than the average L2 miss latency (N is varied from 4 to 64 in our experiments). The best dynamic threshold we looked at did not perform as well as the best static threshold. Due to the variability in the L2 miss latency, it is not feasible to get an accurate prediction on the latency of the current miss based on the average latency of the last few misses.

Figure 3 shows the increase in number of executed instructions and IPC over the baseline processor if we employ the thresholding mechanisms. The best heuristic, in terms of efficiency, prevents the processor from entering runahead mode if the L2 miss has been in progress for more than 400 cycles. The increase in the number of executed instructions on the selected benchmarks is reduced from 45.6% to 26.4% with the best static threshold and to 30.2% with the best dynamic threshold, on average. Average IPC improvement is reduced slightly from 37.6% to 35.4% with the best static threshold and to 36.3% with the best dynamic threshold. Hence, eliminating short runahead periods using a simple miss latency thresholding mechanism significantly increases the efficiency of runahead execution.

Figure 4 shows the distribution of the runahead period length and the useful L2 misses prefetched by runahead load instructions for each period length in the baseline runahead processor (left graph) and after applying the *static threshold 400* mechanism (right graph). The data shown in this figure are averaged over all benchmarks. A useful miss is defined as an L2 load miss that is generated in runahead mode and later used in normal mode and that could not be captured by the processor's instruction window if runahead execution was not employed. Without the efficiency optimization, there are many short runahead periods that result in very few useful prefetches. For example, the runahead processor enters periods of shorter than 50 cycles 4981 times, but a total of only 22 useful L2 misses are generated during these periods (leftmost points in the left graph in Figure 4). Using the *static threshold 400* mechanism eliminates all occurrences of periods of shorter than 100 cycles, as shown in the right graph. This mechanism also eliminates some of the very long runahead periods (longer than 600 cycles), but we find that the efficiency loss due to eliminating a smaller number of long periods is more than offset by the efficiency gain due to eliminating a larger number of short periods.

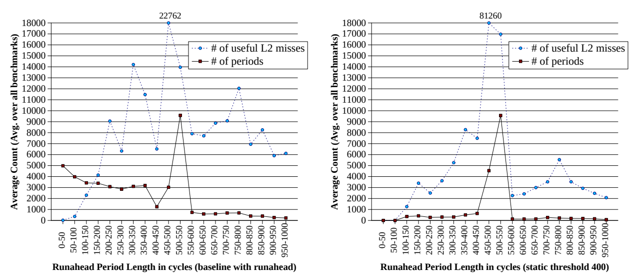

Figure 4. Distribution of runahead period length (in cycles) and useful misses generated for each period length.

5.2. Eliminating Overlapping Runahead Periods

Two runahead periods are defined to be overlapping if some of the instructions the processor executes in both periods are the same dynamic instructions. Overlapping periods occur due to two reasons:

1. *Dependent L2 misses (Figure 5a):* Load A causes entry into runahead period 1. During this period, load B is executed and found to be dependent on load A; therefore load B becomes INV. The processor executes and pseudo-retires N instructions after load B and exits runahead period 1 when the miss for load A returns from main memory. The pipeline is flushed and fetch is redirected to load A. In normal mode, load B is executed and found to be an L2 miss. The processor enters into runahead period 2 due to load B. In runahead period 2, the processor executes the same N instructions that were executed in period 1.
2. *Independent L2 misses with different latencies (Figure 5b):* This is similar to the previous case, except load A and load C are independent. The L2 miss caused by load C takes longer to service than the L2

miss caused by load A. Note that runahead period 2 may or may not also be a *short* period, depending on the latency of the L2 cache miss due to load C.

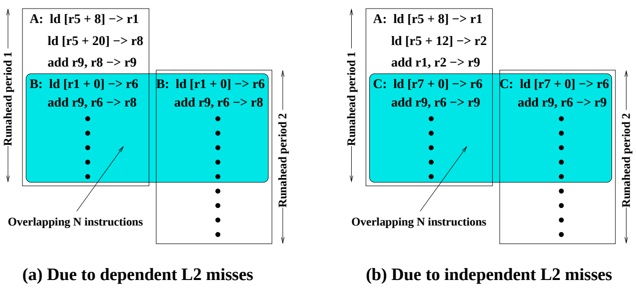

Figure 5. Overlapping runahead periods.

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 2, which may result in the generation of useful prefetches that could not have been generated in runahead period 1. On the other hand, if the availability of the result of load A does not lead to the generation of new load addresses that generate prefetches, the processor will execute the same N instructions twice in runahead mode without obtaining any benefit. In any case, overlapping runahead periods can be a major cause of inefficiency, because they result in the execution of the same instruction 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. When an L2 miss load at the head of the reorder buffer is encountered during normal mode, the processor compares these two counts. If the number of instructions fetched after the exit from runahead mode is less than the number of instructions pseudo-retired in the previous runahead period, the processor does not enter runahead mode (*full threshold policy*). Note that this mechanism is predictive. The processor may pseudo-retire instructions on the wrong-path during runahead mode due to the existence of an unresolvable mispredicted INV branch (called *divergence point* in [13]). Therefore, it is not guaranteed that a runahead period caused before fetching the same number of pseudo-retired instructions in the previous runahead period overlaps with the previous runahead period. Hence, this mechanism may eliminate some non-overlapping runahead periods. To reduce the probability of eliminating non-overlapping periods, we examined a policy (*half threshold policy*) where the processor does not enter runahead mode if it has not fetched more than half of the number of instructions pseudo-retired during the last runahead period.

Figure 6 shows the increase in number of executed instructions and IPC over the baseline processor if we employ the *half threshold* and *full threshold* policies. Eliminating overlapping runahead periods significantly reduces the increase in the number of executed instructions from 45.6% to 28.7% with the *half threshold* policy and to 20.2% with the *full threshold* policy, on average. This reduction comes with a small impact on IPC improvement, which is reduced from 37.6% to 36.1% with the *half threshold* policy and to 35% with the *full threshold* policy. Art, ammp, and swim are the only benchmarks that see significant reductions in IPC improvement because overlapping periods due to independent L2 misses sometimes provide useful prefetching benefits in these benchmarks. It is possible to recover the performance loss by predicting the usefulness of overlapping periods and eliminating only those periods predicted to be useless, using schemes similar to those described in the next section. We do not discuss this option due to space limitations.

5.3. Eliminating Useless Runahead Periods

Useless runahead periods that do not result in prefetches for normal mode instructions are another cause of inefficiency in a runahead processor. These periods exist due to the lack of memory-level parallelism [6] 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, the processor needs to know whether or not the period will provide prefetching benefits before initiating runahead execution on an L2 miss. As this is not possible without knowledge of the future, we use techniques to predict the usefulness of a runahead period. We describe several of the most effective techniques we examined.

5.3.1. Predicting Useless Periods Based on Past Usefulness of Runahead Periods Initiated by the Same Static Load. The first technique makes use of past information on the usefulness of previous runahead periods caused by a load instruction to predict whether or not to enter runahead mode on that static load instruction again. The usefulness of a runahead period is approximated by whether or not the period generated at least one L2 cache miss.[2] 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 processor uses a table of two-bit saturating counters called Runahead Cause Status Table (RCST) to collect information on each L2-miss load. The state diagram for an RCST entry is shown in Figure 7.

[2] Note that this is a heuristic and not necessarily an accurate metric for the usefulness of a runahead period. The generated L2 cache miss may actually be on the wrong path during runahead mode and may never be used in normal mode.

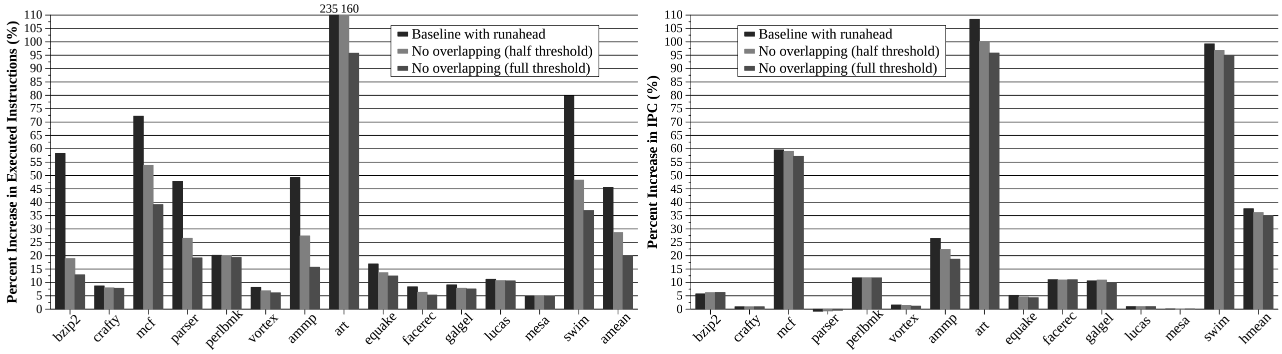

Figure 6. Increase in executed instructions and IPC after eliminating overlapping runahead periods.

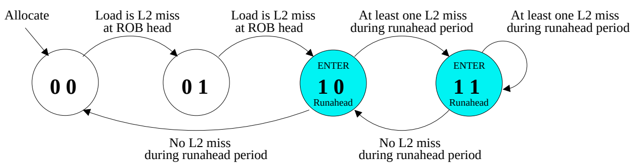

Figure 7. State diagram of the RCST counter.

When an L2-miss load is the oldest instruction in the instruction window, it accesses RCST using its instruction address to check whether it should initiate entry into runahead mode. If there is no counter associated with the load, runahead is not initiated, but a counter is allocated and reset. During each runahead period, the processor keeps track of the number of L2 load misses that are generated and cannot be captured by the processor's instruction window.[3] Upon exit from runahead mode, if there was at least one such L2 load miss generated during runahead mode, the two-bit counter associated with the runahead-causing load is incremented. If there was no such L2 miss, the two-bit counter is decremented. When the same static load instruction is an L2-miss at the head of the instruction window later, the processor accesses the counter associated with the load in RCST. If the counter is in state 00 or 01, runahead is not initiated, but the counter is incremented. We increment the counter in this case, because we do not want to ban any load from initiating entry into runahead. If we do not increment the counter, the load will never cause entry into runahead until its counter is evicted from RCST, which we found to be detrimental for performance because it eliminates many useful runahead periods along with useless ones. In our experiments, we used a 4-way RCST containing 64 counters.

5.3.2. Predicting Useless Periods Based on INV Dependence Information. The second technique we propose makes use of information that becomes available while the processor is in runahead mode. The purpose of this technique is to predict the available memory-level parallelism during the existing runahead period. If there is not enough memory-level parallelism, the processor exits runahead mode right away. To accomplish this, while in runahead mode, the processor keeps a count of the number of load instructions executed and a count of how many of those are INV (i.e., dependent on an L2 miss). After N cycles of execution during runahead mode, the processor starts checking what percentage of all executed loads are INV in the current runahead mode. If the percentage of INV loads is greater than some threshold T%, the processor initiates exit from runahead mode (We found that good values for N and T are 50 and 90, respectively. Waiting for 50 cycles before deciding whether or not to exit runahead mode reduces the probability of prematurely incorrect predictions). In other words, if too many of the loads executed during runahead mode are INV, this is used as an indication that the current runahead period will not generate useful prefetches, therefore the processor is better off exiting runahead mode. We call this the *INV Load Count* technique.

5.3.3. Coarse-Grain Uselessness Prediction Via Sampling. The previous two approaches (RCST and INV Load Count) aim to predict the usefulness of a runahead period in a fine-grain fashion. Each possible runahead period is predicted as useful or useless. In some benchmarks, especially in bzip2 and mcf, we find that usefulness of runahead periods exhibits more coarse-grain, phase-like behavior. Runahead execution tends to consistently generate or not generate L2 load misses in a large number of consecutive periods. This behavior is due to: (1) the phase behavior in benchmarks [4], where some phases show high memory-level parallelism and others do not, (2) the clustering of L2 misses [2].

To capture the usefulness (or uselessness) of runahead execution over a large number of periods, we propose the use of *sampling-based prediction*. In this mechanism, the processor periodically monitors the total number of L2 load misses generated during N consecutive runahead periods. If this number is less than a threshold T, the processor does not enter runahead mode for the next M cases where an L2-miss load is at the head of the instruction window. Otherwise, the processor enters runahead mode for the next N periods and monitors the number of misses generated in those periods.

[3] An L2 miss caused by a load that is pseudo-retired at least N instructions after the runahead-causing load, where N is the instruction window (reorder buffer) size, cannot be captured by the instruction window. N=128 in our simulations.

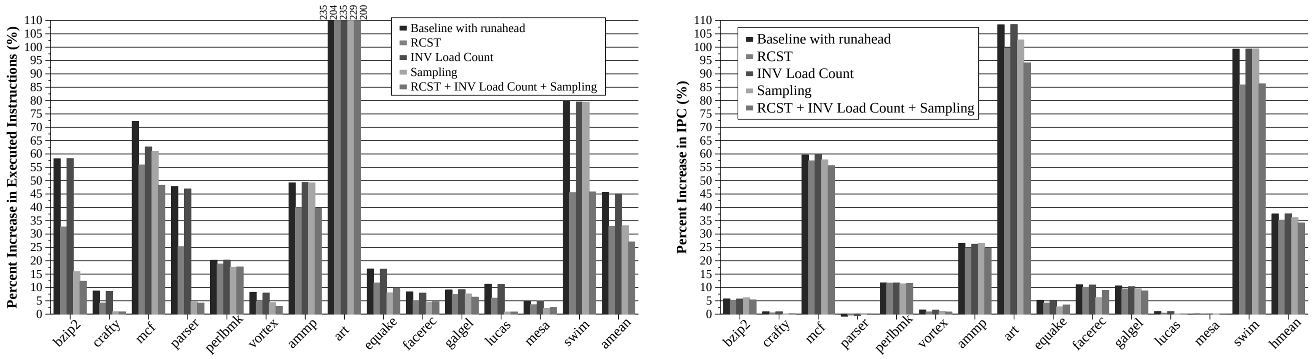

Figure 8. Increase in executed instructions and IPC after eliminating useless runahead periods.

This mechanism uses the number of misses generated in the previous N runahead periods as a predictor for the usefulness of the next M runahead periods. In our simulations, we set N to 100, T to 25, and M to 1000. We did not tune the values of these parameters. It is possible to vary the values of the parameters dynamically to increase the efficiency even more, but a detailed study of parameter tuning or phase detection is out of the scope of this paper.

Figure 8 shows the effect of applying the three *uselessness prediction* techniques individually and together on the increase in executed instructions and IPC. The *RCST* technique increases the efficiency in many INT and FP benchmarks. In contrast, the *INV Load Count* technique increases the efficiency significantly in only one INT benchmark, mcf. In many other benchmarks, load instructions are usually not dependent on other loads and therefore the *INV Load Count* technique does not affect these benchmarks. We expect the *INV Load Count* technique to work better in workloads with significant amount of pointer-chasing code, such as database workloads. The *Sampling* technique results in significant reductions in executed instructions in especially bzip2 and parser, two benchmarks where runahead is very inefficient. All three techniques together increase the efficiency more than each individual technique, implying that the techniques identify different useless runahead periods. On average, all techniques together reduce the increase in executed instructions from 45.6% to 27.1%, while reducing the IPC increase from 37.6% to 34.2%.

5.3.4. Compile-Time Techniques to Eliminate Useless Runahead Periods Caused by a Static Load. Some static load instructions rarely lead to the generation of independent L2 cache misses when they cause entry into runahead mode. For example, in bzip2, one load instruction causes 62,373 entries into runahead mode (57% of all runahead entries), which result in only 561 L2 misses that cannot be captured by the processor's instruction window. If the runahead periods caused by such static loads are useless due to some inherent reason in the program structure or behavior, these load instructions can be designated as *non-runahead loads* (loads which cannot cause entry into runahead) by the compiler after code analysis and/or profiling runs. This section briefly examines improving efficiency using compile-time profiling techniques to identify *non-runahead loads*.

We propose a technique in which the compiler profiles the application program by simulating the execution of the program on a runahead processor. During the profiling run, the compiler keeps track of the number of runahead periods initiated by each static load instruction and the total number of L2 misses generated in the runahead periods initiated by each static load instruction. If the ratio of the number of L2 misses generated divided by the number of runahead periods initiated is less than some threshold T for a load instruction, the compiler marks that load as a *non-runahead load*, using a single bit which is augmented in the load instruction format of the ISA. At run-time, if the processor encounters a *non-runahead load* as an L2-miss instruction at the head of the instruction window, it does not initiate entry into runahead mode. We examined threshold values of 0.1, 0.25, and 0.5, and found that 0.25 yields the best average efficiency value. In general, a larger threshold yields a larger reduction in the number of executed instructions by reducing the number of runahead periods, but it also reduces performance because it results in the elimination of some useful periods.

Figure 9 shows the increase in executed instructions and IPC after using profiling by itself (second and third bars from the left for each benchmark) and in combination with the previously discussed three uselessness prediction techniques (fourth and fifth bars from the left). When profiling is used in combination with the other uselessness prediction techniques, the processor dynamically decides whether or not to enter runahead on a load that is not marked as *non-runahead*. This figure shows that profiling by itself can significantly increase the efficiency of a runahead processor. Also, the input set used for profiling does not significantly affect the results.[4] However, profiling is less effective than the combination of the dynamic techniques. Combining profiling with the three dynamic techniques reduces the increase in executed instructions from 45.6% to 25.7%, while

[4] In Figure 9, "same input set" means that the same input set was used for the profiling run and the simulation run. For the "different input set" case, we used the train input set in the profiling run.

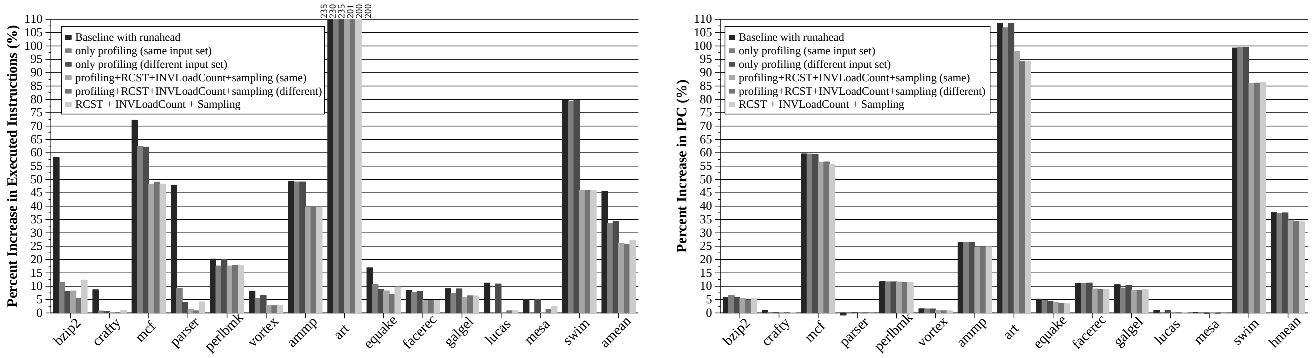

Figure 9. Increase in executed instructions and IPC after using profiling to eliminate useless runahead periods.

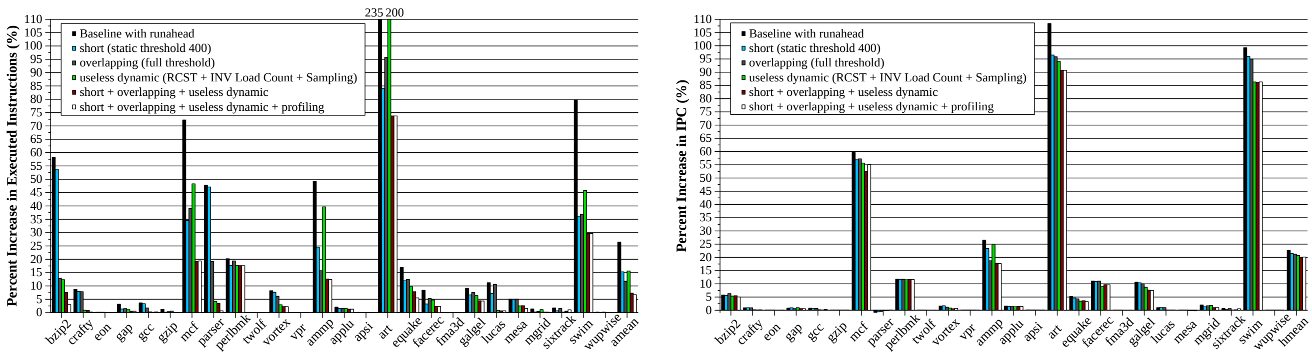

Figure 10. Increase in executed instructions and IPC after using the proposed techniques individually and together.

reducing the IPC increase from 37.6% to 34.3%. These results are better than what can be achieved only with the dynamic uselessness prediction techniques, indicating that there is room for improvement if compile-time information is utilized in combination with dynamic information.

5.4. Combining the Efficiency Techniques

The causes of inefficiency (short, overlapping, and useless runahead periods) are sometimes disjoint from each other and sometimes they are not. To eliminate all causes, we examine the impact of combining our techniques.

Figure 10 shows the increase in executed instructions and IPC after applying the proposed techniques to eliminate short, overlapping, and useless runahead periods individually and together on all SPEC 2000 benchmarks. The rightmost two bars for each benchmark show the effect of using the efficiency-increasing techniques together, with the rightmost bar including the profiling technique. Average increase in number of instructions is minimized when all techniques are applied together rather than when each technique is applied in isolation. This is partly because different benchmarks benefit from different efficiency-increasing techniques and partly because different techniques sometimes eliminate disjoint causes of inefficiency. When all techniques other than profiling are applied, the increase in executed instructions is reduced from 26.5% to 7.3% (6.7% with profiling), whereas the IPC improvement is only reduced from 22.6% to 20.0% (20.1% with profiling).

6. Performance Optimizations for Efficient Runahead Execution

Techniques we have considered so far focused on increasing the efficiency of runahead execution by reducing the number of extra instructions executed, without significantly reducing the performance improvement. Another way to increase efficiency, which is perhaps harder to accomplish, is to increase the performance improvement without significantly increasing or while reducing the number of executed instructions. As the performance improvement of runahead execution is mainly due to the useful L2 misses prefetched during runahead mode [14, 2], it can be increased with optimizations that lead to the discovery of more L2 misses during runahead mode. This section examines optimizations that have the potential to increase efficiency by increasing performance. The three techniques we examine are: (1) turning off the floating point unit during runahead mode, (2) early wake-up of INV instructions, and (3) optimization of the hardware prefetcher update policy during runahead mode.

6.1. Turning Off the Floating Point Unit

FP operate instructions do not contribute to the address computation of load instructions. As runahead execution targets the pre-computation of load addresses, FP operate instructions do not need to be executed during runahead mode. Not executing the FP instructions during runahead mode has two advantages. First, it enables the turning off of the FP unit, including the FP physical register file, during runahead mode, which results in dynamic and static energy savings. Second, it enables the processor to make further progress in the instruction stream during runahead mode, since FP instructions can be dropped before execution, which spares resources for potentially more useful instructions.[5] 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. This case happens rarely in the benchmarks we examined.

Turning off the FP unit may increase the number of instructions executed by a runahead processor, since it allows more instructions to be executed during a runahead period by enabling the processor to make further progress in the instruction stream. On the other hand, this optimization reduces the number of FP instructions executed during runahead mode, which may increase efficiency.

To examine the performance and efficiency impact of turning off the FP unit during runahead mode, we simulate a processor that does not execute the operate and control-flow instructions that source FP registers during runahead mode. An operate or control-flow instruction that sources an FP register and that either has no destination register or has an FP destination register is dropped after being decoded.[6] With these optimizations, FP instructions do not occupy any processor resources in runahead mode after they are decoded. Note that FP loads and stores, whose addresses are generated using INT registers, are executed and are treated as prefetch instructions. Their execution is accomplished in the load/store unit, just like in traditional out-of-order processors. The impact of turning off the FP unit in runahead mode on performance and efficiency is evaluated in the next section.

6.2. Early Wake-up of INV Instructions

If one source operand of an instruction is INV, that instruction will produce an INV result. Therefore, the instruction can be scheduled right away once any source operand is known to be INV, regardless of the readiness of its other source operands. Previous proposals of runahead execution did not take advantage of this property. In [13], an instruction waits until all its source operands become ready before being scheduled, even if the first-arriving source operand is INV. Alternatively, a runahead processor can keep track of the INV status of each source operand of an instruction in the scheduler and wake up the instruction when any of its source operands becomes INV. We call this scheme *early INV wake-up*. This optimization has the potential to improve performance, because an INV instruction can be scheduled before its other source operands become ready. Early wake-up and scheduling will result in the early removal of the INV instruction from the scheduler. Hence, scheduler entries will be spared for valid instructions that can potentially generate prefetches. A disadvantage of this scheme is that it increases the number of executed instructions, which may result in a degradation in efficiency, if the performance improvement is not sufficient.

In previous proposals of runahead execution, INV bits were only present in the physical register file. In contrast, *early INV wake-up* requires INV bits to be present in the wake-up logic, which is possibly on the critical path of the processor. The wake-up logic is extended with one more gate that takes the INV bit into account. Although we evaluate *early INV wake-up* as an optimization, whether or not it is worthwhile to implement in a runahead processor needs to be determined after critical path analysis, which depends on the implementation of the processor.

Figure 11 shows the increase in executed instructions and IPC over the baseline processor when we apply the *FP turn-off* and *early INV wake-up* optimizations individually and together to the runahead processor. Turning off the FP unit increases the average IPC improvement of the runahead processor from 22.6% to 24%. *Early INV wakeup* increases the IPC improvement of the runahead processor to 23.4%. Turning off the FP unit reduces the increase in executed instructions from 26.5% to 25.5%, on average. Both of the optimizations are more effective on the FP benchmarks. Integer benchmarks do not have many FP instructions, therefore turning off the FP unit does not help their performance. *Early INV wake-up* helps benchmarks where at least one other source operand of an instruction is produced later than the first INV source operand. FP benchmarks show this characteristic more than the integer benchmarks since they have more frequent data cache misses and long-latency FP instructions. Data cache misses and FP instructions are frequently the causes of the late-produced sources. Performing both optimizations together adds little gain to the performance improvement obtained by only turning off the FP unit. This is because turning off the FP unit reduces the latency with which late-arriving operands of an instruction are produced and therefore reduces the opportunities for early INV wake-up. These results suggest that turning off the FP unit during runahead mode is a valuable optimization that both increases performance and saves energy in a runahead processor. In contrast, *early INV wake-up* is not worthwhile to implement since its performance benefit is more efficiently captured by turning off the FP unit and its implementation increases the complexity of the scheduling logic.

[5] In fact, FP instructions can be dropped immediately after fetch during runahead mode, if extra decode information is stored in the instruction cache indicating whether or not an instruction is an FP instruction.

[6] An FTOI instruction which moves the value in an FP register to an INT register is not dropped. It is immediately made ready to be scheduled, once it is placed into the instruction window. After it is scheduled, it marks its destination register as INV and it is considered to be an INV instruction.

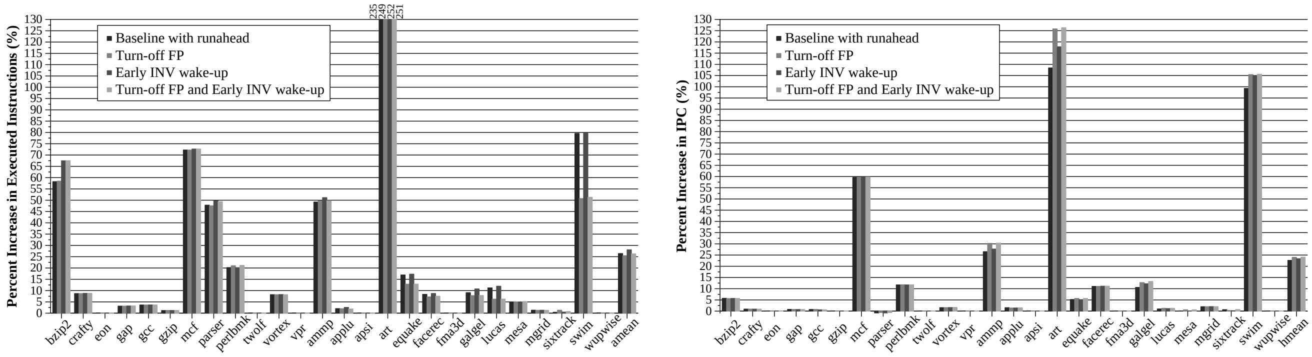

Figure 11. Increase in executed instructions and IPC after turning off the FP unit and using early INV wake-up.

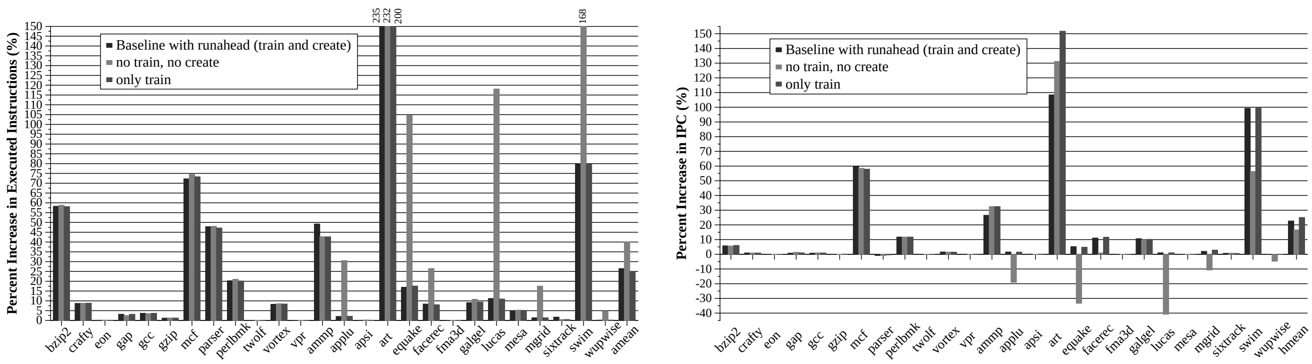

Figure 12. Increase in executed instructions and IPC based on prefetcher training policy during runahead mode.

6.3. Optimizing the Prefetcher Update Policy

One of the potential benefits of runahead execution is that the 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 and hence improve performance. Update of the prefetcher during runahead mode may also create new prefetch streams, which can result in performance improvement. 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. We experiment with three different policies to determine the impact of prefetcher update policy on the performance and efficiency of a runahead processor.

Our baseline runahead implementation assumes no change to the prefetcher hardware. Just like in normal mode, L2 accesses during runahead mode train the existing streams and L2 misses during runahead mode create new streams (*train and create* policy). We also evaluate a policy where the prefetcher is turned off during runahead mode; i.e., runahead L2 accesses do not train the stream buffers and runahead L2 misses do not create new streams (*no train, no create* policy). The last policy we examine allows the training of existing streams, but disables the creation of new streams during runahead mode (*only train* policy).

Figure 12 shows the increase in executed instructions and IPC over the baseline processor with the three update policies. On average, the *only train* policy performs best with a 25% IPC improvement while also resulting in the smallest (24.7%) increase in executed instructions. Hence, the *only train* policy increases both the efficiency and the performance of the runahead processor. This suggests that creation of new streams during runahead mode is detrimental for performance and efficiency. In art and ammp, creating new streams during runahead mode reduces performance compared to only training the existing streams, due to the low accuracy of the prefetcher in these two benchmarks. For art and ammp, if new streams are created during runahead mode, they usually generate useless prefetches which cause cache pollution and resource contention with the more accurate runahead memory requests. Cache pollution caused by the new streams results in more L2 misses during normal mode (hence, more entries into runahead mode), which do not exist with the *only train* policy. That is why the increase in executed instructions is smaller with the *only train* policy.

The *no train, no create* policy reduces the performance improvement of runahead on applu, equake, facerec, lucas, mgrid, and swim significantly. It also increases the number of instructions executed in these benchmarks, because it increases the number of L2 cache misses, which results in increased number of entries into runahead mode that are not very beneficial. In these benchmarks, the main benefit of useful runahead execution periods comes from increasing the timeliness of the prefetches generated by the hardware

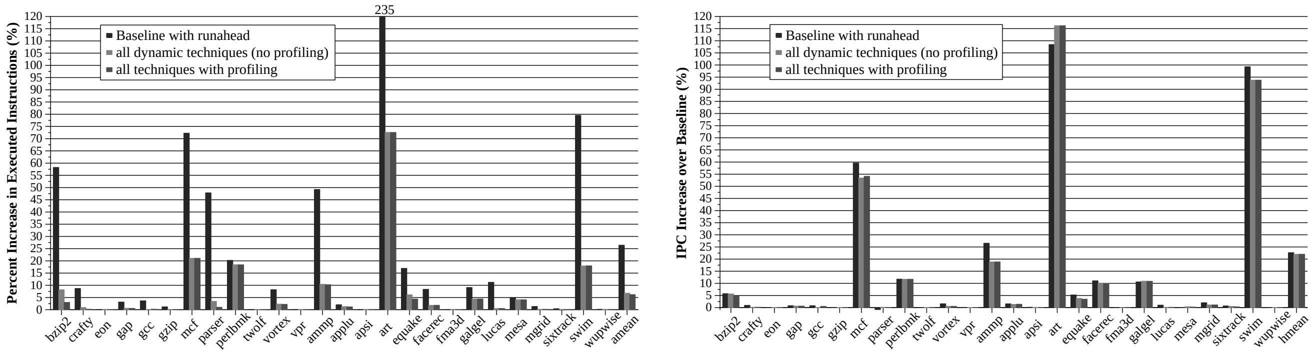

Figure 13. Increase in executed instructions and IPC after all efficiency and performance optimizations.

prefetcher. If runahead mode does not update the prefetcher, it results in little benefit and pipeline flushes at the end of these useless runahead periods reduce the IPC significantly.

7. Putting It All Together

We examine the overall effect of the efficiency and performance enhancement techniques proposed in Sections 5 and 6. We consider the following efficiency and performance optimizations together: *static threshold 400* (Section 5.1), *full threshold* (Section 5.2), *RCST*, *INV Load Count*, and *Sampling* (Section 5.3), turning off the FP unit (Section 6.1), and the *only train* policy (Section 6.3). We also evaluate the impact of using profiling (Section 5.3) in addition to these dynamic techniques.

Figure 13 shows the increase in executed instructions and IPC over the baseline processor. Applying the proposed techniques significantly reduces the average increase in executed instructions in a runahead processor, from 26.5% to 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.

7.1. Effect of Main Memory Latency

We examine the effectiveness of using the proposed dynamic techniques with different memory latencies. The left graph in Figure 14 shows the increase in IPC over the baseline processor if runahead execution is employed with or without all the dynamic techniques. The data shown are averaged over INT and FP benchmarks. The right graph in Figure 14 shows the increase in executed instructions. We used a *static threshold* of 50, 200, 400, 650, and 850 cycles for main memory latencies of 100, 300, 500, 700, and 900 cycles respectively, in order to eliminate short runahead periods. Other parameters used in the techniques are the same as described in the previous sections. As memory latency increases, both the increase in IPC and the increase in executed instructions due to runahead execution increase. For almost all memory latencies, employing the proposed dynamic techniques increases the average IPC improvement on the FP benchmarks while only slightly reducing the IPC improvement on the INT benchmarks. For all memory latencies, employing the proposed dynamic techniques significantly reduces the increase in executed instructions. We conclude that the proposed techniques are effective for a wide range of memory latencies, even without tuning the parameters used in the techniques.

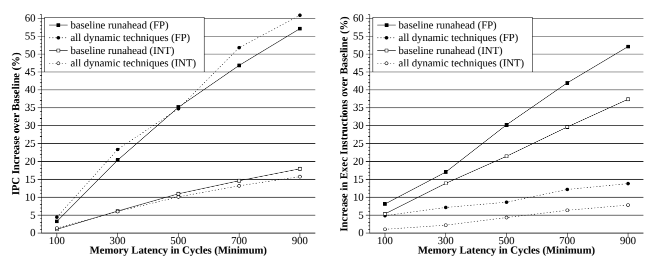

Figure 14. Increase in IPC and executed instructions with and without the proposed techniques (no profiling).

8. Related Work

Several previous papers evaluated the use of runahead execution for improving the latency tolerance of microprocessors. None of the previous work we are aware of addressed the efficiency issues in runahead execution.

Dundas and Mudge [5] first proposed runahead execution as a means to improve the performance of an in-order processor. They described the implementation details on an in-order processor.

Mutlu et al. [13] proposed runahead execution as a means to increase the main memory latency tolerance of an out-of-order processor. They provided the implementation details on an out-of-order processor and evaluated some performance enhancements such as the "runahead cache." A few tradeoffs in runahead entry/exit policies and branch predictor update policy during runahead were mentioned. Our paper examines in detail the tradeoff between the performance improvement and the number of extra instructions executed in runahead execution.

Mutlu et al. [14] also found that the major benefit of runahead execution is the prefetching of independent L2 data misses in parallel with the runahead-causing L2 miss. We make use of this observation and utilize the number of L2 misses generated in runahead mode as a predictor for the usefulness of runahead execution.

Chou et al. [2] examined runahead execution as a technique to enhance memory-level parallelism. They demonstrated that runahead execution is very effective in improving memory-level parallelism, because it prevents the instruction and scheduling windows, along with serializing instructions from being bottlenecks. Our paper shows that runahead execution can be made even more effective with simple optimizations.

Iacobovici et al. [8] evaluated the prefetch accuracy and coverage of runahead execution. They found that runahead execution interacts positively with their multi-stride hardware data prefetcher. Positive interaction of runahead execution and stream-based data prefetchers was also reported in [13]. Neither of these papers analyzed the effect of prefetcher update policy on the performance of a runahead processor. We show that the positive interaction between runahead and prefetchers can be increased by optimizing the prefetcher update policy during runahead mode.

Manne et al. [11] proposed *pipeline gating* to reduce the extra work performed in the processor due to branch mispredictions. Their scheme gates the pipeline stages if the number of low-confidence branches fetched exceeds a threshold. Our proposals have the same goal with Manne et al.'s, i.e., to reduce the speculatively executed instructions in order to reduce the energy consumption of the processor while retaining the performance benefit of speculation. However, the kind of speculation we target, runahead execution, is different from and orthogonal to branch prediction.

The efficiency of runahead execution can potentially be increased by eliminating the re-execution of instructions via reuse [12]. However, even an ideal reuse mechanism does not significantly improve performance [12] and it likely has significant hardware cost and complexity, which may offset the energy reduction due to improved efficiency.

9. Conclusion

This paper proposes simple techniques to increase the efficiency of a runahead processor while retaining almost all of its performance benefit. First, we show that a runahead processor executes significantly more instructions than an out-of-order processor, sometimes without providing any performance benefit. Hence, runahead execution is not efficient for some benchmarks. We identify three causes of inefficiency in a runahead processor: short, overlapping, and useless runahead periods. We propose simple techniques that reduce these causes of inefficiency without significantly reducing performance.

In an attempt to increase efficiency by increasing performance and reducing or not significantly increasing the number of executed instructions, this paper examines several performance optimizations that can be employed during runahead execution. We show that turning off the FP unit is a simple optimization that increases the performance of a runahead processor, while reducing the number of executed instructions and saving energy during runahead mode. We also show that optimizing the prefetcher update policy during runahead mode significantly increases both performance and efficiency.

The combination of all the effective techniques proposed in this paper reduces the increase in the number of instructions executed due to runahead execution from 26.5% to 6.2%, on average, without significantly affecting the performance improvement provided by runahead execution. We believe many of our techniques are applicable to other methods of speculative pre-execution, e.g. [1, 3]. Future work includes the extension of our techniques to other forms of pre-execution.

Acknowledgments

We thank Mike Butler, Nhon Quach, Jared Stark, Santhosh Srinath, and other members of the HPS research group for their helpful comments on drafts of this paper. Onur Mutlu is supported by an Intel fellowship.

References

[1] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt. Simultaneous subordinate microthreading (SSMT). In *ISCA-26*, 1999.

[2] Y. Chou, B. Fahs, and S. Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. In *ISCA-31*, 2004.

[3] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen. Dynamic speculative precomputation. In *MICRO-34*, 2001.

[4] P. Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, (1):64–84, Jan. 1980.

[5] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *ICS-11*, 1997.

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

[7] S. Gochman, R. Ronen, I. Anati, A. Berkovits, T. Kurts, A. Naveh, A. Saeed, Z. Sperber, and R. C. Valentine. The Intel Pentium M processor: Microarchitecture and performance. *Intel Technology Journal*, 7(2):21–36, May 2003.

[8] S. Iacobovici, L. Spracklen, S. Kadambi, Y. Chou, and S. G. Abraham. Effective stream-based and execution-based data prefetching. In *ICS-18*, 2004.

[9] A. KleinOsowski and D. J. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 1, June 2002.

[10] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *ISCA-8*, 1981.

[11] S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: Speculation control for energy reduction. In *ISCA-25*, 1998.

[12] O. Mutlu, H. Kim, J. Stark, and Y. N. Patt. On reusing the results of pre-executed instructions in a runahead execution processor. *Computer Architecture Letters*, 4, Jan. 2005.

[13] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *HPCA-9*, 2003.

[14] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An effective alternative to large instruction windows. *IEEE Micro*, 23(6):20–25, Nov./Dec. 2003.

[15] J. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Technical White Paper*, Oct. 2001.