Efficient Runahead Execution Processors

Onur Mutlu

High Performance Systems Group
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, Texas 78712-0240

TR-HPS-2006-007
July 2006

This page is intentionally left blank.

The Dissertation Committee for Onur Mutlu
certifies that this is the approved version of the following dissertation:

Efficient Runahead Execution Processors

Committee:

Yale N. Patt, Supervisor

Craig M. Chase

Nur A. Touba

Derek Chiou

Michael C. Shebanow

Efficient Runahead Execution Processors

by

Onur Mutlu, B.S.; B.S.E.; M.S.E.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2006

Dedicated to my loving parents, Hikmet and Nevzat Mutlu, and my sister Miray Mutlu

Acknowledgments

Many people and organizations have contributed to this dissertation, intellectually, motivationally, or otherwise financially. This is my attempt to acknowledge their contributions.

First of all, I thank my advisor, Yale Patt, for providing me with the freedom and resources to do high-quality research, for being a caring teacher, and also for teaching me the fundamentals of computing in EECS 100 as well as valuable lessons in real-life areas beyond computing.

My life as a graduate student would have been very short and unproductive, had it not been for Hyesoon Kim. Her technical insights and creativity, analytical and questioning skills, high standards for research, and continuous encouragement made the contents of this dissertation much stronger and clearer. Her presence and support made even the Central Texas climate feel refreshing.

Very special thanks to David Armstrong, who provided me with the inspiration to write and publish, both technically and otherwise, at a time when it was difficult for me to do so. To me, he is an example of fairness, tolerance, and open-mindedness in an all-too-unfair world.

Many members of the HPS Research Group have contributed to this dissertation and my life in the past six years. I especially thank:

- José Joao and Ángeles Juarans Font, for our memorable trips and delicious barbecues, for their hearty friendship, and for proofreading this dissertation.

- Chang Joo Lee, for always being a source of fun and optimism, and for bearing with my humor and complaints.
- Moinuddin Qureshi, for many interesting technical discussions and being my roommate during the last year.
- Santhosh Srinath, for being a very good and cheerful friend.
- Francis Tseng, for his helpfulness and hard work in maintaining our group's network and simulation infrastructure.
- Danny Lynch, Aater Suleman, Mary Brown, Sangwook Peter Kim, Kameswar Subramaniam, David Thompson, Robert Chappell, and Paul Racunas for their contributions to our group's simulation infrastructure and for their friendship.

Many people in computer industry have helped shape my career and provided invaluable feedback on my research. I especially thank Jared Stark and Eric Sprangle for their mentorship and assistance in developing research ideas. They strongly influenced my first years as a researcher and I am honored to have them as co-authors in my first scholarly paper and patent. Stephan Meier, Chris Wilkerson, Konrad Lai, Mike Butler, Nhon Quach, Mike Fertig, and Chuck Moore provided valuable comments and suggestions on my research activities and directions.

I would like to thank Derek Chiou, Nur Touba, Michael Shebanow, and Craig Chase for serving on my dissertation committee. Special thanks to Derek for he has always been accessible, encouraging, and supportive.

Throughout graduate school, I have had generous financial support from several organizations. I would like to thank the University of Texas Graduate School for the University Continuing Fellowship, and Intel Corporation for the Intel Foundation PhD Fellowship. I also thank the University Co-op for awarding me the George H. Mitchell Award for

Excellence in Graduate Research. Many thanks to Intel Corporation and Advanced Micro Devices for providing me with summer internships, which were invaluable for my research career.

Finally, I cannot express with words my indebtedness to my parents, Hikmet and Nevzat Mutlu, and my little sister Miray Mutlu, for giving me their unconditional love and support at every step I have taken in my life. Even though they will likely not understand much of it, this dissertation would be meaningless without them.

Onur Mutlu
May 2006, Austin, TX

Efficient Runahead Execution Processors

Publication No. _____

Onur Mutlu, Ph.D.
The University of Texas at Austin, 2006

Supervisor: Yale N. Patt

High-performance processors tolerate latency using out-of-order execution. Unfortunately, today's processors are facing memory latencies in the order of hundreds of cycles. To tolerate such long latencies, out-of-order execution requires an instruction window that is unreasonably large, in terms of design complexity, hardware cost, and power consumption. Therefore, current processors spend most of their execution time stalling and waiting for long-latency cache misses to return from main memory. And, the problem is getting worse because memory latencies are increasing in terms of processor cycles.

The runahead execution paradigm improves the memory latency tolerance of an out-of-order execution processor by performing potentially useful execution while a long-latency cache miss is in progress. Runahead execution unblocks the instruction window blocked by a long-latency cache miss allowing the processor to execute far ahead in the program path. This results in other long-latency cache misses to be discovered and their data to be prefetched into caches long before it is needed.

This dissertation presents the runahead execution paradigm and its implementation on an out-of-order execution processor that employs state-of-the-art hardware prefetching

techniques. It is shown that runahead execution on a 128-entry instruction window achieves the performance of a processor with three times the instruction window size for a current, 500-cycle memory latency. For a near-future 1000-cycle memory latency, it is shown that runahead execution on a 128-entry window achieves the performance of a conventional processor with eight times the instruction window size, without requiring a significant increase in hardware cost and complexity.

This dissertation also examines and provides solutions to two major limitations of runahead execution: its energy inefficiency and its inability to parallelize dependent cache misses. Simple and effective techniques are proposed to increase the efficiency of runahead execution by reducing the extra instructions executed without affecting the performance improvement. An efficient runahead execution processor employing these techniques executes only 6.2% more instructions than a conventional out-of-order execution processor but achieves 22.1% higher Instructions Per Cycle (IPC) performance.

Finally, this dissertation proposes a new technique, called address-value delta (AVD) prediction, that predicts the values of pointer load instructions encountered in runahead execution in order to enable the parallelization of dependent cache misses using runahead execution. It is shown that a simple 16-entry AVD predictor improves the performance of a baseline runahead execution processor by 14.3% on a set of pointer-intensive applications, while it also reduces the executed instructions by 15.5%. An analysis of the high-level programming constructs that result in AVD-predictable load instructions is provided. Based on this analysis, hardware and software optimizations are proposed to increase the benefits of AVD prediction.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 The Problem: Tolerating Long Main Memory Latencies

High-performance processors execute instructions out of program order to tolerate long latencies and extract instruction-level parallelism. An out-of-order execution processor tolerates latencies by moving the long-latency operation "out of the way" of the operations that come later in the instruction stream and that do not depend on it. However, these processors retire instructions in program order to support precise exceptions [103]. To accomplish the out-of-order execution but in-order retirement of instructions, the processor buffers the decoded but not-yet-retired instructions in a hardware buffer called the *instruction window* [90].

If the execution of a long-latency instruction is not complete, that instruction and instructions following it in the sequential instruction stream cannot be retired. Incoming instructions fill the instruction window if the window is not large enough. Once the window becomes full, the processor cannot place new instructions into the window and stalls. This resulting stall is called a full-window stall. It prevents the processor from finding independent instructions to execute in order to tolerate the long latency. The processor's latency tolerance can be increased by increasing the size of the instruction window such that no full-window stall occurs when a long-latency instruction blocks retirement. However, this is a challenging task due to the design complexity, verification difficulty, and increased power consumption of a large instruction window [28, 46]. In fact, Palacharla et al. showed that the complexity and delay of many hardware structures on the critical path

increase quadratically with instruction window size [87].

Unfortunately, today's main memory latencies are so long that out-of-order processors require very large instruction windows to tolerate them. A cache miss to main memory costs about 128 cycles on an Alpha 21264 [122] and 330 cycles on a Pentium-4-like processor [108]. As processor and system designers continue to push for smaller cycle times and larger memory modules and memory designers continue to push for higher bandwidth and capacity, main memory latencies will continue to increase in terms of processor cycles [124, 122].

Figure 1.1 shows that a current x86 processor, modeled after the Pentium 4, with a 128-entry instruction window, a 512 KB L2 cache, and an aggressive hardware prefetcher spends 68% of its execution cycles in full-window stalls (Processor 1). If the L2 cache is made perfect (Processor 2), the processor wastes only 30% of its cycles in full-window stalls, indicating that long-latency L2 misses are the single largest cause of full-window stalls in Processor 1. On the other hand, Processor 3, which has a 2048-entry instruction window and a 512 KB L2 cache spends only 33% of its cycles in full-window stalls. Figure 1.1 also shows that 49% IPC improvement is possible with a 2048-entry instruction window, 82% IPC improvement is possible with an infinite-entry instruction window, and 120% IPC improvement is possible if all L2 cache misses are eliminated. Hence, a processor with a large instruction window tolerates the main memory latency much better than a processor with a small instruction window. Furthermore, significant potential exists to improve the performance of the state-of-the-art out-of-order processors by improving their tolerance to long main memory latencies.

As energy/power consumption has already become a limiting constraint in the design of high-performance processors [45], simple power- and area-efficient memory latency tolerance techniques are especially desirable. Our main research objective is to improve the memory latency tolerance of high-performance out-of-order execution processors

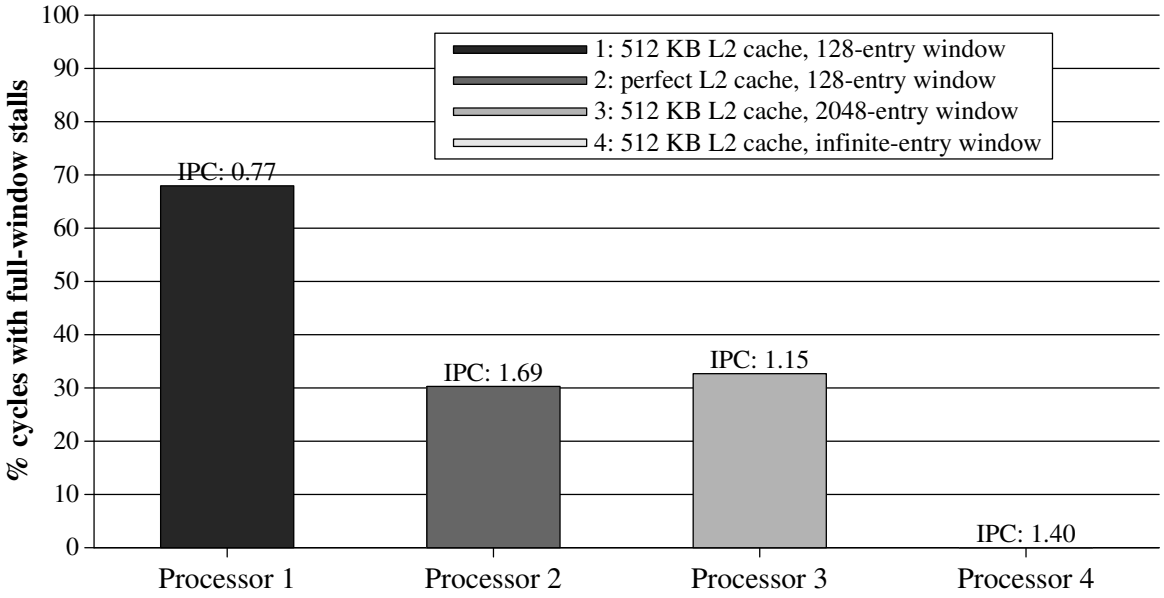


Figure 1.1: Percentage of execution cycles with full-window stalls. Data is averaged over a wide variety of memory-intensive benchmarks. Minimum memory latency is 495 cycles and the processor employs an aggressive stream-based hardware prefetcher. Section 4.1 describes the baseline processor model and the simulated benchmarks.

with simple, implementable, and energy-efficient mechanisms without resorting to building complex and power-hungry large instruction windows. To this end, this dissertation proposes and evaluates *efficient runahead execution*.

1.2 The Solution: Efficient Runahead Execution

The runahead execution paradigm tolerates long main memory latencies by performing useful execution rather than stalling the processor when a long-latency cache miss occurs.[1] A runahead execution processor switches to a special *runahead mode* when the

[1] In this dissertation, we refer to a cache miss to main memory as a *long-latency cache miss* or an *L2 cache miss*, assuming a two-level cache hierarchy. The concepts presented are applicable to processors with more than two caches in their memory hierarchy as well. In that case, a cache miss at the cache level furthest from the processor core is considered a long-latency cache miss.

oldest instruction in the processor incurs a long-latency cache miss, and speculatively executes the program during the cycles that would otherwise be spent idle as full-window stall cycles in a conventional processor. The purpose of this speculative runahead mode execution is to discover and service in parallel long-latency cache misses that cannot be discovered and serviced had the processor been stalled because of a long-latency cache miss. A runahead execution processor tolerates a long-latency cache miss by servicing it in parallel with other long-latency cache misses discovered through speculative runahead mode execution.

The runahead execution paradigm does not require significant hardware cost or complexity because the performed runahead mode execution does not require the buffering of a large number of instructions following a long-latency cache miss. As instructions are speculatively executed in runahead mode, they are removed from the processor's instruction window. Long-latency cache misses and their dependents are removed from the instruction window without waiting for the data to return from main memory. This creates space in the instruction window for the execution of younger instructions and thus eliminates the need for maintaining a large instruction window. When the long-latency cache miss that caused the processor to switch to runahead mode is complete, the runahead processor switches back to *normal execution mode* and re-executes those instructions that were speculatively executed in runahead mode in order to correctly update the architectural program state with their results.

The runahead execution paradigm results in the execution of more instructions than a conventional processor because runahead execution relies on the speculative execution of instructions during otherwise-idle cycles. If uncontrolled, runahead mode execution can result in significant increases in dynamic energy. To avoid this problem, while still preserving the memory latency tolerance benefits of runahead execution, this dissertation proposes and evaluates hardware and software techniques to predict when speculative runa-

head mode execution will be useful. An *efficient runahead execution* processor uses these techniques to perform runahead mode execution only when it is predicted to increase the processor's memory latency tolerance.

1.3 Thesis Statement

Efficient runahead execution is a cost- and complexity-effective microarchitectural technique that can tolerate long main memory latencies without requiring unreasonably large, slow, complex, and power-hungry hardware structures or significant increases in processor complexity and power consumption.

1.4 Contributions

This dissertation makes the following major contributions:

- This dissertation presents the runahead execution paradigm for out-of-order execution processors. Runahead execution is a microarchitectural technique that improves a conventional processor's tolerance to long main memory latencies. This dissertation presents the operation of runahead execution in the context of high performance out-of-order execution processors and describes runahead execution's advantages, disadvantages, and limitations.

- This dissertation presents a cost- and complexity-effective implementation of runahead execution in an aggressive high performance processor and evaluates the design tradeoffs in the microarchitecture of a runahead execution processor. It demonstrates that a conventional processor augmented with runahead execution can surpass the performance of a conventional processor with at least three times the instruction window size. The results presented in this dissertation show that runahead execution is a

simple and cost-efficient alternative to building large instruction windows to tolerate long memory latencies.

- This dissertation identifies the inefficiency of runahead execution as one of its limitations and analyzes the causes of inefficiency in a runahead processor. It defines a new efficiency metric to evaluate the energy-efficiency of runahead execution. It presents new microarchitectural and compiler techniques that significantly improve runahead execution's energy efficiency. These techniques reduce the number of speculatively executed instructions in a runahead processor while maintaining and sometimes even increasing the performance improvement provided by runahead execution.

- This dissertation identifies *dependent cache misses* as another limitation of runahead execution and proposes a new technique, *address-value delta (AVD) prediction*, that overcomes this limitation. AVD prediction allows the parallelization of dependent cache misses in a runahead processor by predicting the load values that lead to the generation of other load instruction addresses. This dissertation evaluates AVD prediction on pointer-intensive benchmarks and shows that a simple, low-cost AVD predictor improves the latency tolerance of both runahead execution and conventional processors to dependent cache misses. Furthermore, this dissertation analyzes the high-level programming constructs that result in AVD-predictable load instructions and evaluates hardware and software optimizations that increase the effectiveness of AVD prediction.

1.5 Dissertation Organization

This dissertation is organized into seven chapters. Chapter 2 presents the runahead execution paradigm, including its operation principles and its implementation on a high performance processor. Chapter 3 describes related work in handling long memory laten-

cies. Chapter 4 presents the performance evaluation of the baseline runahead execution mechanism on both x86 and Alpha ISA processors and evaluates the design tradeoffs in runahead execution. Chapter 5 describes the inefficiency problem in runahead execution and presents techniques that make a runahead execution processor more efficient. Chapter 6 describes the problem of dependent cache misses and presents address-value delta (AVD) prediction as a solution to this problem. Chapter 7 provides conclusions, a summary of the key results and insights presented in this dissertation, and suggestions for future research in runahead execution.

Chapter 2

The Runahead Execution Paradigm

2.1 The Basic Idea

The runahead execution paradigm provides an alternative to building large instruction windows to tolerate long-latency operations. Instead of moving the long-latency operation "out of the way" of younger independent operations (as a conventional out-of-order execution processor does), which requires buffering the operation and the instructions following it in the instruction window, runahead execution on an out-of-order execution processor tosses the long-latency operation out of the instruction window. This eliminates the need for buffering the long-latency operation and the instructions following it in large hardware structures (such as large schedulers, register files, load/store buffers and reorder buffers), and creates space in the instruction window to speculatively process operations independent of the long-latency operation.

When the instruction window is blocked by the long-latency operation, the state of the architectural register file is checkpointed. The processor then enters "runahead mode." It distributes a bogus result for the blocking operation and tosses it out of the instruction window. The instructions following the blocking operation are fetched, executed, and pseudo-retired from the instruction window. By pseudo-retire, we mean that the instructions are retired as in the conventional sense, except that they do not update architectural state. When the blocking operation completes, the processor re-enters "normal mode." It restores the checkpointed state and re-fetches and re-executes instructions starting with the blocking operation.

Runahead execution's benefit comes from transforming a small instruction window which is blocked by long-latency operations into a non-blocking window, giving it the performance of a much larger window. The instructions fetched and executed during runahead mode create very accurate prefetches for the data and instruction caches. These benefits come at a modest hardware cost which we will describe later.

This chapter first analyzes the memory latency tolerance of out-of-order execution processors to motivate the concept of runahead execution. An analysis of the impact of the scheduling window, the instruction window, and the memory latency on the performance of a current processor is provided. The chapter then describes the detailed operation of runahead execution and its implementation on a modern out-of-order processor.

2.2 Out-of-Order Execution and Memory Latency Tolerance

2.2.1 Instruction and Scheduling Windows

Out-of-order execution can tolerate cache misses better than in-order execution by scheduling and executing operations that are independent of the cache misses while the cache misses are being serviced. An out-of-order execution machine accomplishes this using two windows: the instruction window and the scheduling window. The instruction window holds all the instructions that have been decoded but not yet committed to the architectural state. Its main purpose is to guarantee in-order retirement of instructions in order to support precise exceptions. The scheduling window holds a subset of the instructions in the instruction window. This subset consists of instructions that have been decoded but not yet scheduled for execution. The main function of the scheduling window is to search its instructions each cycle for those that are ready to execute and to schedule them to be executed in dataflow order.

A long-latency operation blocks the instruction window until the operation is com-

pleted. Although later instructions may have completed execution, they cannot retire from the instruction window. If the latency of the operation is long enough and the instruction window is not large enough, instructions pile up in the instruction window and the window becomes full. The machine then stalls and stops making forward progress. Note that the machine can still fetch and buffer instructions during a full-window stall, but it cannot decode, schedule, execute, and retire them.

A long-latency operation may also block the scheduling window if the scheduling window is not large enough to hold all the instructions dependent on the long-latency operation. In this case, the scheduling window becomes full. Then the machine stalls and stops making forward progress even though the instruction window may have available space.

2.2.2 Main-Memory Latency Tolerance of Out-of-Order Execution

In this section, we show that an idealized version of a current out-of-order execution machine spends most of its time stalling, mostly waiting for memory. Furthermore, we show that the instruction window, not the scheduling window, is the major bottleneck limiting a current out-of-order processor's tolerance to long main memory latencies.

We model different processors with a 128 or a 2048-entry instruction window. The scheduling window is either small (48 entries) or large (the same size as the instruction window). All other machine buffers are set to either 128 or 2048 entries so they do not create bottlenecks. The fetch engine is ideal in that it never suffers from cache misses and always supplies a fetch-width's worth of instructions every cycle. Thus fetch never stalls. However, the fetch engine does use a real branch predictor. The other machine parameters are the same as those of the current baseline—which is based on the Intel Pentium 4 processor—and are shown in Table 4.2.

Figure 2.1 shows the percentage of cycles the instruction window is stalled for seven different machines. The number on top of each bar is the IPC performance (in

micro-operations retired per cycle) of the respective machine. The data is averaged over all benchmarks simulated (see Section 4.1.1).

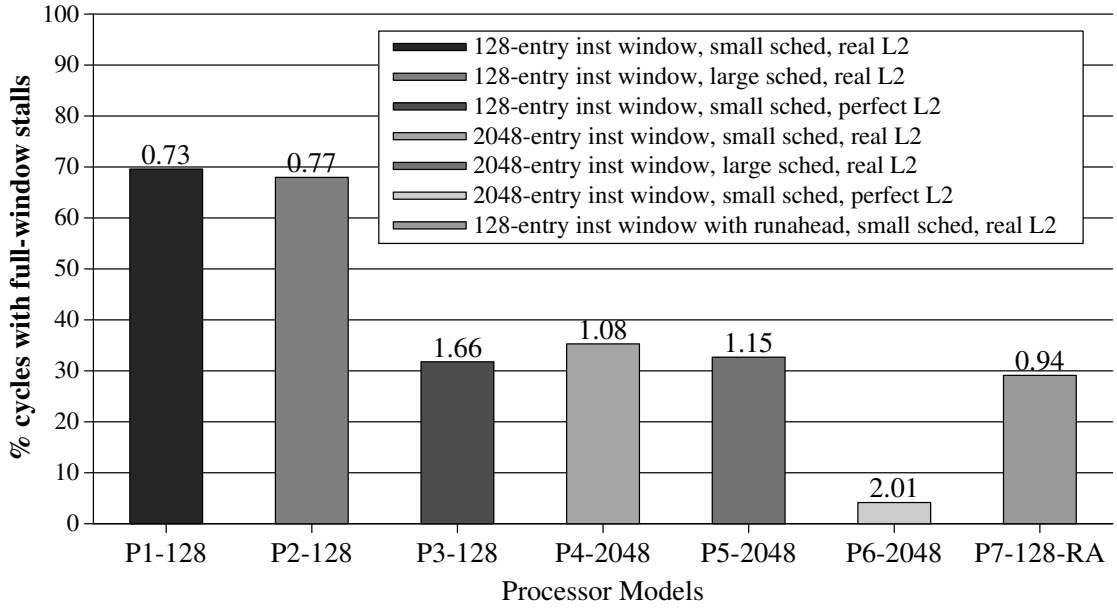


Figure 2.1: Percentage of cycles with full instruction window stalls for processors with small and large scheduling or instruction windows. The number on top of each bar is the IPC of the corresponding machine. Section 4.1 describes the baseline processor model and the simulated benchmarks.

The machine with a 128-entry instruction window, a small scheduling window and a real L2 cache (P1-128) spends 70% of its cycles in full instruction window stalls where no forward progress is made on the running application program. If the scheduling window is removed from being a bottleneck by making the scheduling window size as large as the instruction window size (P2-128), the instruction window still remains a bottleneck with 68% of all cycles spent in full window stalls. Note that the IPC increases only slightly (from 0.73 to 0.77) when the scheduling window is removed from being a bottleneck. Hence, the scheduling window is not the primary bottleneck in a 128-entry-window processor. If instead the main memory latency is removed from being a bottleneck by making the L2 cache perfect (P3-128), the machine only wastes 32% of its cycles in full window stalls. Thus

most of the stalls are due to main memory latency. Eliminating this latency by eliminating all L2 cache misses increases the IPC by 143%.

A machine with a 2048-entry instruction window and a real L2 cache (P4-2048 or P5-2048) is able to tolerate main memory latency much better than the machines with 128-entry instruction windows and real L2 caches. Its percentage of full window stalls is similar to that of the machine with a 128-entry instruction window and a perfect L2 cache (P3-128). However, its IPC is not nearly as high because L2 cache misses increase the number of cycles in which no instructions are retired. L2 cache misses are still not free even on a machine with a 2048-entry instruction window and their latency still adversely impacts the execution time of the program. Note that the scheduling window is still not a major performance bottleneck on a machine with a 2048-entry instruction window. The machine with a 2048-entry instruction window and a perfect L2 cache (P6-2048) is shown for reference. It has the highest IPC and the smallest percentage of full-window stall cycles.

2.2.3 Why Runahead Execution: Providing Useful Work to the Processor During an L2 Cache Miss

A conventional out-of-order processor performs no useful work while the instruction window is stalled waiting for an L2 cache miss to be serviced from main memory. The purpose of runahead execution is to utilize idle cycles that are wasted due to full-window stalls for useful speculative execution of the program, thereby improving the main memory latency tolerance of a small instruction window without increasing the size of the instruction window. The premise is that this non-blocking mechanism lets the processor fetch and execute many more instructions than the instruction window normally permits. If this is not the case (i.e., if a processor's instruction window is already large enough to buffer instructions during an L2 cache miss), runahead execution would provide no performance benefit over conventional out-of-order execution.

We would expect the full-window stall percentage of a 128-entry window processor with runahead execution to be similar to the stall percentage of a 128-entry window processor with a perfect L2 cache (P3-128 in Figure 2.1), since runahead execution does not stall the processor due to L2 cache misses. We would also expect the IPC performance of a 128-entry window processor with runahead execution to be better than that of a 128-entry window processor with a real L2 cache (P1-128) and close to that of a 2048-entry window processor with a real L2 cache (P4-2048), As shown in the rightmost bar in Figure 2.1 (P7-128-RA), runahead execution eliminates almost all of the full-window stalls due to L2 cache misses on a 128-entry window machine and approximates the IPC performance of a 2048-entry window processor. A detailed performance analysis of runahead execution is provided in Chapter 4.

2.3 Operation of Runahead Execution

Figure 2.2 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, it makes no forward progress 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 useless long STALL periods due to L2 cache misses, as shown in Figure 2.2(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 2.2(b). When the processor detects that the oldest instruction is waiting for

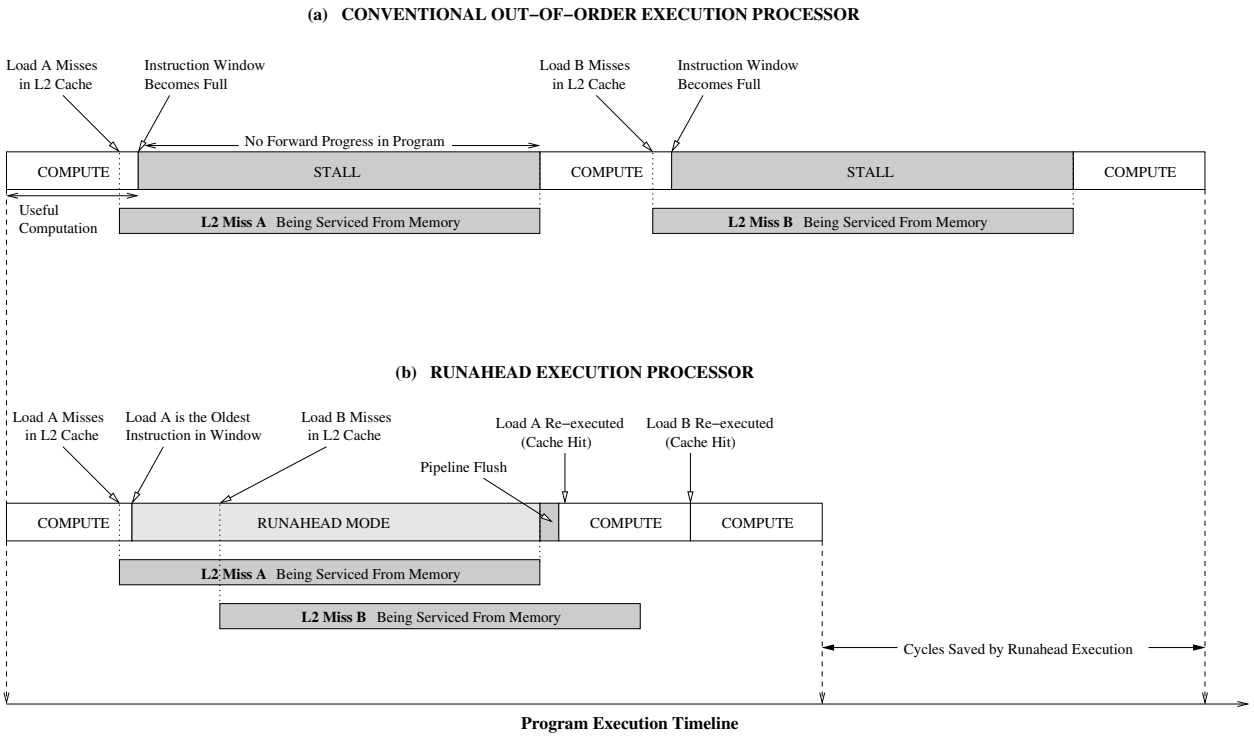


Figure 2.2: Execution timelines showing a high-level overview of the concept of runahead execution. *A runahead processor pre-executes the running application during cycles in which a conventional processor would be stalled due to an L2 cache miss. The purpose of this pre-execution is to discover and 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.*

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 called *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 L2 cache misses and the instructions dependent on them. The results of L2 cache misses and their dependents are identified as bogus (INV). Instructions that generate or source INV

results (called *INV instructions*) are removed from the instruction window so that they do not prevent independent instructions from being placed into the window. Therefore, runahead mode allows the processor to execute more instructions than the instruction window normally permits. 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 2.2(b)). Their miss latencies are overlapped with the latency of the cache miss that caused entry into runahead mode (i.e., 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 2.2(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 2.2(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 speculatively execute the application in order to generate accurate prefetch requests.

2.4 Advantages, Disadvantages and Limitations of the Runahead Execution Paradigm

A great deal of research has been dedicated to building large instruction windows to tolerate long memory latencies. The runahead execution paradigm provides a new alternative to building large instruction windows by capturing the memory-level parallelism (MLP) benefits of a large instruction window without requiring the implementation of large

and complex structures associated with a large window. This approach has advantages as well as disadvantages.

2.4.1 Advantages

Runahead execution improves processor performance by enabling the processor to do useful processing instead of stalling for hundreds of cycles while a long-latency cache miss is being serviced. This speculative processing during runahead mode provides the following performance benefits:

- **Discovery and prefetching of L2 cache data and instruction misses:** Runahead mode execution discovers and initiates L2 cache misses that are independent of the runahead-causing cache miss. Both data and instruction L2 cache misses are discovered and serviced in parallel with the runahead-causing cache miss. Therefore, runahead execution increases the processor's tolerance to the latency of the runahead-causing cache miss and prefetches independent L2 cache misses from DRAM memory into the L2 cache. The nature of prefetching in runahead execution (i.e., execution-based prefetching) enables runahead execution to prefetch both regular and irregular cache miss patterns. Therefore runahead execution can capture a wide variety of access patterns that cannot be captured by conventional software and hardware prefetching techniques which can only capture regular miss patterns.

- **Data and instruction prefetching between levels of the cache hierarchy:** Besides prefetching long-latency misses, runahead execution prefetches data and instructions between levels of the cache hierarchy. For example, an L1 cache miss that hits in the L2 cache is prefetched into the L1 cache during runahead mode.

- **Early training of the hardware data prefetcher:** Load and store instructions speculatively executed during runahead mode train the tables of the hardware data prefetcher

earlier than they would in a conventional processor. This early training improves the timeliness of the accurate hardware prefetch requests.

- **Early resolution of branch instructions and early training of the branch predictor pattern history tables:** Branch instructions that are independent of L2 misses are correctly executed and resolved during runahead mode. These instructions train the branch predictor with their outcomes. Furthermore, structures can be provided to communicate the outcome of branch instructions to normal mode execution.

- **Early execution of L2-miss independent instructions:** The results of L2-miss independent instructions are generated in runahead mode. Performance can be improved if these results are buffered and reused during normal mode. However, Section 5.5.1 shows that such buffering and reuse is not worthwhile to implement.

Overall, runahead execution prepares the processor for processing future instructions instead of keeping it stalled while the initial L2 miss is being serviced.

One important advantage of runahead execution is its simplicity. Compared to the alternative of large instruction windows that require large, cycle-critical, complex, slow, and power-hungry hardware structures in the processor core, runahead execution requires very simple and modest hardware structures, which will be described in detail in Section 2.5. Runahead execution utilizes the existing processing structures to improve memory latency tolerance. None of the additional structures required by runahead execution are large, complex, or on the critical path of the processor. Therefore, the addition of runahead execution to a conventional processor is unlikely to increase the processor's complexity, area, or cycle time.

Instruction execution in runahead mode is *purely speculative* since runahead execution aims to generate only prefetches. A positive consequence of this is that runahead

mode does not impose any functional correctness requirements. Therefore, the processor designer does not have to worry about getting the corner implementation cases functionally right in runahead mode. This makes the addition of runahead execution to a conventional processor possible without significantly increasing the processor's design and verification complexity.

Finally, runahead execution does not require software (programmer, compiler, or ISA) support to improve performance.[1] It is a hardware technique that can improve the performance of existing as well as new program binaries. Therefore, memory-intensive legacy code can benefit from runahead execution and runahead execution can be beneficial in environments where program re-compilation is not a viable or cost-effective option.

2.4.2 Disadvantages

Runahead execution provides a cost- and complexity-effective framework to improve memory latency tolerance. However, runahead execution has some drawbacks that need to be addressed.

First and foremost, runahead execution requires the speculative processing of extra instructions while an L2 cache miss is in progress. These instructions are re-executed in normal mode. Therefore, runahead execution increases the number of instructions executed by a conventional processor, which results in an increase in the dynamic energy dissipated by the processor. Also, this increase in executed instructions may not always result in a performance increase. Chapter 5 addresses this disadvantage of runahead execution and develops new mechanisms that improve the energy-efficiency of a runahead execution processor.

[1] Although, programming and compiler techniques can be developed to increase the performance benefits of runahead execution.

Second, the prefetches generated during runahead mode may not always be accurate because they may be generated on the wrong program path. Inaccurate prefetch requests waste memory bandwidth and can cause pollution in the caches. However, previous analyses found that the positive prefetching effect of wrong-path memory references significantly outweighs these negative effects in a runahead execution processor [76, 77]. Filtering techniques that reduce the negative effects of wrong-path references can also be used with runahead execution [75, 78].

Third, runahead execution may result in performance degradation if the performance benefit of prefetching in runahead mode does not outweigh the performance cost of exiting runahead mode via a pipeline flush. The performance evaluation presented in Chapter 4 shows that this rarely happens.

2.4.3 Limitations

The baseline runahead execution mechanism has two major limitations. These limitations can be reduced by developing complementary mechanisms that address them.

Runahead execution is unable to discover and initiate a cache miss if the miss is dependent on an older L2 miss because the data for the older L2 miss is unavailable during runahead mode. Thus runahead execution cannot parallelize misses that are due to dependent load instructions which are common in programs that utilize and manipulate linked data structures. This limitation can be addressed by augmenting a runahead processor with a value or address prediction mechanism that predicts the values of L2-miss load instructions. Chapter 6 addresses this limitation by developing a new value prediction mechanism that aims to predict the values of *address loads*, i.e. load instructions that load pointers into registers.

Since runahead execution relies on the speculative execution of instructions, it needs an effective instruction supply that brings those instructions into the execution core.

If there are not enough useful instructions to execute during runahead mode, runahead execution may not provide performance benefits. Mispredicted branches that depend on L2 cache misses and instruction/trace cache misses in runahead mode therefore limit the effectiveness of runahead execution. As more effective branch prediction and instruction fetch mechanisms are developed, we would expect the benefits of runahead execution to increase. Section 4.1.3.4 provides an analysis of the impact of the effectiveness of the instruction supply on the performance benefit of runahead execution.

Note that these two limitations also exist in a processor with a large instruction window. Therefore, new techniques that are developed to overcome these limitations are applicable not only to runahead execution but also to large-window processors.

2.5 Implementation of Runahead Execution in a State-of-the-art High Performance Out-of-order Processor

This section describes the implementation of runahead execution in an out-of-order processor and addresses the design tradeoffs that need to be considered when designing a runahead execution processor.

We assume a processor model where instructions access the register file after they are scheduled and before they execute. Pentium 4 [48], MIPS R10000 [126], and Alpha 21264 [58] are examples of such a microarchitecture. In some other microarchitectures, such as the Pentium Pro [47], instructions access the register file before they are placed in the scheduler. The implementation details of runahead execution are slightly different between the two microarchitectures, but the basic mechanism is applicable to both.

2.5.1 Overview of the Baseline Microarchitecture

Figure 2.3 shows the structures in a modern processor's microarchitecture. Additional hardware structures needed for runahead execution are shown in bold and they will be described in this section. Dashed lines show the flow of miss traffic out of the caches.

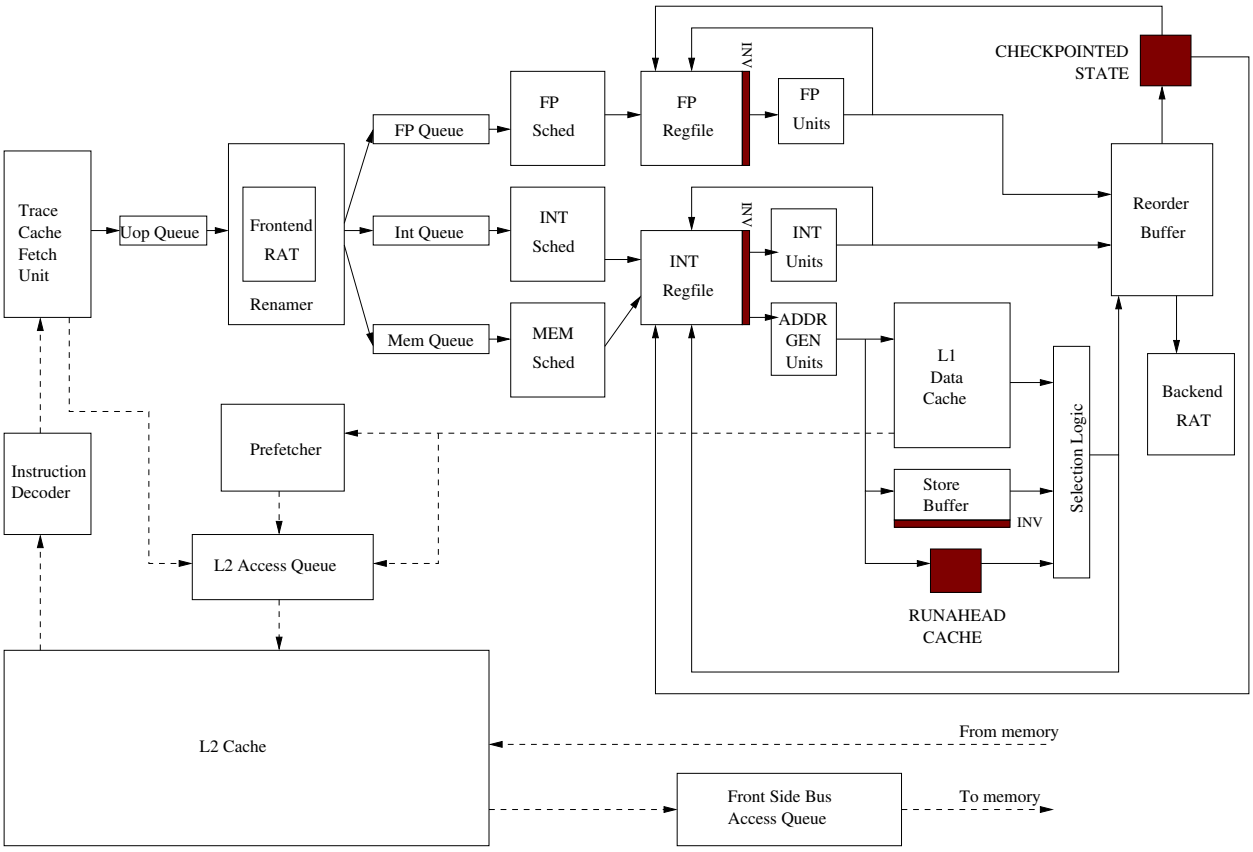


Figure 2.3: Additional hardware structures needed for the implementation of runahead execution in a state-of-the-art out-of-order processor. Additional structures are shown in bold. Dashed lines indicate the flow of cache miss traffic.

Decoded and cracked instructions (micro-operations or *uops*) are fetched from the trace cache and inserted into a Uop Queue. The Frontend Register Alias Table (RAT) [48] is used for renaming incoming instructions and contains the speculative mapping of ar-

chitectural registers to physical registers. The scheduling window of the processor is distributed: integer (INT), floating-point (FP), and memory (MEM) operations have three separate schedulers. Instructions are scheduled from the scheduling window to the execution units when their source operands are ready (i.e., in dataflow order). FP instructions are executed in the FP units after reading their operands from the FP register file. INT and MEM instructions read their operands from the INT register file and are later either executed in the INT units or access the data cache or store buffer after generating their memory addresses. Instructions retire and commit their results to the architectural state in program order. The Backend RAT [48] contains pointers to those physical registers that contain committed architectural values. It is used for recovery of state after branch mispredictions without requiring copying of register data values. The reorder buffer holds the results of completed instructions and ensures the in-order update of the architectural state.

2.5.2 Requirements for Runahead Execution

A processor's pipeline needs to be modified to support runahead execution. We discuss the required modifications and design tradeoffs in three phases of runahead mode operation:

- Entering runahead mode.
- Instruction execution in runahead mode.
- Exiting runahead mode.

A runahead execution processor operates in two different microarchitectural modes: normal mode or runahead mode. One bit (*runahead mode bit*) is used to distinguish the two modes from each other. The state diagram in Figure 2.4 shows the transitions between the two modes.

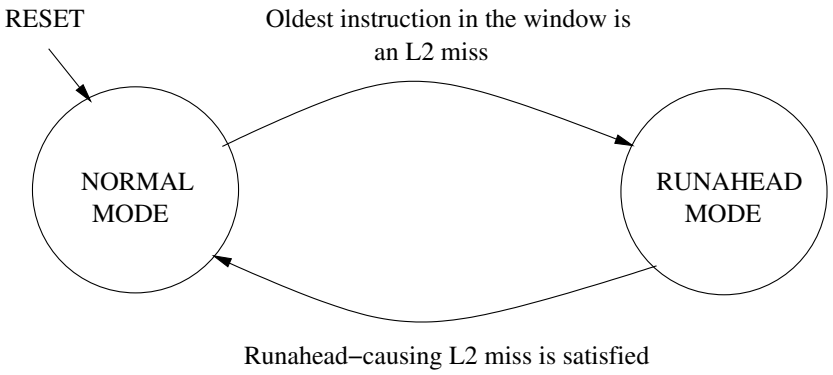


Figure 2.4: State diagram showing the two modes of operation in a runahead execution processor.

2.5.3 Entering Runahead Mode

2.5.3.1 When to Enter Runahead Mode

A runahead execution processor enters runahead mode under two different conditions:

- If the oldest instruction in the window is a load instruction that missed in the L2 cache.
- If the oldest instruction in the window is a store instruction that missed in the L2 cache and the processor's store buffer is full. We assume that an L2-miss store instruction does not block retirement and forward progress unless the store buffer is full.

Several other alternatives were considered as to when to enter runahead mode. We address the design tradeoffs related to these alternatives and explain why they were not chosen.

First, it is possible to enter runahead mode when an L2 miss actually happens instead of waiting for the L2-miss instruction to become the oldest instruction in the window. This would initiate runahead mode early and let the processor execute more instructions in runahead mode, thereby increasing the chances of the discovery of an L2 miss in runahead mode. However, this approach has several drawbacks:

- If the L2-miss instruction is not the oldest, it is not guaranteed that it is a valid instruction that will actually be executed by the processor. The instruction may be on the wrong path due to an older mispredicted branch that has not yet been resolved. Or an older instruction in the window may cause an exception, which requires the flushing of the L2 miss instruction. It is possible to add hardware that checks whether all instructions older than the L2-miss instruction are exception-free and not mispredicted. However, this increases the hardware complexity.

- The potential gains for entering runahead mode at the time an L2 miss happens is limited. We found that an L2-miss instruction becomes the oldest in the window soon (10 cycles on average) after missing in the L2 cache unless an older instruction causes an L2 miss. Hence, entering runahead mode when an L2 miss happens would extend the runahead period by only few cycles, which is negligible compared to the main memory latency that is on the order of hundreds of cycles.

- Entering runahead mode at the time an L2 miss happens requires the state of the machine to be checkpointed at the oldest instruction in the window, which may not be the L2-miss instruction. This throws away the useful work that will be completed soon by relatively short-latency instructions older than the L2-miss instruction, since those instructions will need to be re-executed when the processor returns to normal mode.

Second, it is possible to delay entry into runahead mode until a full-window stall occurs due to an L2-miss instruction that is the oldest in the window. This runahead mode entry policy ensures that a full-window stall will actually occur due to an L2 miss and therefore eliminates possible unnecessary entries into runahead mode. However, it also has two drawbacks that deem it unattractive:

- Waiting for a full-window stall to occur reduces the time spent in runahead mode, which reduces the likelihood of discovering an L2 cache miss in runahead mode.
- We found that the window is already full 78% of the time when the oldest instruction in the window is an L2-miss instruction. Furthermore, the window becomes full 98% of the time after an L2 cache miss happens on the correct program path. Therefore, there is little value in adding extra logic to check for a full window as the window will almost always become full after the occurrence of an L2 miss.

Third, it is possible to enter runahead mode on relatively shorter-latency L1 data cache misses. This would increase the time spent in runahead mode if the L1 miss also misses in the L2 cache. However, most L1 cache misses hit in the L2 cache especially if the L2 cache is large. Furthermore, if the L1 miss hits in the L2 cache, the processor would stay in runahead mode for a very short time. Since an out-of-order processor is good at tolerating relatively short L1 miss/L2 hit latency, entering runahead mode on such misses would not provide any benefit. Therefore our implementation does not enter runahead mode on an L1 cache miss. In contrast, if runahead execution is implemented on an in-order processor [36], it could be beneficial to enter runahead mode on an L1 data cache miss because an in-order processor is unable to tolerate the latency of even an L1 miss.

2.5.3.2 Processor Actions for Runahead Mode Entry

The processor takes the following actions to enter runahead mode when it detects that conditions for entering mode (described in the previous section) are satisfied:

- The address of the instruction that causes entry into runahead mode is recorded in a special register.
- The processor checkpoints the state of the architectural registers. This is required to correctly recover the architectural state on exit from runahead mode.
- The processor also checkpoints the state of the global branch history register and the return address stack so that the context of the branch prediction mechanisms can be recovered to a consistent state when the processor exits runahead mode and restarts normal operation. Note that it is not required to checkpoint the state of these branch prediction structures for functional correctness, but we found that performance is degraded significantly if they are not checkpointed.
- The mode of the processor is switched to runahead mode.

2.5.3.3 Hardware Requirements for Runahead Mode Entry

Entry into runahead mode requires a mechanism to convey to the scheduler that an instruction missed in the L2 cache. This is accomplished by simply extending the miss signal for the instruction's access from the L2 cache controller into the processor core.

The mechanism for checkpointing the architectural registers depends on the microarchitecture. There are several alternatives:

- The entire architectural register file can be checkpointed. The hardware cost of this can be considerable especially if the number of architectural registers is large. However, this may be the only option for microarchitectures that store the architectural

register state in a dedicated architectural register file. A flash copy of the architectural registers can be made using circuit techniques so that no cycles are lost to take the checkpoint.

- Microarchitectures that store the architectural register state in the physical register file can avoid checkpointing the entire architectural register state by checkpointing only the register map that points to the architectural state (i.e. Backend RAT in Figure 2.3). This significantly reduces the hardware cost of the checkpoint. In this case, physical registers that are part of the architectural state are not deallocated during the entire runahead period. Therefore, runahead mode execution has fewer available registers. We found that the performance degradation due to reduced number of available registers is negligible. Our experiments use this option.

- The checkpointing of the architectural register file can also be accomplished by copying the contents of the physical registers pointed to by the Backend RAT. This may take some time. To avoid performance loss due to this copying, the processor can always update the checkpointed architectural register file during normal mode. This option keeps all physical registers available during runahead mode. However, it is unattractive because it increases the processor's energy consumption since it requires the checkpointed register file to be updated during normal mode.

The return address stack can also be checkpointed in several ways. Instead of copying the entire stack, we use a mechanism proposed by Jourdan et al. [57] that checkpoints the return address stack without significant hardware cost.

Note that the hardware needed to checkpoint the state of the architectural register file and return address stack already exists in modern out-of-order processors to support recovery from branch mispredictions. Many current processors checkpoint the architectural register map after renaming a branch instruction [126, 58] and the return address stack after

fetching a branch instruction. Runahead execution only requires modifications to take these checkpoints when the oldest instruction is an L2 miss.

2.5.4 Instruction Execution in Runahead Mode

The execution of instructions in runahead mode is very similar to instruction execution in a conventional out-of-order execution processor with the following key differences:

- Execution in runahead mode is *purely speculative*. Runahead instructions do not update the architectural state.
- L2-miss instructions and their dependents are tracked in runahead mode. Their results are marked as INV and they are not allowed to block the retirement logic.

The main complexities involved in execution of runahead instructions are the propagation of INV results and the communication between stores and loads. This section describes the rules of the machine, the hardware required to support them, and the tradeoffs involved in deciding the rules.

2.5.4.1 INV Bits and Instructions

A runahead execution processor treats L2-miss dependent values and instructions differently from other instructions in runahead mode. L2-miss dependent values and instructions are marked as invalid (INV) by the processor. A runahead processor keeps track of INV instructions for three purposes:

1. INV instructions can be removed from the instruction window without waiting for the completion of the L2 miss they are dependent on. This creates space in the instruction window for L2-miss independent instructions and allows the processor to make forward progress in runahead mode without incurring full-window stalls.

2. Branches that are dependent on L2 misses are not resolved. Since the source data value of an INV branch is not available in runahead mode, the processor relies on the branch predictor's prediction for that branch rather than using a bogus stale value to resolve the branch.

3. Load and store instructions whose addresses are dependent on L2 misses are not allowed to generate memory requests since their addresses would be bogus. This reduces the probability of polluting the caches with bogus memory requests in runahead mode.

Each physical register and store buffer entry has an INV bit associated with it to indicate whether or not it has a bogus value. Any instruction that sources a register or a store buffer entry whose INV bit is set is an INV instruction.

If the data of a store instruction is INV, the store introduces an INV value to the memory image in runahead mode. To handle the communication of INV status of memory locations and valid data values through memory in runahead mode, we use a small speculative memory, called "runahead cache," that is accessed in parallel with the first-level data cache. We describe the rationale for the runahead cache and its design in Section 2.5.4.4.

2.5.4.2 Propagation of INV Values

The first instruction that introduces an INV data value is the instruction that causes the processor to enter runahead mode. If this instruction is a load, it marks its physical destination register as INV. If it is a store, it allocates a line in the runahead cache and marks its destination bytes in memory as INV.

Any INV instruction that writes to a register marks that register as INV after it is scheduled. Any valid operation that writes to a register resets the INV bit associated with its physical destination register.

2.5.4.3 Hardware Requirements to Support INV Bits and Their Propagation

INV status is a property of data that is stored in registers or in memory. Therefore, INV bits are propagated in the datapath with the data they are associated with. The control mechanism that communicates INV bits between dependent instructions is already present in an out-of-order processor. An out-of-order processor communicates data values between dependent instructions. An INV bit can be appended to each data value and simply communicated along with the data value it is associated with. Therefore adding support for INV bits in the processor core would not significantly increase processor complexity.

Furthermore, bits identifying register values dependent on cache misses already exist in aggressive implementations of out-of-order execution to support replay scheduling [107]. A modern out-of-order processor removes the cache-miss dependent instructions from the scheduling window so that they do not occupy the scheduling window entries for a long time [12]. The hardware that exists to support this can be used or can be replicated to support INV bits.

2.5.4.4 Runahead Store Operations and Runahead Cache

In a previous proposal for runahead execution on in-order processors [36], runahead store instructions do not write their results anywhere. Therefore, runahead loads that are dependent on valid runahead stores are regarded as INV instructions and dropped. In our experiments, we found that forwarding the results of valid runahead stores to dependent runahead loads doubles the performance benefit of runahead execution on an out-of-order processor (See Section 4.1.3.5). Therefore, the runahead processor described in this dissertation provides support for the forwarding of the results of runahead store operations to dependent runahead load operations.

If both the store and the load that depends on the store are in the instruction window during runahead mode, forwarding the value of the store to the load is accomplished

through the store buffer that already exists in current out-of-order processors. However, if a runahead load depends on a runahead store that has already pseudo-retired (which means that the store is no longer in the store buffer), it should get the result of the store from some other location. One possibility is to write the result of the store into the data cache. This introduces extra complexity to the design of the data cache (and possibly to the second-level cache) because the data cache needs to be modified so that data written by speculative runahead stores is not used by future non-runahead instructions. Writing the data of speculative runahead stores into the data cache can also evict useful cache lines and therefore reduce the cache hit rate. Another possibility is to have a large fully-associative buffer that stores the results of pseudo-retired runahead store instructions. Such a store buffer can be implemented by enlarging the processor's store buffer or by adding another buffer similar to the existing store buffer, which is a multi-ported content-addressable memory (CAM) structure. But, the size and access time of this associative structure can be prohibitively large. Also, such a structure cannot handle the case where a load depends on multiple stores, without increased complexity.

As a simpler alternative, this dissertation proposes the use of a scratch-pad memory organized as a cache, called *runahead cache*, to hold the results and INV status of pseudo-retired runahead stores. The runahead cache is addressed just like the data cache, but it can be much smaller in size because a small number of store instructions pseudo-retire during runahead mode. Unlike the processor's store buffer, the runahead cache is not a CAM structure but a RAM structure with less complex hardware and less power consumption.

Although we call it a cache because it is physically the same structure as a traditional cache, the purpose of the runahead cache is not to cache data. Its purpose is to provide for the communication of data and INV status between store and load instructions. The evicted cache lines are not stored in any other larger storage; they are simply dropped (Since runahead mode is purely speculative, there is no requirement to satisfy correct prop-

agation of data between store instructions and dependent load instructions). Runahead cache is only accessed by loads and stores executed in runahead mode. In normal mode, no instruction accesses it.

To support correct communication of INV bits between stores and loads, each entry in the store buffer and each byte in the runahead cache has a corresponding INV bit. Each byte in the runahead cache also has another bit associated with it (STO bit) indicating whether or not a store has written to that byte. An access to the runahead cache results in a hit only if the accessed byte was written by a store (STO bit is set) and the accessed cache line is valid. Figures 2.5 shows the high-level block diagram of the runahead cache and the structure of the tag array.

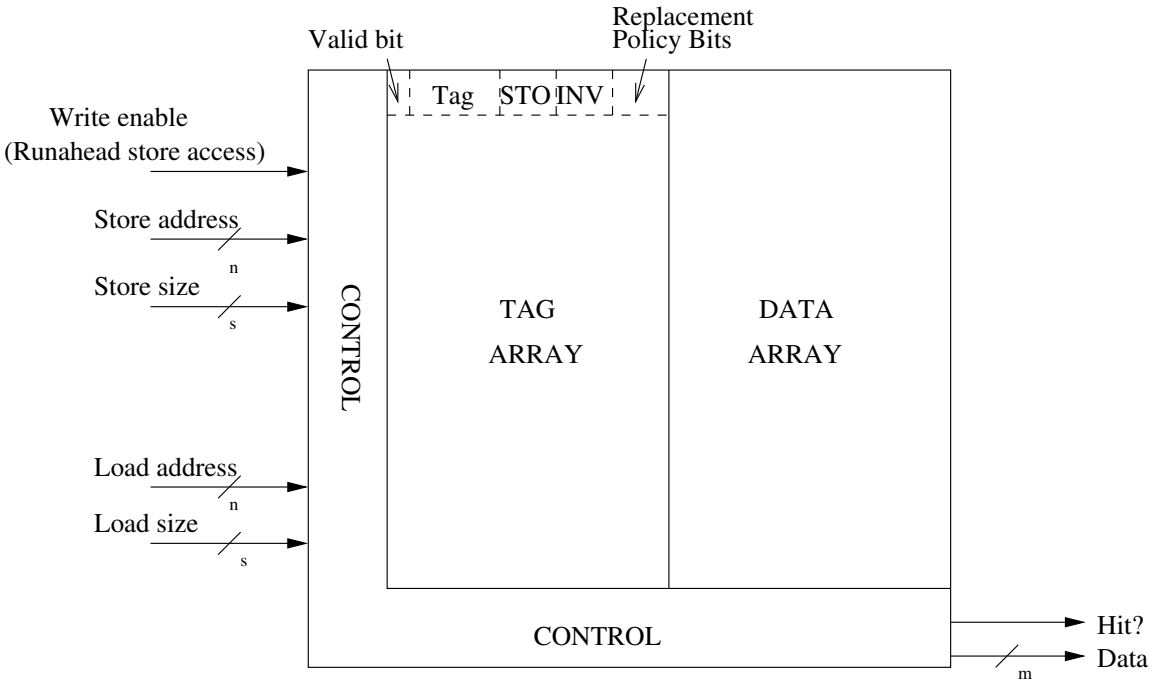


Figure 2.5: High-level block diagram of a runahead cache with a single read port and a single write port.

Store instructions follow the following rules to update the store buffer and the runahead cache during runahead mode:

- When a valid runahead store completes execution, it writes its data into its store buffer entry (just like in a conventional processor) and resets the associated INV bit of the entry. In the meantime, it queries the data cache and sends a prefetch request down the memory hierarchy if it misses in the data cache.
- When an invalid runahead store is scheduled, it sets the INV bit of its associated store buffer entry.
- When a valid runahead store exits the instruction window, it writes its result into the runahead cache and resets the INV bits of the written bytes. It also sets the STO bits of the bytes it writes into.
- When an invalid runahead store exits the instruction window, it sets the INV bits and the STO bits of the bytes it writes into (if its address is valid).
- Runahead stores never write their results into the data cache.

One complication arises when the address of a store operation is INV. In this case, the store operation is simply treated as a NOP. Since loads are unable to identify their dependencies on such stores, it is likely that a load instruction that is dependent on an INV-address store will load a stale value from memory. This stale value might later be used to resolve dependent branches or generate addresses of dependent load instructions, which might result in the overturning of correct branch predictions or the generation of inaccurate memory requests. This problem can be mitigated through the use of memory dependence predictors [72, 25] to identify the dependence between an INV-address store and its dependent load. Once the dependence has been identified, the load can be marked

INV if the data value of the store is INV. If the data value of the store is valid, it can be forwarded to the load that is predicted to be dependent on the store. In our evaluation we found that INV-address stores do not significantly impact performance. If the address of a store is INV, it is more than 92% likely that the address of a dependent load instruction is also INV. Therefore, load instructions that depend on INV-address stores rarely introduce stale values into the processor pipeline.

2.5.4.5 Runahead Load Operations

A runahead load operation can become invalid due to four different reasons:

1. It may source an INV physical register.
2. It may miss in the L2 cache.
3. It may be dependent on a store that is marked as INV in the store buffer.
4. It may be dependent on a INV store that was already pseudo-retired.

The last case is detected using the runahead cache (unless the runahead cache line was evicted due to a conflict). When a valid load executes, it accesses three structures in parallel: the data cache, the runahead cache, and the store buffer. If it hits in the store buffer and the entry it hits is marked as valid, the load gets its data from the store buffer. If the load hits in the store buffer and the entry is marked INV, the load marks its physical destination register as INV.

A load is considered to hit in the runahead cache only if (1) the runahead cache line it accesses is valid and (2) the STO bit of any of the accessed bytes in the cache line is set. If the load misses in the store buffer and hits in the runahead cache, it checks the INV bits of the bytes it is accessing in the runahead cache. The load executes using the data in the

runahead cache if none of the INV bits of the requested data is set. If any of the data bytes sourced by the load are marked INV, then the load marks its destination register as INV.

If the load misses in both the store buffer and the runahead cache, but hits in the data cache, then it uses the value from the data cache and is considered valid. Nevertheless, it may actually be dependent on an L2 miss because of two reasons:

1. It may be dependent on a store with INV address.
2. It may be dependent on an INV store that marked its destination bytes in the runahead cache as INV, but the corresponding line in the runahead cache was evicted due to a conflict.

However, we found that both of these are rare cases that do not affect performance significantly.

If the load misses in all three structures, it sends a request to the L2 cache to fetch its data. If this request hits in the L2 cache, data is transferred from the L2 to the L1 data cache and the load completes its execution. If the request misses in the L2, the load marks its destination register as INV and is removed from the scheduler, just like the load that caused entry into runahead mode. The request that misses in the L2 is sent to main memory as a prefetch request.

2.5.4.6 Hardware Requirements to Support Runahead Store and Load Operations

The runahead cache is the largest structure that is required by runahead execution. However, it is very small compared to the L1 data cache of the processor. In our experiments we found that a 512-byte runahead cache is large enough to capture almost all communication between runahead stores and dependent runahead loads. We also found that the runahead cache is very latency tolerant and therefore the processor does not need

to access it in parallel with the data cache. Hence, the runahead cache is not on the critical path of the processor.

Aside from the runahead cache, runahead execution does not require significant hardware to handle runahead stores and loads. The forwarding of INV bits from the store buffer to a dependent load is accomplished similarly to the forwarding of data from the store buffer. The hardware support for data forwarding from the store buffer already exists in current out-of-order processors – runahead execution adds only one more bit (the INV bit) to the forwarding data path.

2.5.4.7 Prediction and Execution of Runahead Branches

Branches are predicted and resolved in runahead mode exactly the same way they are in normal mode except for one difference: A branch with an INV source cannot be resolved. A valid branch is predicted and resolved during runahead mode. If a valid branch is mispredicted in runahead mode, the state of the machine is recovered and the fetch engine is redirected to the correct fetch address, just like in normal mode. An INV branch is also predicted and it updates the global branch history register speculatively like all normal-mode branches. However, an INV branch cannot be resolved because the data value it is dependent on has not yet come back from memory. This is not a problem if the branch is correctly predicted. However, if the branch is mispredicted, the processor will be on the wrong path after it fetches the branch until it reaches a control-flow independent point. We call the point in the program where a mispredicted INV branch is fetched the *divergence point*. Existence of divergence points is not necessarily bad for performance because wrong-path memory references can provide significant prefetching benefits [77]. Nevertheless, as we will show later, the performance improvement provided by runahead execution would increase significantly if the divergence points did not exist (i.e., if all INV branches were correctly predicted).

When a valid branch pseudo-retires, it can update the pattern history tables (PHTs) of the branch predictor. In contrast, an INV branch is not allowed to update the PHTs because the correct direction of the branch cannot be determined. The update of the branch predictor's pattern history tables (PHTs) with the outcomes of valid branches can be handled in several different ways:

- One option is to allow the valid runahead branches to update the PHTs. If a branch executes first in runahead mode and then in normal mode, this option causes the same PHT entry to be updated twice by the same dynamic branch instruction. The branch prediction counter may therefore be strengthened unnecessarily and it may lose its hysteresis.

- A second option is not to allow the valid runahead branches to update the PHTs. This means that the new information that becomes available during runahead mode is not incorporated into the branch predictor and branches in runahead mode are predicted using stale information from normal mode. This is not a problem if the branch behavior does not change after entering runahead mode. We found that this option results in a slightly reduced branch prediction accuracy in runahead mode.

- A third option is to update the PHTs only with the outcomes of mispredicted valid branches. This has the benefit of keeping the PHTs up-to-date during runahead mode. It also avoids over-strengthening the prediction counters for correctly-predicted valid branches. We found that this option results in the highest prediction accuracy among the first three options and therefore use this as our baseline PHT update policy in runahead mode.

- Finally, a fourth option is to allow the valid runahead branches to update the PHTs but disallow the same branches from updating the PHTs when they are re-executed in

normal mode. This can be accomplished with a FIFO queue that stores the outcomes of pseudo-retired runahead branches. A branch that is fetched in normal mode accesses this queue for a prediction. If the branch was executed in runahead mode, the processor reads from the queue the outcome of the branch that was computed in runahead mode and uses this outcome as the branch's predicted direction in normal mode. This option has higher hardware cost than the previous options. However, it not only avoids the problem of over-strengthening the prediction counters but also provides a way of reusing in normal mode the branch outcomes computed in runahead mode. However, we found that the performance improvement provided by this PHT update policy is not significant (less than 1%) and therefore its implementation is likely not justifiable, a conclusion also supported by previous research in the area [37].

2.5.4.8 Instruction Pseudo-Retirement in Runahead Mode

During runahead mode, instructions leave the instruction window in program order without updating the architectural state of the processor. We call this process *pseudo-retirement*. Pseudo-retirement is very similar to retirement in normal mode except updates to the architectural state are not allowed. Specifically, pseudo-retired instructions do not update the state of the architectural register file that was checkpointed upon entry into runahead mode, and pseudo-retired stores do not update the architectural memory state.

When an instruction becomes the oldest in the instruction window in runahead mode, it is considered for pseudo-retirement. If the instruction considered for pseudo-retirement is INV, it is removed from the window immediately. If it is valid, it needs to wait until it is executed (at which point it may become INV) and its result is written into the physical register file. Upon pseudo-retirement, an instruction releases all resources allocated for its execution, including entries in the instruction window, load/store buffers, and branch checkpoint table.

Both valid and INV instructions update the Backend RAT when they leave the instruction window just like retired normal-mode instructions. This allows for correct state recovery upon a branch misprediction in runahead mode.[2] The Backend RAT does not need to store INV bits associated with each register because physical registers already have INV bits associated with them. However, in a microarchitecture that keeps the retired state in a separate register file (i.e., Retirement Register File as in Pentium Pro [48]) than the physical register file, this separate register file does need to store INV bits.

2.5.4.9 Exceptions and Input/Output Operations in Runahead Mode

Execution in runahead mode is not visible to the architectural state of the program. Therefore, exceptions generated by runahead mode instructions are not handled. However, an exception during the execution of an instruction usually implies that something has gone wrong with the instruction and its result cannot be computed or trusted. For example, a load instruction that incurs a page fault cannot load the correct result into its destination register until after the page fault is handled by the operating system. To avoid using bogus values due to exception-causing instructions, a runahead execution processor marks as INV the destination register or memory location of an instruction that causes an exception in runahead mode.

Since runahead mode is not supposed to modify the architectural state, it cannot modify the architectural state of the devices that communicate with the program. Therefore any operation that changes the architectural state of those devices is not permitted during runahead mode. Examples of such operations are special input and output instructions or load/store instructions that access memory-mapped I/O devices. These operations are

[2] When a mispredicted branch becomes the oldest instruction in the window, the Backend RAT is copied into the Frontend RAT to recover the state of the register file to the point after the branch but before the next instruction. Note that this misprediction recovery mechanism is the same in runahead mode as in normal mode.

not performed in runahead mode and their destination registers or memory locations are marked as INV so that dependent operations do not source bogus values.

2.5.5 Exiting Runahead Mode

2.5.5.1 When to Exit Runahead Mode

An exit from runahead mode can be initiated at any time while the processor is in runahead mode. Our policy is to exit from runahead mode when any of the following happens:

- The runahead-causing L2 cache miss is serviced. An L2 cache miss is serviced when the data required by it has returned to the processor core from main memory.
- An external interrupt with a priority higher than the priority of the running program needs to be serviced.

Several other alternatives were considered as to when to exit runahead mode. We address the design tradeoffs related to these alternatives and explain why they were not chosen in our baseline runahead execution implementation.

First, it is possible to exit runahead mode earlier than when the runahead-causing L2 cache miss is serviced. This mechanism can be advantageous because the processor can switch to runahead mode early and hide the delay associated with exiting from runahead mode. Runahead mode exit requires a pipeline flush and exiting runahead mode early can enable the processor to fill its pipeline and instruction window by the time the runahead-causing L2 cache miss completes. However, this approach has several drawbacks:

- It is not easy to determine exactly when is a good time to exit runahead mode so that the delay associated with runahead exit is hidden. The processor needs to know

when the L2 cache will be serviced in order to decide when to exit runahead mode to hide the exit penalty. However, in current memory hierarchies the time required to service an L2 cache miss is variable and essentially unknown to the processor because of queueing delays, bank conflicts, port contentions that can happen in the DRAM chips and the memory controller.

- Exiting runahead mode early reduces the time spent in runahead mode, which in turn reduces the forward progress made in runahead mode in executing further in the instruction stream. This may reduce the number of L2 cache misses discovered in runahead mode and hence reduce the performance benefits of runahead execution.

We simulated this alternate approach using oracle information to determine exactly when runahead exit should be initiated in order to fully hide the runahead exit penalty. We found that the alternate policy slightly improves performance for some benchmarks whereas it degrades performance for others. On average, exiting runahead mode early–even using oracle information–performs 1% worse than exiting runahead mode when the L2 cache miss is serviced. We found that, in many benchmarks, additional L2 cache misses are discovered in runahead mode if the processor does not exit from runahead mode early. The prefetching benefit due to the discovery of these L2 misses on average outweighs the performance loss due to the pipeline flush penalty at runahead mode exit.

A second alternative approach is to delay runahead exit for a number of cycles after the runahead-causing L2 cache miss is serviced. This could increase the performance benefit of runahead execution if it results in the discovery of additional L2 cache misses and if the performance benefit of prefetching those misses outweighs the performance loss due to delaying normal-mode progress in the running application. On the other hand, if no L2 misses are discovered in the additional cycles devoted to runahead execution, this alternate approach would cause performance degradation. In this case, the processor could

have executed useful instructions in normal mode and made forward progress instead of wasting cycles in runahead mode without gaining any benefit. In our evaluations, we found that some benchmarks (with high levels of memory-level parallelism) benefit from staying in runahead mode even hundreds of additional cycles after the runahead-causing L2 cache miss is serviced. In contrast, many benchmarks' performance degraded if runahead exit was delayed. Therefore, the baseline runahead execution mechanism does not delay runahead exit after the runahead-causing L2 cache miss is serviced.

A more aggressive implementation of runahead execution can predict whether the benefits of continuing in runahead mode outweigh the benefits of exiting runahead mode early or when the runahead-causing miss is serviced. Using this prediction, such an implementation can dynamically decide exactly when to exit runahead mode. This dissertation investigates such mechanisms to improve the efficiency of runahead execution in Section 5.5.3.

2.5.5.2 Processor Actions for Runahead Mode Exit

For simplicity, we handle the exit from runahead mode similarly to the way a branch misprediction is handled. The processor takes the following actions to exit runahead mode:

- All instructions in the machine are flushed and the buffers are allocated for them are deallocated.
- The checkpointed architectural registers are restored. The hardware required for restoring the checkpointed architectural registers depends on the checkpointing mechanism used. Since we use a mechanism that only checkpoints the Backend RAT (as described in Section 2.5.3.3), the restoration of the checkpoint can be accomplished by simply copying the contents of the checkpointed Backend RAT into the Frontend RAT.

- The checkpointed branch history register and return address stack are restored.
- The runahead cache is flushed. All lines in the runahead cache are invalidated and the STO bits in the lines are reset. Note that it is not necessary to flush the runahead cache for correct operation in runahead mode because instructions in normal mode do not access the runahead cache. However, runahead cache is flushed so that load operations in a later runahead mode do not read the stale values produced by stores from a previous runahead mode.
- The program counter is set to the address of the instruction that caused runahead mode.
- The mode of the processor is switched to normal mode. After switching to normal mode the processor starts fetching instructions starting with the instruction that caused entry into runahead mode.

2.5.5.3 Hardware Requirements for Runahead Mode Exit

The hardware needed for flushing the processor pipeline and restoring the checkpointed state is the same as the hardware needed for recovering from a branch misprediction. Therefore, branch misprediction recovery hardware can be used for runahead mode exit without increasing processor complexity.

In addition, hardware support for flushing the runahead cache is required. This can be done using gang-invalidation (or flash invalidation), which clears the whole cache using circuit techniques and is already implemented in processors that support the flushing of the translation lookaside buffers (TLBs).

2.5.6 Multiprocessor Issues

Implementation of runahead execution in a processor does not introduce any complications in the design of a multiprocessor system that consists of runahead execution processors. This is because the results produced during runahead mode in one processor are not visible to any other processor or device in the system; they are purely speculative.

As described before, L2 miss requests to main memory generated in runahead mode are treated as prefetch requests. Multiprocessor systems already support speculative prefetch requests and prefetch requests generated in runahead mode are handled the same way as a conventional prefetch request. In a multiprocessor system, if the data requested by the prefetch resides in the L2 cache of another processor, it is transferred from that processor to the requesting processor. This causes the prefetched cache line to transition to "shared" state in the coherence mechanism because the same cache line would reside in multiple different L2 caches. Aside from this, runahead mode execution in one processor does not cause any side effects in the L2 caches of other processors.

2.5.6.1 Lock Operations

Programmers can use special lock operations (e.g. the LOCK prefix in the x86 ISA [52] or the CASA (compare and swap) instruction in the SPARC ISA [106]) to atomically acquire, test, or modify a lock variable in shared memory, which is used to ensure that critical sections are not executed concurrently by different threads in multithreaded programs. A runahead execution processor does not need to obey the atomicity semantics of lock operations in runahead mode because purely-speculative runahead mode execution cannot modify any critical shared data. A runahead execution processor treats the lock operations differently from a conventional processor in the following cases:

- If a lock operation is encountered in runahead mode, it is simply treated as a normal

instruction that does not require atomic updates to the memory location that stores the lock variable. For example, the `LOCK INC [edx]` instruction in the x86 ISA which is supposed to atomically increment the memory location pointed to by the `edx` register is simply treated as `INC [edx]` in runahead mode. Since runahead mode cannot make any changes to the lock variable or to the critical data protected by the lock variable, there is no need to obey the atomicity dictated by lock operations. This could enable faster progress in runahead mode without incurring the latency associated with lock operations.

- If a lock operation incurs an L2 cache miss in normal mode, it causes entry into runahead mode similar to a load instruction. The runahead-causing lock operation is then treated as a normal instruction that does not have lock semantics. Hence, runahead mode can be initiated without acquiring the lock and the processor can enter a critical section. This allows the processor to speculatively prefetch the data required by the critical section.

2.5.6.2 Serializing Instructions

Programmers can use special serializing instructions to guarantee that memory operations occur in the intended order in a multiprocessor system. Two examples are the MFENCE (memory fence) instruction in the x86 ISA [52] and the MB (memory barrier) instruction in the Alpha ISA [101]. These operations make sure that all the previous memory access operations in the program running on the processor are globally visible to other processors before any of the following memory access operations are globally visible to other processors. Hence, serializing instructions enforce an order between the memory operations before them and the memory operations after them. Most modern processors handle these instructions by draining the pipeline before a serializing instruction is executed [23], which requires all the previous instructions to be committed to the architectural

state. This causes a performance loss.

As runahead mode is purely speculative, it does not need to obey the serialization constraints imposed by such instructions. Therefore a runahead processor treats the serializing instructions differently from a conventional processor in runahead mode. If a serializing instruction is encountered in runahead mode, it is simply treated as a normal instruction that does not enforce serialization. For example, the MFENCE instruction is treated as a NOP. This ensures fast forward progress in runahead mode by eliminating the pipeline drain due to such instructions. Also, it allows the processor to speculatively execute load and store instructions after the serializing instruction and service the misses generated by them in parallel with other misses generated by instructions before the serializing instruction.

This dissertation evaluates runahead execution and proposes mechanisms to enhance its efficiency and effectiveness on a uniprocessor. Due to the purely speculative nature of runahead mode execution, the proposed mechanisms are directly applicable to multiprocessor systems that consist of runahead execution processors without any modifications. However, a rigorous performance evaluation of runahead execution on multiprocessor systems is out of the scope of this dissertation and is left for future work.

Chapter 3

Background and Related Work

Memory access is a very important long-latency operation that has concerned researchers for a long time. This section classifies the relevant approaches into six categories (caching, prefetching, out-of-order processing, enhanced in-order processing, multithreading, and parallelizing dependent load instructions) and briefly describes the proposed approaches in relation to runahead execution and the techniques proposed in this dissertation. We also provide an overview of the recent related work in runahead execution.

3.1 Related Research in Caching

Caches [121] tolerate memory latency by exploiting the temporal and spatial reference locality of applications. Kroft [62] improved the latency tolerance of caches by allowing them to handle multiple outstanding misses and to service cache hits in the presence of pending misses. Caches are very effective and are widely used in modern processors. However, their effectiveness is limited due to two main reasons. First, some applications exhibit poor temporal and spatial locality. Second, cache miss penalties are prohibitively large, so even with a very low but non-zero cache miss rate, the effective memory hierarchy access time remains high. To reduce the cache miss rates, researchers developed software, hardware, and thread-based prefetching techniques.

This dissertation evaluates runahead execution on a baseline processor with large caches. We show that runahead execution is an effective means of improving processor

performance in the presence of large L2 caches (Section 4.1.3.6).

3.2 Related Research in Prefetching

3.2.1 Software Prefetching

Software prefetching techniques [60, 16, 74, 68] are effective for applications where the compiler can statically predict which memory references will cause cache misses. For many integer applications this is not a trivial task. These techniques also insert prefetch instructions into applications, increasing instruction bandwidth requirements. In contrast to these techniques, runahead execution does not require the modification of existing binaries and can prefetch memory reference patterns that cannot be predicted at compile time.

3.2.2 Hardware Prefetching

Hardware prefetching techniques [41, 56, 5, 55, 29] use dynamic information to predict what and when to prefetch. They do not require any instruction bandwidth. Different prefetch algorithms cover different types of access patterns. These techniques work well if the reference stream of the running application is regular. Streaming and striding prefetchers [56, 88] that are designed to capture regular patterns are already employed in processors [48, 113]. The design of hardware prefetchers that can capture both irregular and regular reference streams is still an open research topic. Hardware prefetching techniques also degrade performance due to cache pollution and unnecessary bandwidth consumption, if the accuracy of the hardware prefetcher is low.

Runahead execution, on the other hand, can capture both regular and irregular memory access patterns because it relies on the execution of instructions rather than the discovery of regular memory access patterns. This dissertation evaluates runahead execution on a baseline processor that employs aggressive stream-based prefetching. A detailed compar-

ison of runahead execution and stream-based prefetching is provided in Sections 4.1.3.1, 4.1.3.2, and 4.2.5.1. The results show that runahead execution is more effective than and complementary to stream-based hardware prefetching.

3.2.3 Thread-based Prefetching

Thread-based prefetching techniques [27, 67, 132, 96] use idle thread contexts on a multithreaded processor to run threads that help the primary thread [20]. These helper threads execute code that prefetches for the primary thread. The advantage of these techniques is they can prefetch irregular memory reference patterns. The main disadvantages of these techniques are: (1) they require the generation of helper threads, (2) they require idle thread contexts and spare resources (e. g., fetch and execution bandwidth), which are not available when the processor is well used. These techniques also require hardware support for executing multiple threads simultaneously. In contrast to these techniques, runahead execution neither requires a separate active thread context nor contends with the main program thread for resources, but it can still prefetch irregular memory access patterns.

3.3 Related Research in Out-of-order Processing

Out-of-order execution [116] has long been used to tolerate the latency of memory accesses. Precise exception support was first incorporated into out-of-order execution by Patt et al. [90]. The latency tolerance provided by an out-of-order processor that supports precise exceptions is limited by the instruction window size. Although out-of-order execution has been widely employed to achieve high performance, the instruction window of an out-of-order processor needs to be very large to tolerate current main memory latencies, as shown in the previous section. Therefore, recent related research in out-of-order execution focused on two major efforts: (1) More efficiently utilizing resources in out-of-order processors by augmenting small instruction windows and (2) building large

instruction windows to enhance out-of-order execution's capability to tolerate long main memory latencies.

3.3.1 Efficient Utilization of Small Instruction Windows

Several recent papers [73, 69, 30] proposed more efficient use of the resources in out-of-order processors so that small instruction windows do not stall as frequently as they do. These proposals allow the instructions to release the load buffer [69], store buffer [69], and physical register file [73] entries before the instructions are retired, under certain conditions. An orthogonal scheme that delays the allocation of physical registers until they are needed was also proposed [44]. With these schemes, instructions do not unnecessarily occupy resources that may be needed by other instructions in the instruction window. However, these mechanisms do not allow the instruction window (reorder buffer) entries to be released, which reduces their memory latency tolerance. The evaluations of out-of-order processors presented in this dissertation incorporate in the baseline processor the maximum possible benefits that could be gained by these previous proposals. Section 4.2.5.3 shows that runahead execution provides significant benefits on a small instruction window that already utilizes its resources efficiently.

Balasubramonian et al. [6] proposed a mechanism to execute future instructions when a long-latency instruction blocks retirement in an out-of-order processor. Their mechanism dynamically allocates a portion of the register file to a *future thread*, which is launched when the *primary thread* stalls. This mechanism requires partial hardware support for two different thread contexts. Unfortunately, when the resources are partitioned between the two threads, neither thread can make use of the machine's full resources, which decreases the future thread's benefit and increases the primary thread's stalls. Also, this mechanism adds significant complexity to the register file and scheduler entries, requiring counters that need to be incremented every cycle, and adds complex control structures

to manage communication between the future thread and the primary thread. In contrast, runahead execution does not require a separate active thread context and it does not significantly increase processor complexity. In addition, both normal and runahead mode can make use of the machine's full resources, which helps the machine to get further ahead during runahead mode.

Pai and Adve [86] proposed a compiler technique that increases the memory latency tolerance of a small instruction window. In their mechanism, the compiler restructures the code such that long-latency misses occur in clusters so that they can be captured by a small instruction window that exists in current processors. Unfortunately, their proposed algorithm is effective only for benchmarks where the memory reference pattern is regular and predictable at compilation time. Pai and Adve's mechanism can also be used to increase the effectiveness of runahead execution. If the code is structured such that more long-latency cache misses are clustered in a program, a runahead processor can discover more long-latency misses during runahead mode.

3.3.2 Building Large Instruction Windows

3.3.2.1 The Problem

There has been significant amount of research in techniques to implement large instruction windows in recent years. Implementing a large instruction window is a difficult problem because the size of several processor structures need to be increased proportionally with the instruction window size in order to accommodate a larger number of instructions. This results in increases in the energy/power consumption, design and verification complexity, area, and access time of some structures that are on the critical path, which may also result in the increase of the processor's cycle time or pipeline depth. The following structures need to be enlarged in order to build a large instruction window:

- **Reorder buffer:** This buffer keeps ordering, control, and other bookkeeping information about all the instructions in the window. The sizes of the reorder buffer and structures used to store bookkeeping information about in-flight instructions need to be scaled with the instruction window size. This increases the hardware cost and power consumption of these structures. However, as reorder buffer is not on the critical path of normal execution, the design complexity is not significantly increased.

- **Physical register file:** In-flight instructions need to store their results in registers. The physical register file is a multi-ported structure that stores the values written by in-flight instructions so that dependent instructions can access those values. As the window size gets larger, the size of the register file also needs to get larger. This increases the area, power consumption, access time, and design complexity of the register file significantly. Furthermore, increasing the access time of the register file may result in performance loss because the register file is on the critical path of execution.

- **Store buffer (store queue):** This structure keeps the ordered list of store instructions in the window. Each entry in the store buffer stores the address and the data of the corresponding store instruction. It is used for *memory disambiguation* (to determine whether a load instruction can be scheduled for execution by comparing the address of the load instruction to the addresses of the older store instructions) and *data forwarding* (to supply the data of a store instruction to a dependent load instruction). To determine whether a load instruction is dependent on an older store, the store buffer needs to be associatively searched using the load instruction's address. Such an associative search requires the store buffer to be implemented as content-addressable memory (CAM) and is therefore very expensive in terms of hardware cost, complexity, power consumption, and access time. As the size of the store buffer is increased, the number of entries participating in the expensive associative search also increases.

Furthermore, the size of the storage required for the data values of stores also needs to increase.

- **Load buffer (load queue):** This structure keeps the ordered list of load instructions in the window. It is used to guarantee the correct ordering of memory references in both uniprocessors and multiprocessors. Modern processors speculatively schedule and execute load instructions before making sure that there is no older store instruction (either from the same thread of execution or from a separate processor) that writes to the same address. The purpose of the load buffer is to check the correctness of this speculative execution. When a store instruction's address is generated, the load buffer is associatively searched using the store address to determine if any younger load that accesses the same address was speculatively issued. Therefore, the load buffer is also implemented as a CAM structure. Increasing its size significantly increases the hardware cost and power consumption.

- **Scheduling window (issue queues or reservation stations):** As described in Section 2.2.1, this structure is used to determine whether an instruction is ready for execution. It is associatively searched every cycle. As the window size increases, the scheduler size also needs to increase to accommodate more instructions. Increasing the size of the scheduler is not an easy task because the complexity and power consumption of the scheduler increases quadratically with the window size [87].

3.3.2.2 The Proposed Solutions

Recent proposals focused on enlarging the size of the described buffers without having to increase the cycle time of the processor. Many of these proposals employ hierarchical buffers in which one level is small and fast, and the next level is large and slow.

Building Large Reorder Buffers

Akkary et al. [3] and Cristal et al. [33, 31] proposed using checkpointing [50] to build large reorder buffers. These proposals replace the conventional reorder buffer with a checkpoint-based structure that retires checkpoints (rather than instructions) in program order. Compared to the previous proposal for checkpointing [50], these proposals take checkpoints infrequently at either low-confidence branches [3] or at long-latency loads and periodically every so many instructions [31] in order to reduce the hardware cost associated with checkpoints. Although these proposals can increase the size of the reorder buffer, they do not address the increases in design and verification complexity and power consumption associated with supporting a large number of in-flight instructions.

The advantage of these approaches is that they allow information to be kept on a per-checkpoint basis instead of per-instruction basis, which reduces the bookkeeping information tracked by the processor. Also, a physical register allocated to an architectural register that is overwritten within the same checkpoint can be deallocated early if all instructions that need the value have already sourced the register (similarly to the previously proposed early register release mechanism [73]). However, this introduces extra control complexity because the processor needs to always correctly track the number of readers of each physical register even in the presence of branch mispredictions and exceptions.

The checkpoint-based retirement approach also has several performance-related disadvantages. First, if any of the instructions associated with a checkpoint causes an exception or a branch misprediction, the processor recovers to the beginning of the checkpoint and executes instructions one by one in order to recover to a precise architectural state. This results in a performance loss if checkpoints are taken infrequently. Second, the physical registers cannot be freed on a per-instruction basis. Rather, in order to free a physical register, the checkpoint that overwrites the architectural register that maps to the physical register needs to be committed. This increases the lifetime of physical reg-

isters and results in a performance degradation over a processor that performs instruction retirement and physical register deallocation on a per-instruction basis [31].

Building Large Register Files

Cruz et al. [34] proposed using a two-level register file [112] to keep the access time of the register file small while providing storage for a large number of registers. A small register file cache is used at the first level to provide fast access to recently-used registers. The second level consists of a large and slow backup register file that is accessed when there is a miss in the register file cache. Register file caches were previously proposed for in-order processors that support a large number of architectural registers [128].

The disadvantage of a two-level register file is the performance degradation incurred on a register file cache miss. Cruz et al. [34] showed that this approach degrades IPC by 10% on SPEC CPU95 integer benchmarks compared to a large monolithic register file. Furthermore, already-scheduled instructions dependent on a result that misses in the register file cache need to be re-scheduled, which increases the complexity of the scheduling window. Extra complexity is also introduced to manage the two levels of the register file. Research on designing two-level register files is ongoing with a focus on improved and less complex management of the two levels [7, 13, 14].

Building Large Load/Store Buffers

The design of large and scalable load/store buffers has recently received considerable attention from researchers. Akkary et al. [3] proposed a two-level store buffer organization to allow a large number of in-flight store instructions without degrading the cycle time. Two-level store buffers are conceptually similar to the two-level register files. The first level is a small and fast CAM structure that holds the youngest few store instructions in the window. The slow and large second-level CAM structure holds the remaining store instructions. The disadvantage of this mechanism is the extra latency incurred to access

the second level when a load requires the data of a store that resides in the second level. Moreover, the scheme introduces extra complexity in the design of the store buffer because control mechanisms need to be provided to manage the two levels in the store buffer hierarchy.

Sethumadhavan et al. [98] proposed using a Bloom filter to reduce the associative searches in the load/store buffers. This reduces the power consumption of the load/store buffers, but does not reduce the size and the complexity of the store buffer. Sethumadhavan et al. also showed that it is possible to eliminate the load buffer using a conservative Bloom Filter. However, this design causes a significant performance loss (34% on an aggressive microarchitecture) even though it simplifies the design. Similarly, Cain and Lipasti [15] proposed eliminating the load buffer. Their scheme re-executes every load instruction prior to retirement in program order to ensure correct ordering between memory operations. If it is determined that the load instruction got the wrong data in its speculative execution, the instructions dependent on the load are re-executed. Even though this scheme eliminates the load buffer, it causes every load instruction to be executed at least twice, which significantly increases energy consumption and possibly degrades performance.

Sha et al. [99] recently proposed a promising approach to reduce the complexity of the store buffer. Their mechanism eliminates the associative search of the store buffer. When a load is fetched, it accesses a memory dependence predictor [72, 25] to predict whether or not it is dependent on a store. The memory dependence predictor supplies the index of the store buffer entry the load is predicted dependent on. This index is later used by the load instruction to access the store buffer, which is organized as a RAM structure. Load instructions are re-executed before they are retired to verify the prediction made by the memory dependence predictor. The disadvantage of this scheme is that it causes the re-execution of load instructions, which increases energy consumption and contention for the data cache ports.

Building Large Scheduling Windows

Lebeck et al. [63] proposed a scheduler design that requires only a small number of entries to be associatively searched every cycle for ready instructions. All instructions are initially inserted into this small scheduler when they are issued. If a load instruction incurs a long-latency cache miss, it and the instructions that depend on it are removed from the scheduler and inserted into a large *waiting instruction buffer (WIB)*. This creates space in the small scheduler for other instructions independent of the long-latency miss and allows them to be scheduled in the presence of long-latency cache misses. Hence, long-latency cache misses cannot block the scheduling window. When the long-latency load instruction is serviced, it and its dependent instructions are re-inserted into the scheduling window so that they are scheduled and executed. This approach, which is conceptually similar to the *replay scheduling* implemented in the Pentium 4 [12], allows the performance of a large scheduler without enlarging the CAM-based scheduling window. However, to be effective it requires a large instruction window with its associated cost (large register files, load/store buffers, and reorder buffer) since instructions that stay in the WIB still need to hold on to their entries in the physical register files, load/store buffers, and reorder buffer.

Srinivasan et al. [109] improved the WIB design by deallocating the registers allocated to instructions that are moved to the WIB (called Slice Data Buffer (SDB) in [109]) from the scheduler. When the long-latency cache miss is serviced, new registers are allocated to the instructions in the SDB. Later, the instructions are re-inserted into the scheduler. With this scheme, L2-miss dependent instructions do not occupy scheduler or register file entries while the L2 miss they are dependent on is in progress. However, the L2-miss dependent instructions still occupy entries in the SDB and the load/store buffers. Moreover, structures needed to deallocate and re-allocate registers, as well as structures needed to guarantee deadlock-free forward progress introduce additional control complexity into the processor pipeline.

3.3.2.3 Putting It All Together

Two recent papers [109, 32] proposed using large instruction windows by combining the previously-proposed mechanisms for building large processor buffers. These proposals combine mechanisms for building large processor buffers (described in Section 3.3.2.2) and mechanisms for efficiently utilizing those buffers (described in Section 3.3.1) to support a large number of in-flight instructions.

Even though the idea of building a large instruction window is promising and techniques to build large processor buffers are worthy of research, previous research that combined these mechanisms has not shown that implementing an instruction window that supports a large number of instructions is possible without unreasonably increasing the design complexity, the verification complexity, and both the static and dynamic energy consumption of current processors.[1] The proposed solutions still require very large buffers, albeit perhaps less complex than conventional ones, in the processor core. They also require non-trivial control logic to manage the proposed hierarchical structures, which increases the design and verification complexity of the processor. Therefore, it is not clear that enlarging the instruction window to accommodate a large number of instructions is easy to accomplish even when the previously described mechanisms are used.

Finally, the proposed mechanisms for building large processor buffers invariably result in an IPC performance loss compared to an idealized large instruction window which has large, monolithic structures. For example, Cristal et al. [31] showed that combining the proposed mechanisms for building a large reorder buffer using checkpointing and a large scheduling window using a scheme similar to the WIB (while still having large, monolithic

[1] By "a large number of instructions," we mean a number of instructions that is significantly larger than what is supported by today's aggressive out-of-order processors. Today's processors support between 100-128 in-flight instructions (e.g., Intel Pentium 4 [48] supports 126 and IBM POWER4 [113] supports 100). By this definition, we consider a window that can hold at least 200 instructions a large window.

structures for the register files and load/store buffers) results in an IPC loss of 10% compared to a 4096-entry instruction window with a monolithic 4096-entry reorder buffer and a monolithic 4096-entry scheduler.[2]

Large Instruction Windows vs. Runahead Execution

This dissertation proposes a cost- and complexity-effective alternative to large instruction windows. The purpose of our research is to design a simple mechanism that can provide the latency tolerance provided by a large instruction window without having to design and implement structures that support a large number of in-flight instructions. In contrast to the proposals surveyed in the previous section, the runahead execution implementation described in Section 2.5 adds very little hardware cost and complexity to an existing out-of-order execution processor.

A detailed performance comparison between large instruction windows and runahead execution is presented in Sections 4.1.3.3 and 4.2.5.3. The performance of runahead execution is compared to that of large instruction windows for both an x86 processor modeled after the Intel Pentium 4 and a near-future processor model implementing the Alpha ISA. The evaluations of large instruction windows do not assume a particular implementation. Rather, they model monolithic register files, load/store buffers, scheduling window, and reorder buffer, all of which are the same size as the evaluated instruction window. Therefore, the performance loss due to a particular proposed implementation of these buffers is not factored in our experiments, which biases the performance results in favor of the large window processors. Even so, the results presented in this dissertation show that a runahead execution processor with a 128-entry instruction window can achieve the same performance as a conventional out-of-order processor with a 384-entry instruction window

[2]Their experiments assumed a memory latency of 1000 cycles. For a comparison of the performance of runahead execution and large instruction windows at a 1000-cycle memory latency, see Section 4.2.5.3.

when the memory latency is 500 cycles. When the memory latency is increased to an expected near-future latency of 1000 cycles, a runahead processor with a 128-entry window can achieve the same performance as a conventional processor with a 1024-entry window.

3.4 Related Research in Enhanced In-order Processing

In-order execution is unable to tolerate the latency of cache misses. To increase the memory latency tolerance of in-order execution, two schemes have been proposed.

Runahead processing [36] was first proposed and evaluated as a method to improve the data cache performance of a five-stage pipelined in-order execution machine. It was shown to be effective at tolerating long-latency data cache and instruction cache misses on an in-order processor that does not employ any hardware prefetching techniques [36, 37]. Unfortunately, runahead processing on an in-order execution machine suffers from the major disadvantage of in-order execution: the inability to tolerate the latency of multi-cycle operations, such as first-level cache misses and floating point operations. For this reason, the performance of an in-order runahead processor is significantly worse than an out-of-order processor especially when the performance of a program is execution-bound rather than memory bound, as we show in Section 4.2.5.4. In our research, we aim to improve the performance of the higher-performance out-of-order processors, which are the state-of-the-art in high-performance computing. We provide a detailed performance comparison of runahead execution on in-order and out-of-order processors in Sections 4.1.3.8 and 4.2.5.4.

Barnes et al. [9] proposed two-pass pipelining to reduce the stalls caused by data cache misses in an in-order processor. In their proposal, the execution pipeline of an in-order processor is replicated. Instructions that miss in the data cache do not stall the first pipeline. Instead, they and their dependents are *deferred* to the second pipeline. While instructions dependent on cache misses are executed in the second pipeline, independent

instructions can continue to be executed in the first pipeline. This design increases the tolerance of an in-order processor to cache misses. There are three disadvantages of two-pass pipelining. First, it requires a replicated back-end execution pipeline and auxiliary structures, which increases the complexity of the processor. Second, the performance of this design is limited because there is no way to avoid stalls in the second pipeline even if the instructions that are being processed in the second pipeline are independent of each other. Hence, an out-of-order processor would have higher performance than two-pass in-order execution. Third, the tolerance of the two-pass design to long main memory latencies is limited by the size of structures that are on the critical path. To tolerate long main memory latencies, the complexity of the two-pass pipeline would need to increase significantly.

Barnes et al. [10] recently proposed an improvement over their two-pass pipelining scheme, called multipass pipelining. This scheme eliminates the replication of the back-end execution pipeline and allows an in-order processor to make multiple passes of execution over instructions following a data cache miss. The number of instructions that are processed in the shadow of a data cache miss is limited by the size of a structure called *Instruction Queue*. Therefore, the latency that can be tolerated by a multipass processor is limited by the *Instruction Queue* size.[3] Barnes et al. showed that, on binaries aggressively optimized for an in-order processor and with a relatively short 145-cycle memory latency, an in-order processor augmented with multipass pipelining can achieve 77% of the performance benefit of out-of-order execution relative to in-order execution. However, they did not show performance comparisons to an out-of-order processor incorporating runahead execution. Our evaluations in Section 4.2.5.4 show that an out-of-order processor with runahead execution significantly outperforms both a conventional out-of-order processor

[3]Note that it is possible to combine runahead execution and multipass pipelining to increase the latency tolerance of the multipass pipelining scheme. When the *Instruction Queue* is full, the processor can enter runahead mode until the long-latency cache miss is serviced. This removes the size of the *Instruction Queue* from being a performance bottleneck in the presence of a long-latency cache miss.

and an in-order processor with runahead execution.

3.5 Related Research in Multithreading

Multithreading has been used to tolerate long main memory latencies. A multithreaded machine [114, 102, 89, 85, 49, 118] has the ability to process instructions from several different threads without performing context switches. The processor maintains a list of active threads and decides which thread's instructions to issue into the machine. When one thread is stalled waiting for a cache miss to be serviced, instructions from non-stalled threads can be issued and executed. This improves the overall throughput of the processor.

Multithreading has been shown to improve the overall performance of the processor. However, the performance of each individual thread is not improved by multithreading. In fact, it is likely that the performance of each thread is degraded since resources must be shared between all active threads. Hence, multithreading can tolerate main memory latencies when running a multiprogrammed or multithreaded workload, but provides no benefit on a single-threaded application. The goal of this dissertation is to improve the memory latency tolerance of single-threaded applications or each individual thread in multithreaded applications.

Hardware schemes to parallelize single-threaded applications have been proposed. In multiscalar processors [39], a program's instruction stream is divided into tasks that are executed concurrently on several processing units. Processing units are organized as a ring and values are communicated between different tasks using the ring interconnect. Multiscalar processors can tolerate a long-latency cache miss in one task by executing independent operations in other tasks. Unfortunately, multiscalar processors require significant hardware complexity to communicate register and memory values between different

tasks. Also, how to break a single sequential instruction stream into parallel tasks is a hard problem for many applications.

Dynamic multithreading [2] was also proposed to parallelize a single-threaded application at run time. A dynamic multithreading processor creates threads automatically at procedure and loop boundaries and executes them speculatively on a simultaneous multithreading pipeline. For example, when the processor reaches a call instruction, it spawns a speculative thread that is executed on a separate hardware thread context, starting with the next sequential instruction after the call. The non-speculative parent thread executes the function call whereas the spawned speculative thread does not. A cache miss encountered in the non-speculative thread can be tolerated by executing independent instructions in speculative threads. Dynamic multithreading, however, comes at a significant hardware cost which includes a simultaneous multithreading pipeline, support for spawning speculative threads, support for communicating data values between speculative and non-speculative threads, and support for detecting value mispredictions in speculative threads.

3.6 Related Research in Parallelizing Dependent Cache Misses

Chapter 6 of this dissertation proposes a new mechanism, called *address-value delta (AVD) prediction*, to parallelize dependent cache misses during runahead execution by predicting the values produced by pointer load instructions. Several previous papers focused on predicting the addresses generated by pointer loads for value prediction or prefetching purposes. Most of the previously proposed mechanisms require significant storage cost and hardware complexity. The major contribution of AVD prediction is a simple and efficient low-cost mechanism that allows the prediction of the values loaded by a subset of pointer loads by exploiting stable address-value relationships. This section briefly discusses the related research in value prediction and prefetching for pointer loads.

3.6.1 Related Research in Load Value/Address Prediction

The most relevant work to AVD prediction is in the area of predicting the destination register values of load instructions. Load value prediction [66] was proposed to predict the destination register values of loads. Many types of load value predictors were examined, including last value [66], stride [38, 97], FCM (finite context method) [97], and hybrid [120] predictors. While a value predictor recognizes stable/predictable values, an AVD predictor recognizes stable arithmetic differences (deltas) between the effective address and data value of a load instruction. This dissertation evaluates the performance of AVD prediction in comparison to a state-of-the-art stride value predictor in Section 6.6.6. The analyses provided in that section and Section 6.3 show that AVD prediction provides significant performance improvements over stride value prediction.

Load address predictors [38, 11] predict the effective address of a load instruction early in the pipeline. The value at the predicted address can be loaded to the destination register of the load before the load is ready to be executed. Memory latency can be partially hidden for the load and its dependent instructions.

Complex (e.g., stride or context-based) value/address predictors need significant hardware storage to generate predictions and significant hardware complexity for state recovery. Moreover, the update latency (i.e., the latency between making the prediction and determining whether or not the prediction was correct) associated with stride and context-based value/address predictors significantly detracts from the performance benefits of these predictors over simple last value prediction [92, 64]. Good discussions of the hardware complexity required for complex address/value prediction can be found in [11] and [92].

Pointer caches [26] were proposed to predict the values of pointer loads. A pointer cache caches the values stored in memory locations accessed by pointer load instructions. It is accessed with a load's effective address in parallel with the data cache. A pointer cache hit provides the predicted value for the load instruction. To improve performance, a pointer

cache requires significant hardware storage (at least 32K entries where each entry is 36 bits [26]) because the pointer data sets of the programs are usually large. In contrast to the pointer cache, an AVD predictor stores information based on pointer load instructions. Since the pointer load instruction working set of a program is usually much smaller than the pointer data working set, the AVD predictor requires much less hardware cost. Also, an AVD predictor does not affect the complexity in critical portions of the processor because it is small and does not need to be accessed in parallel with the data cache, as we will explain in Section 6.4.

Zhou and Conte [130] proposed the use of value prediction only for prefetching purposes in an out-of-order processor such that no recovery is performed in the processor on a value misprediction. They evaluated their proposal using a 4K-entry stride value predictor that predicts the values produced by all load instructions. Similar to their work, we employ the AVD prediction mechanism only for prefetching purposes, which eliminates the need for processor state recovery. In contrast to their work, we propose a new prediction mechanism that requires little hardware cost by predicting only the values generated by loads that lead to the address generation of dependent loads rather than predicting the values generated by all loads.

3.6.2 Related Research in Pointer Load Prefetching

In recent years, substantial research has been performed in prefetching the addresses generated by pointer load instructions. AVD prediction differs from pointer load prefetching in that it is *more than just a prefetching mechanism*. As we will show in Section 6.6.7, AVD prediction can be used for simple prefetching. However, AVD prediction is more beneficial when it is used as a targeted value prediction technique for pointer loads that enables the pre-execution of dependent load instructions, which may generate prefetches.

Hardware-based pointer prefetchers [21, 55, 94, 95, 26] try to dynamically cap-

ture the prefetch addresses generated by loads that traverse linked data structures. These approaches usually require significant hardware cost to store a history (trace)of pointers. For example, hardware-based jump pointer prefetching requires jump pointer storage that has more than 16K entries (64KB) [95]. Compared to hardware-based pointer prefetchers, AVD prediction has much less hardware overhead and complexity. A low-overhead content-based hardware pointer prefetcher was proposed by Cooksey et al. [29]. It can be combined with AVD prediction to further reduce the negative performance impact of dependent L2 cache misses.

Software and combined software/hardware methods have also been proposed for prefetching loads that access linked data structures [65, 68, 95, 125, 105, 22, 123, 1]. Of these techniques, the one most relevant to AVD prediction is MS Delta [1], which is a purely software-based prefetching technique. MS Delta uses the garbage collector in a run-time managed Java system to detect regular distances between objects in linked data structures whose traversals result in significant number of cache misses. A just-in-time compiler inserts prefetch instructions into the program using the identified regular distances in order to prefetch linked objects in such traversals. Such software-based prefetching techniques require non-trivial support from the compiler, the programmer, or a dynamic optimization and compilation framework. Existing binaries cannot utilize software-based techniques unless they are re-compiled or re-optimized using a dynamic optimization framework. AVD prediction, on the contrary, is a purely hardware-based mechanism that does not require any software support and thus it can improve the performance of existing binaries. However, as we will show in Section 6.7.2, AVD prediction can provide larger performance improvements if software is written or optimized to increase the occurrence of stable AVDs.

3.7 Recent Related Research in Runahead Execution

After the publication of our initial papers on runahead execution [83, 84], there has been a vibrant interest in the computer architecture community on techniques building on runahead execution. This section briefly discusses such related research.

Chou et al. [23] evaluated runahead execution on large commercial database applications on an out-of-order execution processor implementing the SPARC ISA. They showed that runahead execution significantly improves memory-level parallelism [42] and performance on such applications because it eliminates the instruction and scheduling windows, as well as serializing instructions, from being performance bottlenecks in the presence of long memory latencies. Chou et al. [24] also recently showed that runahead execution can significantly improve the memory-level parallelism of store operations in commercial database applications.

Iacobovici et al. [51] evaluated the interaction of runahead execution with different stream-based prefetching techniques. They showed that runahead execution and stream-based prefetching techniques interact positively when used together. Recently, Ganusov and Burtscher [40] also showed that runahead execution interacts positively with both stream-based prefetchers and an execution-based prefetching scheme they developed.

Three recent papers [23, 59, 17] combined runahead execution with value prediction. Kirman et al. [59] and Ceze et al. [17] proposed a mechanism in which L2-miss load instructions are value-predicted in runahead mode. If the predictions for those instructions are correct the processor does not need to re-execute the instructions executed in runahead mode when it returns to normal mode. While this approach requires more hardware than the baseline runahead execution mechanism, it may improve performance on benchmarks where the values of L2-miss loads are predictable. This dissertation evaluates the value prediction of L2-miss instructions in Section 5.5.2.

Zhou [129] proposed *dual-core execution*, in which one processor performs runahead execution for another in a dual-core chip multiprocessor with an L2 cache shared by the cores. In Zhou's proposal, two cores in a chip multiprocessor are connected using a FIFO queue that transmits instructions from the *front* processor to the *back* processor. The front processor is a purely speculative processor that enters runahead mode on an L2 cache miss and speculatively prefetches data into the shared L2 cache. The back processor is a conventional out-of-order processor that processes instructions it reads from the FIFO queue and stalls when it encounters an L2 miss. The advantage of this mechanism is that the front processor can continuously stay ahead of the back processor (similar to slipstream processors [111]) and provide prefetching benefits to the back processor, which is responsible for updating the architectural state of the program. This can result in more effective prefetching than what can be accomplished with using runahead execution on a single processing core during the cycles spent waiting for L2 misses. The disadvantage is that the proposed mechanism requires an extra processor that speculatively executes all instructions in the program. Hence, the hardware cost and the energy consumption of the dual-core execution mechanism can be high.

Chapter 4

Performance Evaluation of Runahead Execution

This chapter evaluates the performance of the runahead execution processor described in Chapter 2. The performance of runahead execution is evaluated on processors implementing two different instruction set architectures (ISAs), the x86 ISA [52] and the Alpha ISA [101]. Section 4.1 presents the evaluation methodology and results for the x86 ISA and Section 4.2 presents the evaluation methodology and results for the Alpha ISA.

4.1 Evaluation of Runahead Execution on an x86 ISA Processor

The evaluation of runahead execution on the x86 ISA is performed using a detailed cycle-accurate microarchitecture simulator and a wide variety of memory-intensive benchmarks provided by Intel.

4.1.1 Simulation Methodology and Benchmarks

We used a simulator that was built on top of a micro-operation (uop) level x86 architectural simulator that executes Long Instruction Traces (LIT). A LIT is not a trace in the conventional sense of the term, but a checkpoint of the processor state, including memory, that can be used to initialize an execution-driven performance simulator. A LIT also includes a list of *LIT injections*, which are system interrupts needed to simulate events like DMA. Since the LIT includes the entire snapshot of memory, the simulator can simulate both user and kernel instructions, as well as wrong-path instructions.

Table 4.1 shows the benchmark suites used for evaluation. These benchmark suites are selected from a set of 280 LITs that were carefully chosen for the design and evaluation of the Intel Pentium 4 processor [48, 12]. We evaluated the performance of runahead execution on LITs that gain at least 10% IPC improvement with a perfect L2 cache compared to our baseline model. In all, there are 80 benchmarks, comprising 147 LITs. Each LIT is 30 million x86 instructions long and it is carefully selected to be representative of the overall benchmark. Unless otherwise stated, all average IPC performance values (in retired uops per cycle) reported in this section are harmonic averages over all 147 LITs.

Suite	Number of Benchmarks	Number of Traces	Benchmarks
SPEC CPU95 (S95)	10	10	vortex + all SPEC CPU95 FP except fppp [110]
SPECfp2K (FP00)	11	18	wupwise, swim, mgrid, applu, galgel, equake, facerec, ammp, lucas, apsi [110]
SPECint2K (INT00)	6	9	vpr, gcc, mcf, parser, vortex, twolf benchmarks from SPEC CPU2000 integer suite [110]
Internet (WEB)	18	32	specjbb [110], webmark2001 [8], volanomark [119], flash animation
Multimedia (MM)	9	16	MPEG encoder/decoder, speech recognition, quake game, 3D visualization
Productivity (PROD)	17	31	sysmark2k [8], winstone [131], Microsoft Office applications (word, excel, ppt)
Server (SERV)	2	17	TPC-C [117], timesten [115]
Workstation (WS)	7	14	computer-aided design applications (e.g., nastran), functional and gate-level Verilog simulation

Table 4.1: Simulated benchmark suites for the x86 processor.

4.1.2 Baseline Microarchitecture Model

The performance simulator is an execution-driven cycle-accurate simulator that models a superscalar out-of-order execution microarchitecture similar to that of the Intel Pentium 4 microprocessor. An overview of the baseline machine is provided in Section 2.5.1. More details on the baseline Pentium 4 microarchitecture can be found in Hinton

et al. [48] and Boggs et al. [12].

We evaluate runahead execution for two baselines. The first is a 3-wide machine with microarchitecture parameters similar to the Pentium 4, which we call the *current baseline*. The second is a more aggressive 6-wide machine with a pipeline twice as deep and buffers four times as large as those of the current baseline, which we call the *future baseline*. Figure 4.1 shows the minimum branch misprediction pipeline of the current baseline. Table 4.2 provides the machine parameters for both baselines.

Stage	Cycles	Summary
Fetch	5	generate next PC, access trace cache, read decoded uops
Allocate	3	allocate resources (window, load/store buffer, register file entries) for uops
Rename	5	compute source and destination register tags
Steer	2	steer instructions to clusters and insert them into respective queues
Drive	1	issue instructions into the scheduler
Schedule	3	find and select ready instructions for execution
Dispatch	1	send scheduled instructions to register file and functional units
RF Read	5	read the physical register file to fetch source operands
Execute	1	execute instructions in functional units
Br Check	1	compare branch results to predictions
Recover	2	send misprediction signal and re-direct the fetch engine
Total	29	

Figure 4.1: Minimum branch recovery pipeline of the current baseline.

The baseline machine models include a detailed model of the memory subsystem that faithfully models buses and bus contention. Bandwidth, port contention, bank conflicts, and queuing effects are faithfully modeled at all levels in the memory hierarchy. Both baselines include an aggressive stream-based hardware prefetcher, which is described in [48] and [53]. Unless otherwise noted, all performance results using the x86 processor are relative to a baseline using this prefetcher, which will be referred to as HWP. In addition, the baseline models include a hardware instruction prefetcher that prefetches the next two cache lines when an instruction fetch request misses in the L2 cache.

PARAMETER	CURRENT	FUTURE
Processor frequency	4 GHz	8 GHz
Fetch/Issue/Retire width	3	6
Branch misprediction penalty	29 stages	58 stages
Instruction window size	128	512
Scheduling window size	16 int, 8 mem, 24 fp	64 int, 32 mem, 96 fp
Load and store buffer sizes	48 load, 32 store	192 load, 128 store
Physical register file size	136 int, 136 fp	136 int, 136 fp
Functional units	3 int, 2 mem, 1 fp	6 int, 4 mem, 2 fp
Conditional branch predictor	1K-entry, 32-bit history perceptron predictor [54]	3K-entry, 32-bit history perceptron predictor
Indirect branch predictor	1K-entry, tagged last-target predictor	1K-entry tagged last-target predictor
Hardware data prefetcher	Stream-based (16 streams)	Stream-based (16 streams)
HWP prefetch distance	16 cache lines ahead	32 cache lines ahead
Instruction prefetcher	next-two-lines prefetch	next-two-lines prefetch
Trace cache	12k uops, 8-way	64k uops, 8-way
Memory disambiguation	Perfect	Perfect
Memory Subsystem		
L1 Data Cache	32 KB, 8-way, 64-byte line size	64 KB, 8-way, 64-byte line size
L1 Data Cache Hit Latency	3 cycles	6 cycles
L1 Data Cache Bandwidth	512 GB/s, 2 load/store accesses/cycle	4 TB/s, 4 load/store accesses/cycle
Max In-flight L1 Misses	64	256
L2 Unified Cache	512 KB, 8-way, 64-byte line size	1 MB, 8-way, 64-byte line size
L2 Unified Cache Hit Latency	16 cycles	32 cycles
L2 Unified Cache Bandwidth	128 GB/s, 1 access/cycle	256 GB/s, 1 access/cycle
Max In-flight L2 Misses	32	128
Main Memory Latency	495 processor cycles	1008 processor cycles
Bus Bandwidth	4.25 GB/s	8.5 GB/s
Max Pending Bus Transactions	10	20

Table 4.2: Processor parameters for current and future baselines.

4.1.3 Performance Results on the x86 Processor

4.1.3.1 Runahead Execution vs. Hardware Data Prefetcher

We first evaluate how runahead execution performs compared to the stream-based hardware prefetcher. Figure 4.2 shows the IPC of four different machine models. For each suite, bars from left to right correspond to:

1. A model with no prefetcher and no runahead (current baseline without the prefetcher),
2. A model with stream-based prefetcher, but without runahead (current baseline)
3. A model with runahead but no prefetcher,
4. A model with stream-based prefetcher and runahead (current baseline with runahead).

Percentage numbers on top of the bars is the IPC improvement of runahead execution (model 4) over the current baseline (model 2).

The model with only runahead outperforms the model with only HWP for all benchmark suites except for SPEC95. This means that runahead is a more effective prefetching scheme than the stream-based hardware prefetcher for most of the benchmarks. Overall, the model with only runahead (IPC:0.58) outperforms the model with no HWP or runahead (IPC:0.40) by 45%. It also outperforms the model with only the HWP (IPC:0.52) by 12%. But, the model that gets the best performance is the one that leverages both the hardware prefetcher and runahead (IPC:0.64). This model has 58% higher IPC than the model with no HWP or no runahead. It has 22% higher IPC than the current baseline. This IPC improvement over the current baseline ranges from 52% for the Workstation suite to 12% for the SPEC95 suite.

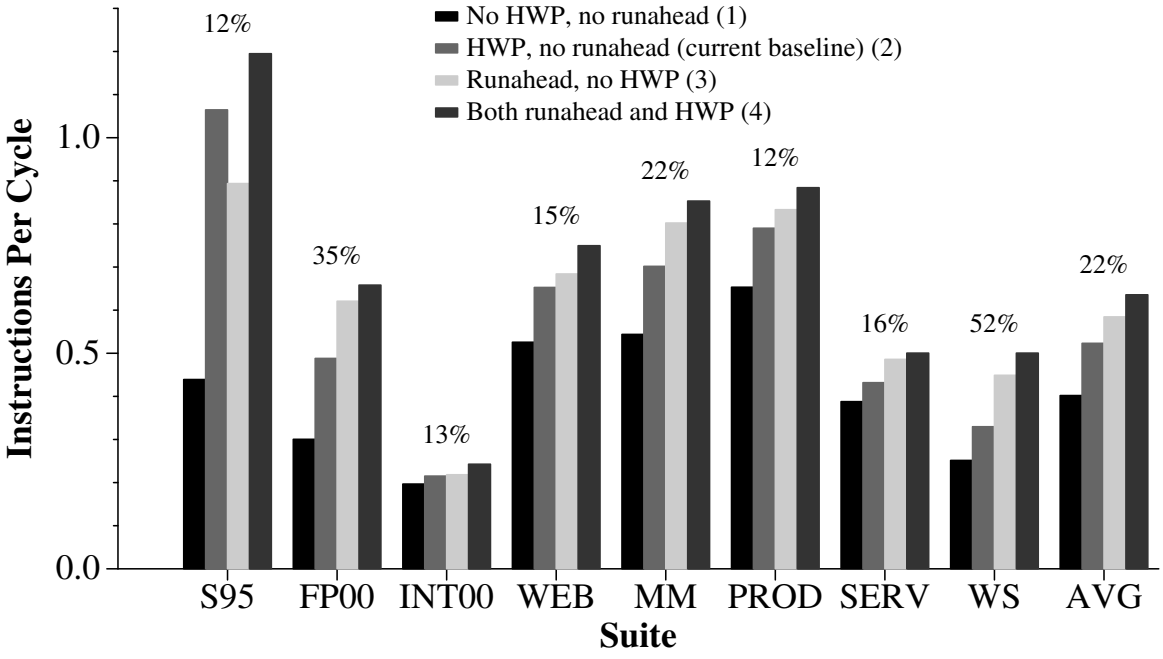


Figure 4.2: Runahead execution performance on the current baseline. Percentage numbers show the performance improvement of model (4) over model (2).

These results show that runahead execution provides significant benefits over an aggressive stream-based prefetcher. Furthermore, runahead execution and the HWP interact positively. One advantage of runahead execution is that it can capture irregular cache miss patterns with no predictable stride as well as predictable striding patterns. Therefore, runahead execution provides higher performance improvements than the HWP for all benchmark suites except SPEC95. The SPEC95 suite consists mainly of floating-point benchmarks whose access patterns are predominantly striding. The specialized stream-based prefetcher is therefore more efficient and effective at capturing those patterns than runahead execution.

4.1.3.2 Interaction Between Runahead Execution and the Hardware Data Prefetcher

It is worthwhile to examine the interaction between runahead execution and the hardware data prefetcher. If runahead execution is implemented on a machine with a hardware prefetcher, the prefetcher tables can be trained and new prefetch streams can be created while the processor is in runahead mode. Thus, runahead memory access instructions not only can generate prefetches for the data they need, but also can trigger hardware data prefetches. These triggered hardware data prefetches, if useful, would likely be initiated much earlier than they would be on a machine without runahead execution. We found that if the prefetcher is accurate, using runahead execution on a machine with a prefetcher usually performs best. However, if the prefetcher is inaccurate, it may degrade the performance of a processor implementing runahead execution.

Prefetches generated by runahead instructions are inherently quite accurate because these instructions are likely on the program path. There are no traces whose performance is degraded due to the use of runahead execution. The IPC improvement of runahead execution ranges from 2% to 401% over a baseline that does not have a prefetcher for all traces simulated. This range is from 0% to 158% over a baseline with the stream-based prefetcher.

This section examines the behavior of some applications that demonstrate different patterns of interaction between runahead execution and the hardware data prefetcher. Figure 4.3 shows the IPCs of a selection of SPECfp2k and SPECint2k benchmarks on four models. The number on top of each benchmark denotes the percentage IPC improvement of model 4 over model 2. For gcc, mcf, vortex, mgrid, and swim, both runahead execution and the hardware prefetcher alone improve the IPC. When both are combined, the performance of these benchmarks is better than their performance if runahead execution or the hardware prefetcher is used alone. Especially in mgrid, the hardware prefetcher generates very accurate prefetches during runahead mode. Therefore, the IPC of mgrid on a machine

with runahead execution and prefetcher (IPC:1.28) is 13% better than it is on a model with just runahead (IPC:1.13).

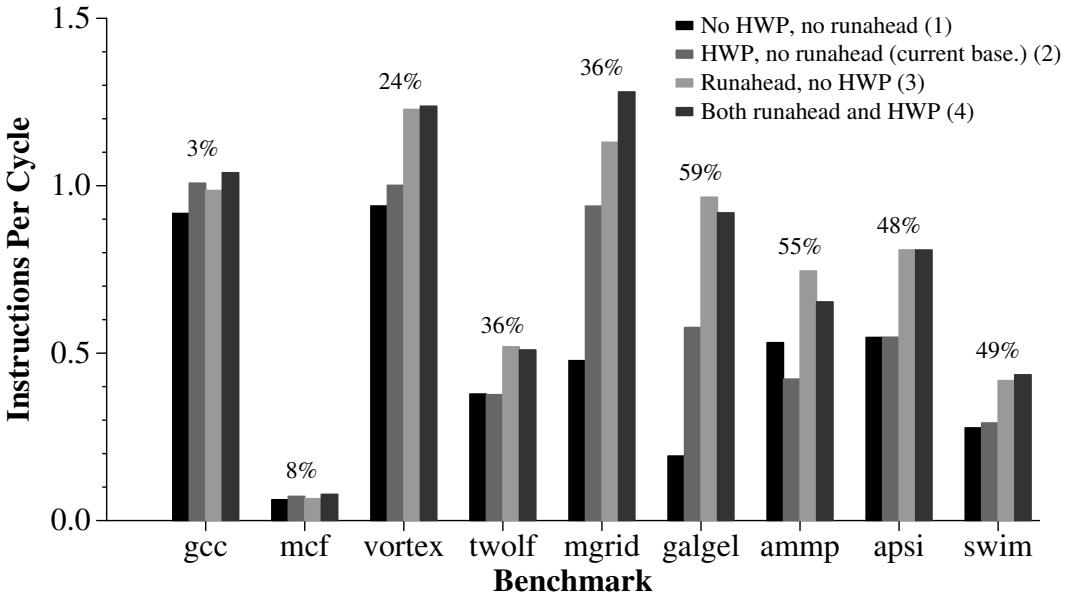


Figure 4.3: Hardware prefetcher-runahead execution interaction. Percentage numbers show the performance improvement of model (4) over model (2).

Sometimes, having a prefetcher on a machine that implements runahead execution degrades performance. Galgel, twolf, and ammp are examples of this case. For galgel, although the hardware prefetcher is accurate, it degrades performance on a machine with runahead execution because the extra traffic generated by the prefetcher causes contention for L2 cache and bus bandwidth. For twolf, the hardware prefetcher causes bandwidth contention by sending useless prefetches. For ammp, the hardware prefetcher sends out quite inaccurate prefetches, which causes the IPC of a machine with runahead and the streaming prefetcher to be 12% less than that of a machine with runahead and no prefetcher.

The performance of a runahead execution processor can be improved by optimizing the interaction between runahead execution and the HWP in order to reduce the performance loss in cases described above. We examine such optimizations in Section 5.3.3.

4.1.3.3 Runahead Execution vs. Large Instruction Windows

This section shows that, with the prefetching benefit of runahead execution, a processor can attain the performance of a machine with a larger instruction window without requiring the large, slow, and power-hungry buffers that are needed to implement a large-window processor.

Figure 4.4 shows the IPCs of six different processors for each benchmark suite. From left to right, the first bar shows the IPC of the baseline processor. The next bar shows the IPC of the baseline with runahead execution. The other four bars show the IPC's of processors without runahead and instead with 256, 384, 512, and 1024-entry instruction windows, respectively. The sizes of all other processor buffers (register files, load/store queues, scheduling window) of these evaluated large-window machines are scaled proportionally to the increase in the instruction window size to support the larger number of in-flight instructions. The percentages on top of the bars show the IPC improvement of runahead execution over the machine with 256-entry window.

On average, runahead execution on the 128-entry window baseline has an IPC 3% greater than a model with a 256-entry window. Also, it has an IPC within 1% of that of a machine with a 384-entry window. That is, runahead execution on a 128-entry window processor attains almost the same IPC as a processor with three times the instruction window size. For two suites, SPECint2k and Workstation, runahead on current baseline performs 1-2% better than the model with the 1024-entry window because the performance of these suites is limited mainly by L2 cache misses.

For SPEC95, runahead execution on the 128-entry window has 6% lower IPC than the machine with the 256-entry window. This is because the SPEC95 suite mostly contains floating-point benchmarks that execute long-latency FP operations. A 256-entry window can tolerate the latency of those operations better than a 128-entry one. Runahead execution, with its current implementation, does not offer any solution to tolerating the latency

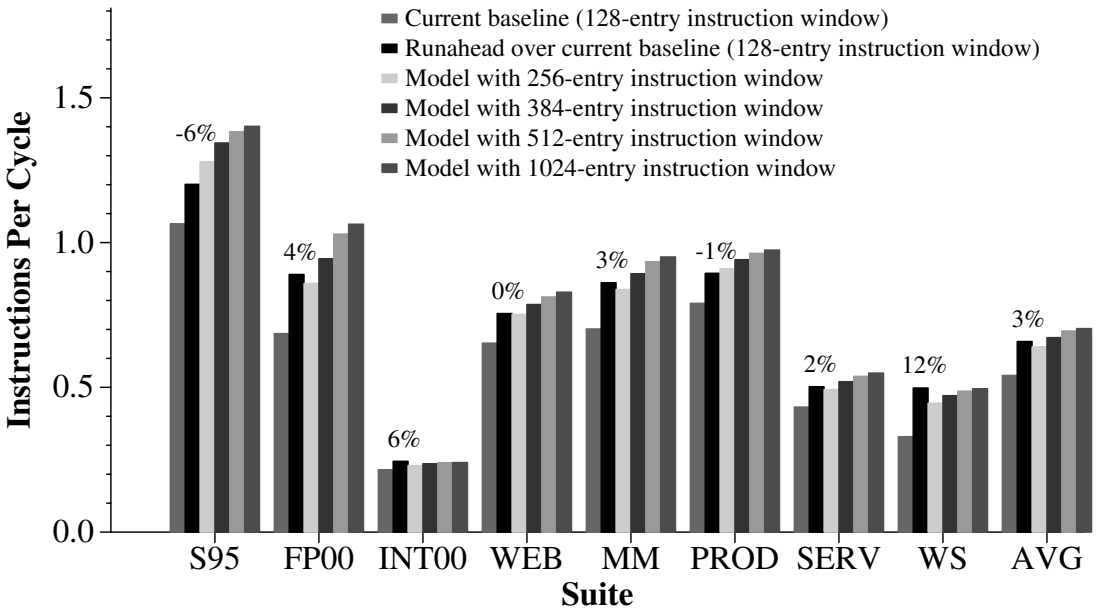


Figure 4.4: Performance of runahead execution versus processors with larger instruction windows. Percentage numbers show the performance improvement of the model with runahead execution and 128-entry window over the model with 256-entry window.

of such operations. Note that the behavior of SPECfp2k suite is different. For SPECfp2k, runahead performs 2% better than the model with 256-entry window because traces for this suite are more memory-limited than FP-operation limited.

Note that the probable increase in the cycle time and the almost inevitable increase in pipeline depth due to increased instruction window size are not accounted for in the performance results presented in this section. If these increases are taken into account the performance of processors with larger instruction windows would be lower. Furthermore, the processor buffers whose sizes are scaled with a larger window are kept as monolithic structures in our performance model. Implementing large cycle-critical monolithic structures in a real processor is likely not feasible due to cycle-time constraints. If these structures (especially the register files and load/store buffers) were implemented as hierarchical two-level structures (as suggested by previous researchers [34, 3]), the performance of larger

instruction windows would be lower. For these reasons, the performance results presented in this section favor large instruction windows.

Qualitative Comparison of Runahead Execution and Large Instruction Windows

The instruction processing model of a runahead execution processor differs from that of a processor with a fixed-size, large instruction window in three major aspects:

1. A runahead processor provides the prefetching benefits of a large instruction window only when an L2 cache miss is in progress. If no L2 cache miss is in progress, a runahead processor is the same as a processor with a small instruction window. In contrast, a large-window processor can provide latency tolerance for relatively shorter-latency operations than L2 misses. For example, increasing the size of the instruction window by implementing larger processor buffers increases a processor's latency tolerance to floating-point operations and L1 cache misses that hit in the L2 cache. However, if we instead implemented runahead execution, the latency tolerance to such operations would not increase. The effects of this can be seen in Figure 4.4 by examining the performance of SPEC95 and Productivity suites. For these two suites, increasing the window size to 256 performs better than adding runahead execution to a processor with a 128-entry window. In SPEC95 and Productivity suites, a larger window increases the latency tolerance to respectively floating-point operations and L1 misses that hit in the L2 cache, whereas runahead execution does not.

2. The number of instructions that are speculatively executed in runahead mode is not limited by the fixed instruction window size in a runahead processor. In contrast, the number of instructions that are executed in a processor with a large window while an L2 cache miss is in progress is limited by the fixed instruction window size. In the presence of long memory latencies, an unrealistically large instruction window

is needed so that the large-window processor does not stall. Therefore, a runahead execution processor can provide better latency tolerance to long L2-miss latencies than a large-window processor by providing the ability to look farther ahead in the program to discover later L2 cache misses. Our analysis shows that this effect is salient in SPECint2k and Workstation suites shown in Figure 4.4. For these two suites, runahead execution on a 128-entry window provides higher performance than a processor with a 1024-entry window because runahead mode can execute more than 1024 instructions under an L2 cache miss and thus can generate more L2 cache misses than what is possible with the fixed-size 1024-entry instruction window.

3. A runahead processor requires a pipeline flush at the end of runahead mode. After the pipeline flush, the runahead processor re-executes some of the instructions executed in runahead mode. No such flush or re-execution happens in a processor with a large instruction window. As discussed in Section 2.5.5.1, this flush does not necessarily result in a performance disadvantage for the runahead processor.

In terms of hardware cost and complexity, a large-window processor has significantly higher hardware cost than a runahead execution processor with a small window. Runahead execution does not add large structures in the processor core, nor does it increase the size of the structures that are on the critical path of execution. In contrast, a large-window processor requires the size of the cycle-critical structures to be increased proportionally to the instruction window size.

Table 4.3 shows that a runahead execution processor with a 128-entry instruction window can provide 92.5% of the IPC performance of a processor with a 1024-entry window and 85.7% of the performance of a processor with a 4096-entry window. However, the hardware required to add runahead execution to a 128-entry window is very small as described in Section 2.5. In contrast, a 1024-entry window requires eight times the load/store

256-entry window	384-entry window	512-entry window	1024-entry window	2048-entry window	4096-entry window	infinite-entry window
107.1%	98.9%	94.8%	92.5%	88.9%	85.7%	77.8%

Table 4.3: Percentage of the performance of a larger-window processor that is achieved by a runahead execution processor with a 128-entry instruction window. Data shown is an average over all evaluated benchmarks.

buffer entries, physical registers, and the entries in bookkeeping structures for in-flight instructions that exist in a 128-entry window. As such, runahead execution provides a cost- and complexity-effective alternative that approximates the performance of a large instruction window with a very small increase in hardware cost and complexity.

4.1.3.4 Effect of a Better Frontend

There are two reasons why a more aggressive and effective frontend would increase the performance benefit of runahead execution:

1. A better instruction supply increases the number of instructions executed during runahead mode, which increases the likelihood of the discovery of additional L2 cache misses.

2. A better branch prediction mechanism decreases the likelihood of an INV branch being mispredicted during runahead mode. Therefore, it increases the likelihood of the discovery of correct-path L2 cache misses. In effect, it moves the *divergence point* later in time during runahead.

If an unresolvable mispredicted INV branch is encountered during runahead mode, it does not necessarily mean that the processor cannot generate useful prefetches that are on the program path. The processor can reach a control-flow independent point, after which

it continues on the correct program path again. Although this may be the case, our experimental data shows that it is usually better to eliminate the divergence points. Averaged over all traces, the number of runahead instructions pseudo-retired before the divergence point is 431 per runahead mode entry, and the number of runahead instructions pseudo-retired after the divergence point is 280. The number of useful L2 miss requests generated before the divergence point is 2.38 per runahead mode entry, whereas this number is 0.22 after the divergence point. Hence, far fewer useful memory-to-L2 prefetches are generated after the divergence point than before.

Figure 4.5 shows the performance of runahead execution as the frontend of the machine becomes more ideal. Models with *perfect trace cache* model a machine that never misses in the trace cache and whose trace cache supplies the maximum instruction fetch bandwidth possible. In this model, the traces are formed using a real branch predictor.

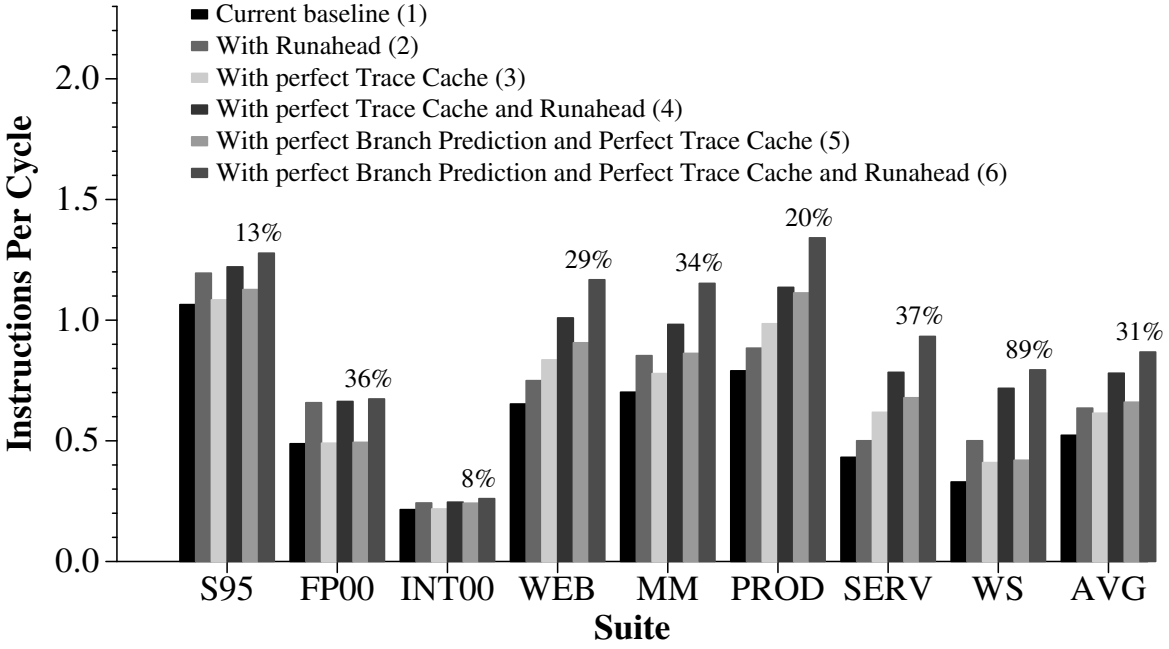


Figure 4.5: Performance of runahead execution as the frontend of the machine is idealized.
Percentage numbers show the performance improvement of model (6) over model (5).

As the frontend of the machine becomes more ideal, the performance improvement of runahead execution increases. For all suites except for SPECint2k, IPC improvement between bars 5 and 6 (as shown in percentages above the bars) is larger than the IPC improvement between bars 1 and 2. As mentioned before, runahead execution improves the IPC of current baseline by 22%. Runahead on the current baseline with perfect trace cache and a real branch predictor improves the IPC of that machine by 27%. Runahead on the current baseline with perfect trace cache and perfect branch prediction improves the performance of that machine by 31%. On a machine with perfect branch prediction and a perfect trace cache, the number of pseudo-retired instructions per runahead mode entry goes up to 909 and the number of useful L2 misses discovered per runahead mode entry increases to 3.18.

These results are promising because they show that there is significant potential for performance improvement in a runahead execution processor. As computer architects continue to improve branch prediction and instruction fetch units, the performance improvement provided by runahead execution will increase.

4.1.3.5 Effect of Store-Load Communication in Runahead Mode

This section evaluates the importance of handling store-to-load data communication during runahead execution. We evaluate the performance difference between a model that uses a 512-byte, 4-way runahead cache with 8-byte lines versus a model that does not perform memory data forwarding between pseudo-retired runahead stores and their dependent loads (but still does perform data forwarding through the store buffer). In the latter model, instructions dependent on pseudo-retired stores are marked INV. For this model, we also assume that communication of INV bits through memory is handled correctly using oracle information without any hardware and performance cost, which gives an unfair advantage to that model. Even with this advantage, a machine that does not perform data

communication between pseudo-retired stores and dependent loads during runahead mode loses much of the performance benefit of runahead execution.

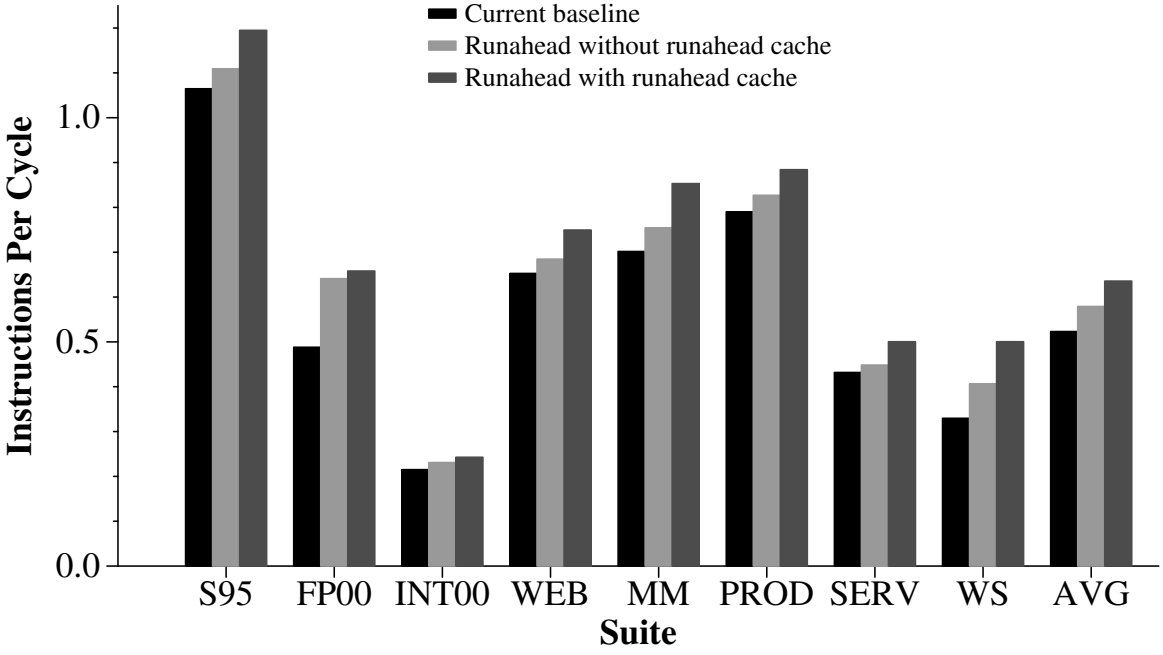


Figure 4.6: Performance of runahead execution with and without the runahead cache.

Figure 4.6 shows the results. In all suites but SPECfp2k, inhibiting store-load data communication through memory significantly decreases the performance gain of runahead execution. The overall performance improvement of runahead without using the runahead cache is 11% versus 22% with the runahead cache. For all suites except SPECfp2k and Workstation, the IPC improvement for the model without the runahead cache remains well under 10%. The improvement ranges from 4% for SPEC95 to 31% for SPECfp2k. The runahead cache used in this study correctly handles 99.88% of communication between pseudo-retired stores and their dependent loads. We found that the runahead cache size can be further decreased to 128 bytes without significantly losing the performance improvement provided by a 512-byte runahead cache. A 128-byte, 4-way runahead cache correctly

handles 91.25% of communication between pseudo-retired stores and their dependent loads but still enables a performance improvement of 21.5% over the current baseline. Table 4.4 shows the sensitivity of the performance improvement of runahead execution to the runahead cache size.

no runahead cache	16 bytes	64 bytes	128-bytes	256 bytes	512 bytes	1 KB
11.4%	13.2%	17.4%	21.5%	21.8%	22.1%	22.1%

Table 4.4: Performance improvement of runahead execution with different runahead cache sizes. Data shown is an average over all evaluated benchmarks.

4.1.3.6 Sensitivity of Runahead Execution Performance to the L2 Cache Size

L2 cache sizes are increasing as microprocessor designers push for higher performance. This section evaluates runahead execution on processors with larger caches. The results show that runahead execution continues to be effective with larger L2 caches.

We examine the IPC improvement of implementing runahead execution on our baseline processor for three different cache sizes: 512 KB, 1 MB, and 4 MB. L2 cache latency was kept the same in all these experiments. Figure 4.7 shows the performance improvement of adding runahead execution on top of each of these L2 cache sizes. The three percentage numbers on top of each suite show respectively the IPC improvement of adding runahead to the baseline processor with 512 KB, 1 MB, and 4 MB L2 cache. This figure shows that implementing runahead execution improves the IPC by 17% and 16% respectively on processors with 1 MB and 4 MB L2 caches. Thus, runahead execution remains effective for large L2 caches.

In general, increasing the L2 cache size decreases the benefit gained from runahead execution, as expected, because increasing the L2 cache size reduces the number of L2 misses, which reduces both the number of times the processor enters runahead mode and

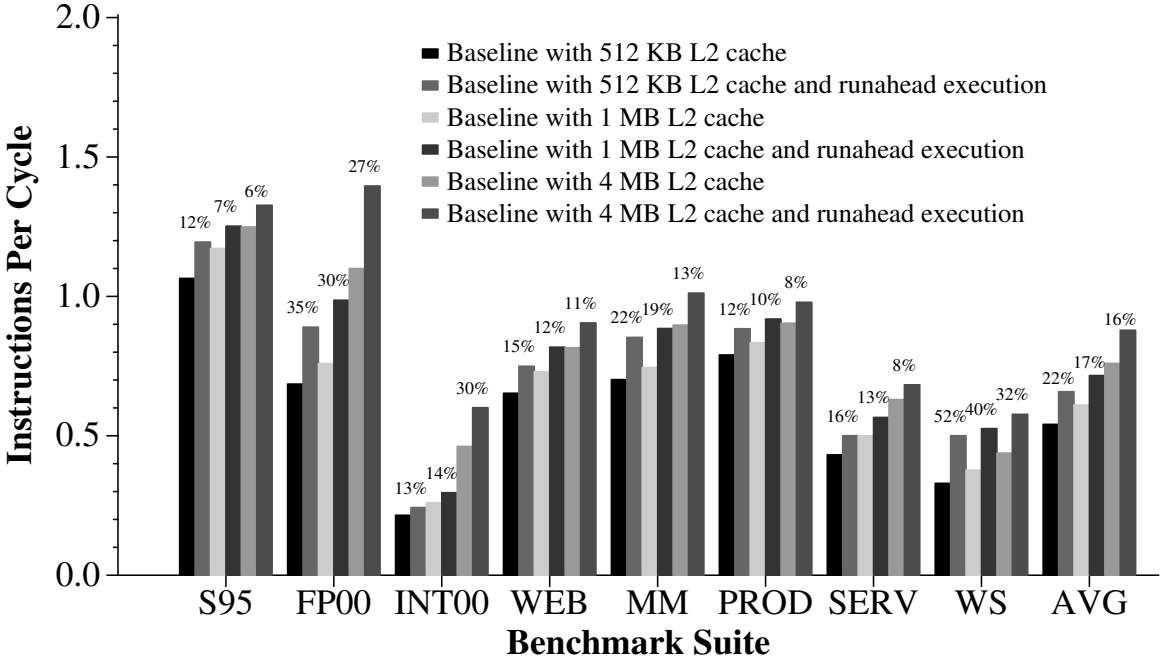


Figure 4.7: IPC improvement of runahead execution for 512 KB, 1 MB, and 4 MB L2 cache sizes. Percentage numbers show the performance improvement due to runahead execution when implemented on a machine with the respective cache size.

the number of L2 misses it generates during runahead mode. The SPECint2k benchmark suite is an interesting exception to this trend. Runahead execution becomes more effective for this suite as the L2 cache size increases. The reason is that a larger L2 cache is more tolerant to the pollution caused by runahead execution due to inaccurate prefetches generated in runahead mode. As the L2 cache size increases, the pollution caused by these inaccurate prefetches is reduced and the performance improvement of runahead execution increases.

Note that the IPC of the processor with 512 KB L2 cache and runahead execution (IPC:0.64) is higher than the IPC of the processor with 1 MB L2 cache (IPC:0.59). This suggests that runahead execution may also be used as an area-efficient alternative to larger caches, if there is enough memory bandwidth available (Note that our simulations realistically model limited bandwidth availability.)

4.1.3.7 Analysis of the Benefits of Runahead Execution

The benefits of runahead execution can be divided into two: performance improvement due to instruction prefetching and performance improvement due to data prefetching. Instruction prefetching improvement comes from prefetching runahead instructions into the L2 cache and the trace (or instruction) cache and training the conditional and indirect branch predictors during runahead mode. Data prefetching improvement comes from prefetching runahead load/store requests into the L2 cache and the L1 data cache and training the hardware data prefetcher buffers during runahead mode. Middle bars in Figure 4.8 show the IPC of runahead execution when the instruction prefetching benefits are removed. Percentage numbers on top of the bars show the percentage of IPC improvement that is due to data prefetching. On average, 88% of the IPC improvement of runahead execution comes from data prefetching. All benchmark suites, except for the Server suite, owe most of the performance improvement to data prefetching. Because the evaluated server applications are branch intensive and have large instruction footprints leading to high cache miss rates as shown in [70], almost half of the benefit of runahead execution for the Server suite comes from instruction prefetching.

Runahead execution improves performance because the processing during runahead mode fully or partially eliminates the cache misses incurred during normal mode. On average, our baseline processor incurs 13.7 L1 data cache misses and 4.3 L2 data misses per 1000 instructions. If we add runahead execution to the baseline, the L1 data cache miss rate during normal mode is reduced by 18%. With runahead, 15% of the L2 data misses of the baseline processor are never seen during normal mode. Another 18% of the L2 data misses seen in the baseline processor are initiated during runahead mode, but are not fully complete by the time they are needed by instructions in normal mode. Therefore, they are partially covered by runahead execution. We found that the performance improvement of runahead execution correlates well with the reduction in L2 data misses, indicating that the

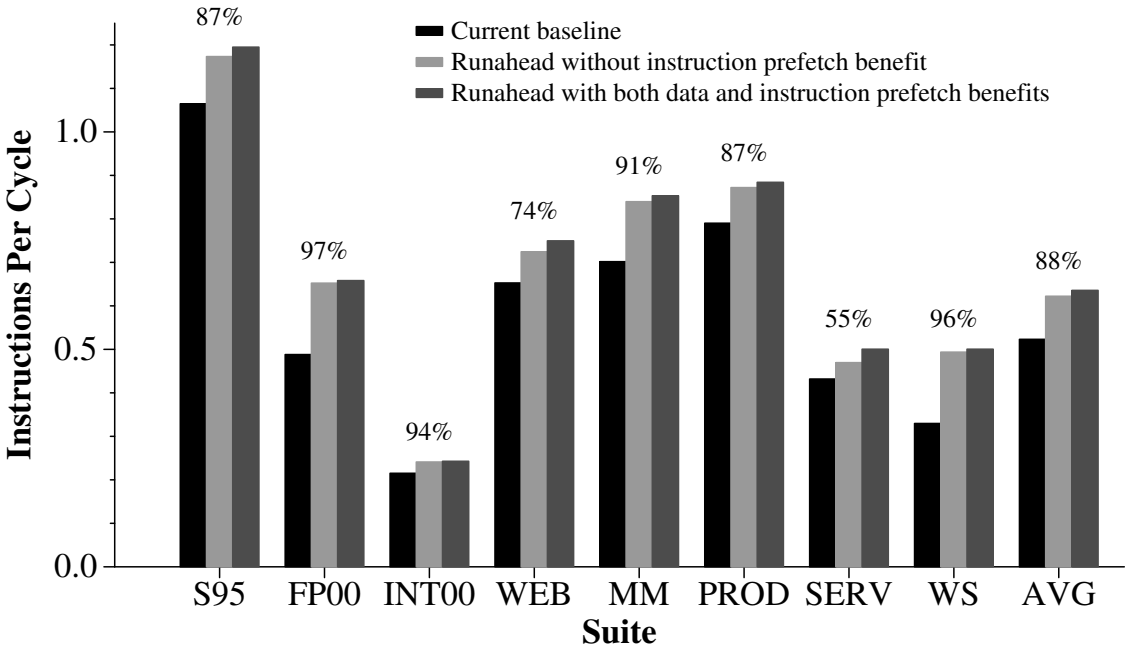


Figure 4.8: Data vs. instruction prefetching benefits of runahead execution. Percentage numbers show the fraction of the performance improvement due to only data prefetching.

main benefit of runahead execution comes from prefetching data from main memory to the L2 cache.

Overall, the L2 data miss coverage (full and partial) of runahead execution is 33%. The L2 data prefetch accuracy of runahead execution is 94% on average. This shows that runahead execution is a very accurate data prefetching technique, as one would expect, because it follows the path that would actually be followed by the instruction stream in the future. The L2 instruction miss coverage of runahead execution is 14% and the L2 instruction prefetch accuracy is 98%. It is important to keep in mind that these coverage and accuracy numbers are obtained on a baseline that already employs an aggressive stream-based data prefetcher and a next-line instruction prefetcher that prefetches the next two cache lines when an instruction fetch request misses in the L2 cache.

4.1.3.8 Runahead Execution on In-order vs. Out-of-order Processors

An in-order processor is less tolerant to latency compared to an out-of-order processor. Therefore, the performance impact of long-latency operations is worse on in-order processors. This section compares the performance benefit of runahead execution on an in-order processor with its performance benefit on an out-of-order processor. The in-order processor we simulate has the same pipeline depth, issue width, cache sizes, latencies, reorder buffer size, and supporting structures (e.g., hardware prefetchers) as the out-of-order current baseline, except that it schedules instructions in program order. It also makes use of register renaming to avoid stalling on write-after-write and write-after-read dependencies. Runahead execution is initiated on an L1 data cache miss in the in-order processor because in-order execution is unable to tolerate the latency of L1 cache misses.

Figure 4.9 shows the IPC of four machines from left to right for each suite: in-order execution, in-order execution with runahead, out-of-order execution, out-of-order execution with runahead. The two percentage numbers on top of the bars for each suite indicate the IPC improvement of adding runahead execution to the in-order baseline and the out-of-order baseline, respectively. On average, implementing runahead execution on the in-order baseline improves the IPC of that processor by 40%. This IPC improvement is much greater than the 22% IPC improvement obtained by implementing runahead on the out-of-order baseline, confirming the intuition that runahead execution would be more useful on a less latency-tolerant processor. It is also important to note that a significant IPC difference exists between the in-order baseline and the out-of-order baseline (86% on average). This difference is still significant, albeit smaller, between the in-order baseline with runahead and the out-of-order baseline with runahead (62% on average).

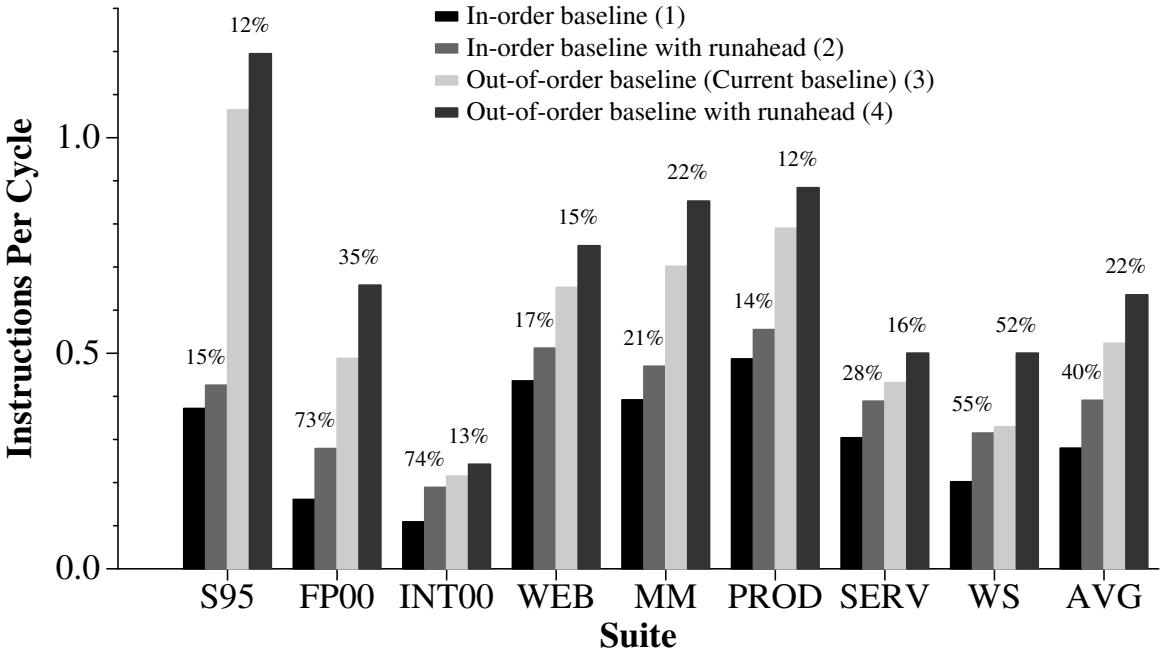


Figure 4.9: IPC improvement of runahead execution on in-order and out-of-order processors. Percentage numbers show the performance improvement of runahead execution respectively on in-order and out-of-order machines.

4.1.3.9 Runahead Execution on the Future Model

This section briefly evaluates the performance of runahead execution on a future x86 processor model that is more aggressive than the current baseline. A more thorough evaluation of runahead execution on aggressive future processors is provided in the rest of this dissertation using the Alpha ISA processors.

Figure 4.10 shows the IPC of the future baseline machine, future baseline with runahead execution, and future baseline with perfect L2 cache. The number on top of each suite is the percentage improvement due to adding runahead execution over to the future baseline. Due to their long simulation times, benchmarks art and mcf were excluded from evaluation of the future model. Runahead execution improves the performance of the future baseline by 23%, increasing the average IPC from 0.62 to 0.77. This data shows that

runahead execution is effective on a wider, deeper, and larger machine.

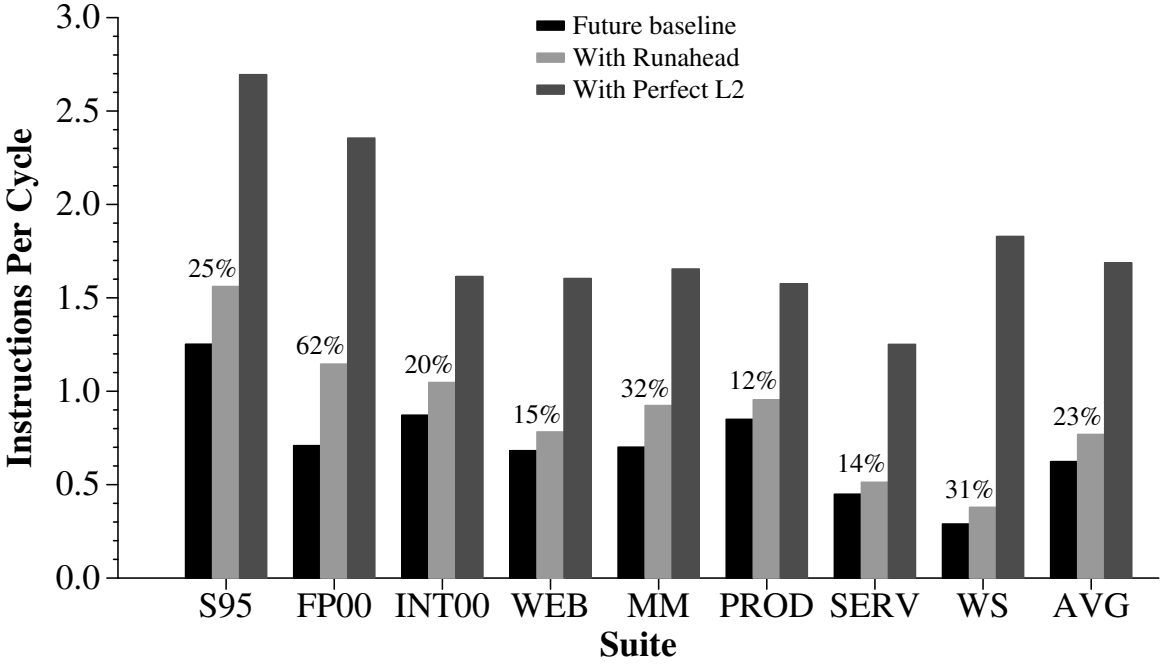


Figure 4.10: Runahead execution performance on the future baseline. Percentage numbers show the performance improvement of runahead execution on the future baseline.

The importance of a better frontend in a runahead execution processor becomes much more pronounced on a larger machine model, as shown in Figure 4.11. The number on top of each suite is the percentage IPC improvement due to runahead execution. The average IPC of the future baseline with a perfect instruction supply is 0.90 whereas the IPC of the future baseline with a perfect instruction supply and runahead execution is 1.38, which is 53% higher. Contrasting this number to the 31% IPC improvement provided by runahead execution on the current baseline with a perfect frontend, we can conclude that a better frontend is even more essential for runahead performance on the future processor. This is because branch mispredictions and fetch breaks, which adversely affect the number of useful instructions the processor can execute in runahead mode, are costlier on a wider, deeper, and larger machine.

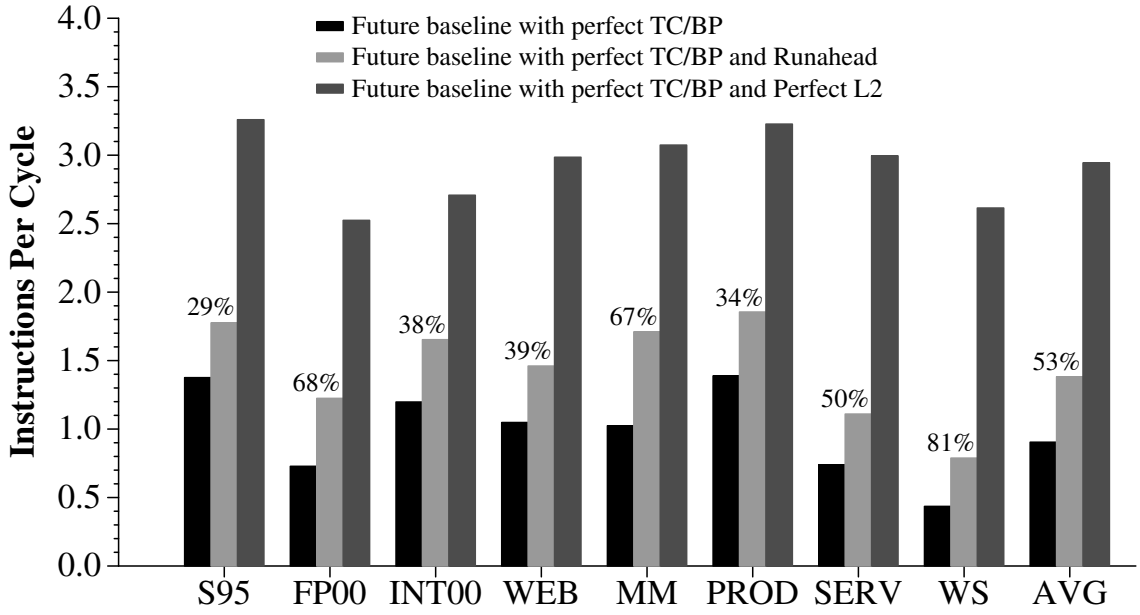


Figure 4.11: Effect of a perfect frontend on runahead execution performance on the future model. Percentage numbers show the performance improvement of runahead execution on the future baseline with perfect frontend.

4.2 Evaluation of Runahead Execution on an Alpha ISA Processor

The experimental evaluations presented in the remainder of this dissertation are performed using a cycle-accurate microarchitecture simulator that models an aggressive future processor implementing the Alpha ISA.

4.2.1 Simulation Methodology

The simulator used is an execution-driven simulator that is developed by the members of the High Performance Substrate (HPS) Research Group [19]. The machine model used for evaluation is based on the HPS research project [90, 91]. The evaluated baseline processor is an aggressive processor that implements wide superscalar execution, out-of-

order instruction scheduling and execution, aggressive branch prediction mechanisms, deep execution pipelines, and aggressive data prefetching techniques.

The simulator used faithfully models the execution of an aggressive execution engine that is expected to be in development within the next decade. We have carefully calibrated the parameters of the processor and especially the model of the memory system. An accurate model of the memory system is necessary in performing an accurate evaluation of runahead execution because runahead execution aims to improve the performance of the memory system. Furthermore, an accurate model of the processor is also necessary because runahead execution relies on the execution of instructions in the processing core to improve performance.

It is important to emphasize that the machine model evaluated is an aggressive baseline for experimenting with runahead execution. Aggressive prefetching techniques are used so that the benefits of runahead execution are not inflated. An aggressive and idealized model is used for out-of-order execution so that the baseline processor gains the maximum possible latency tolerance benefits from a given instruction window size. The next section describes the baseline machine model.

4.2.2 Baseline Microarchitecture Model

Table 4.5 shows the configuration of the baseline Alpha ISA processor. The register files, load/store buffer, and the scheduling window are the same size as the instruction window so that they do not cause the baseline processor to stall. This extracts the maximum possible latency-tolerance benefits from the baseline 128-entry instruction window. An aggressive late physical register allocation scheme [44] and an early register deallocation scheme [73] are also employed in the baseline processor.

The baseline processor handles stores instructions aggressively. A store instruction that misses in the caches does not block instruction retirement. Instead, it is removed from

Pipeline	24-stage pipeline, 20-cycle minimum branch misprediction penalty
Front End	64KB, 4-way instruction cache with 2-cycle latency; 8-wide decoder with 1-cycle latency; 8-wide renamer with 4-cycle latency
Branch Predictors	64K-entry gshare [71], 64K-entry PAs [127] hybrid with a 64K-entry selector; 4K-entry, 4-way branch target buffer; 64-entry return address stack; 64K-entry target cache [18] for indirect branches; wrong-path execution faithfully modeled (including misprediction recoveries on the wrong path)
Instruction Window	128-entry reorder buffer; 160-entry INT, 160-entry FP physical register files with 4-cycle latency; 128-entry scheduling window; 128-entry load/store buffer, a store miss does not block the instruction window unless the store buffer is full
Execution Core	8 general-purpose functional units, fully-pipelined except for FP divide; full bypass network; all operate instructions have 1-cycle latency except for integer multiply (8-cycles), FP operations (each 4-cycle), and FP divide (16 cycles); AGEN (address generation) latency is 1 cycle for ld/st instructions
On-chip Caches	64KB, 4-way L1 data cache with 8 banks and 2-cycle latency, allows 4 load and 1 store accesses per cycle; 1MB, 32-way, unified L2 cache with 8 banks and 10-cycle latency, maximum 128 outstanding L2 misses; 1 L2 read port, 1 L2 write port, 8 L2 banks; all caches use LRU replacement and have 64B lines; all intermediate queues and traffic are modeled
Buses and Memory	500-cycle minimum main memory latency; 32 DRAM banks; 32B-wide, split-transaction core-to-memory bus at 4:1 frequency ratio; maximum 128 outstanding misses to main memory; bank conflicts, port conflicts, bandwidth, and queueing delays are faithfully modeled
Hardware Prefetcher	Stream-based prefetcher [113]; 32 stream buffers; can stay 64 cache lines ahead of the processor reference stream
Runahead Support	128-byte runahead cache for store-load data forwarding during runahead mode

Table 4.5: Machine model for the baseline Alpha ISA processor.

the instruction window if it is the oldest instruction in the window and if it is guaranteed that it will not cause an exception. The store remains in the store buffer until its miss is serviced. Once its miss is serviced, the store instruction writes its data into the L1 data cache and is removed from the store buffer. Note that younger instructions can be placed in the instruction window and can be executed and retired while the retired store instruction's miss is being serviced. This allows the baseline processor to make forward progress in the presence of an L2-miss store.

Current processors spend a significant portion of their execution time on the wrong

program path and future processors are expected to spend even more. Execution on the wrong path was found to significantly affect a processor's memory-related performance [77]. The baseline processor models wrong-path execution faithfully. Branch misprediction recoveries that occur on the wrong path are correctly modeled and the effects of wrong-path memory references are correctly simulated.

4.2.3 Baseline Memory Model

The modeled memory system is shown in Figure 4.12. At most, 128 I-Cache and D-Cache requests may be outstanding. These requests may reside in any of the four buffers in the memory system (L2 Request Queue, Bus Request Queue, Memory Controller, and L2 Fill Queue). Two of these queues, L2 Request Queue and Bus Request Queue are priority queues where requests generated by older instructions have higher priority. Such prioritization is fairly easy to implement on-chip and reduces the probability of a full window stall by servicing older instructions' requests earlier. The bus is pipelined, split-transaction, 256-bit wide, and has a one-way latency of 50 processor cycles. At most two requests can be scheduled onto the bus every bus cycle, one from the Bus Request Queue and one from the Memory Controller. Processor frequency is four times the bus frequency. The Memory Controller takes memory requests from the bus and schedules accesses to DRAM banks. Requests to independent banks can be serviced in parallel. Requests to the same bank are serialized and serviced in FIFO order. We model 32 DRAM banks, each with an access latency of 400 processor cycles. Hence, the round-trip latency of an L2 miss request is a minimum of 500 processor cycles (400-cycle memory access + 100-cycle round-trip on the bus) without any queuing delays or bank conflicts. On an L2 cache miss, the requested cache line is brought into both the L2 cache and the first-level cache that initiated the request. The L2 cache is inclusive of the L1 caches. A store instruction request that misses the data cache or the L2 cache allocates a line in the respective cache. A store instruction

initiates a fetch request for the cache line it needs once its effective address is available. Therefore, store instructions can generate speculative memory requests like load instructions do. Write-back requests from D-Cache are inserted into the L2 Request Queue and write-back requests from the L2 Cache are inserted into the Bus Request Queue as bandwidth becomes available from instruction and data fetch requests.

The baseline processor employs an aggressive stream-based prefetcher, similar to the one described by Tendler et al. [113], that can detect and generate prefetches for 32 different access streams. The prefetcher prefetches into the L2 cache. The memory system gives priority to load and store instruction requests over prefetcher requests. A prefetch stream is created on an L2 cache load or store miss. After a stream is created, the prefetcher monitors L2 accesses to nearby addresses to determine the direction of the access stream.[1] Once the direction of the access stream is determined, the prefetcher starts monitoring L2 accesses to addresses that are nearby (within 8 cache lines of) the address that was last accessed on the stream. If nearby addresses are accessed, the prefetcher predicts that subsequent addresses on the access stream will also be accessed and generates prefetch requests. For each address accessed on the stream, prefetch requests for two subsequent addresses are generated. The prefetch requests are buffered in the Prefetch Request Queue (see Figure 4.12). Requests from the Prefetch Request Queue are inserted into the L2 Request Queue when enough bandwidth is available from I-Cache and D-Cache requests. We model an aggressive prefetcher that can stay up to 64 cache lines ahead of the processor's access stream.

[1]The direction of the access stream is determined by monitoring whether consecutive accesses are made to increasing or decreasing memory addresses.

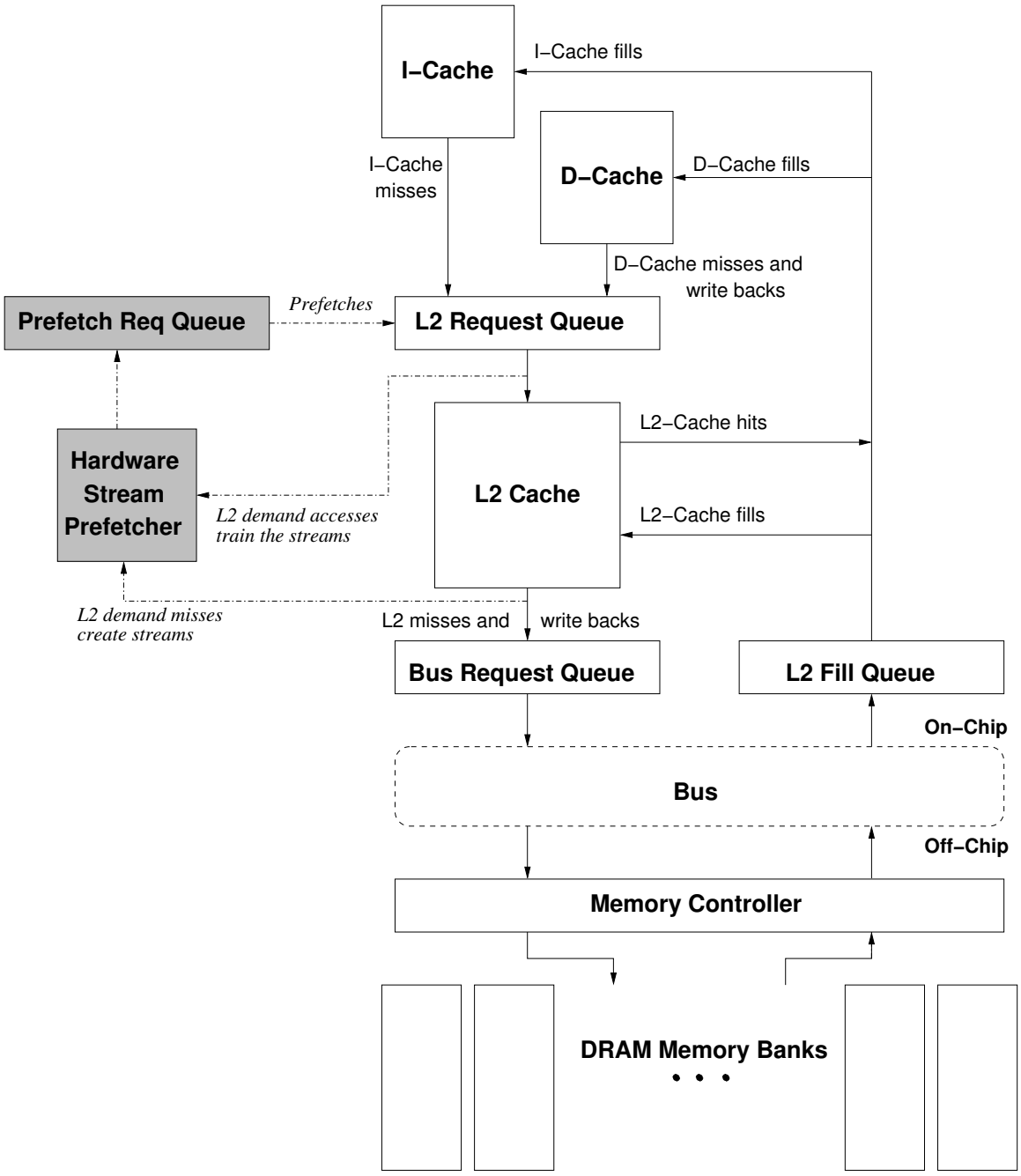


Figure 4.12: The memory system modeled for evaluation. Shaded structures belong to the hardware data prefetcher.

4.2.4 Benchmarks

Experiments were performed using the SPEC CPU2000 benchmark suite [110]. All benchmarks were compiled for the Alpha EV6 ISA with the `-fast` optimizations and profiling feedback enabled. Pointer-intensive Olden integer benchmarks were used to evaluate the address-value delta prediction mechanism presented in Chapter 6. Olden benchmarks were simulated for all other experiments, but these experiments are not included for brevity. The simulation results of the Olden benchmarks support the conclusions drawn in the rest of this dissertation.

The input sets used for the SPEC CPU2000 integer benchmarks were taken from the MinneSPEC suite [61] to reduce the simulation times. Official reference input sets provided by SPEC were used to simulate the SPEC CPU2000 floating-point benchmarks. The initialization portion was skipped for each floating-point benchmark and detailed simulation was performed for the next 250 million instructions. The input sets used for the Olden benchmarks are described in Section 6.5. Table 4.6 describes the benchmarks used in the evaluations of runahead execution on the Alpha processor.

Benchmark	Suite	Description
bzip2	SPEC INT	data compression algorithm
crafty	SPEC INT	chess game
eon	SPEC INT	probabilistic ray tracer
gap	SPEC INT	mathematical group theory program
gcc	SPEC INT	C programming language optimizing compiler
gzip	SPEC INT	data compression algorithm
mcf	SPEC INT	combinatorial optimization for vehicle scheduling
parser	SPEC INT	syntactic text parser for English
perlbmk	SPEC INT	perl programming language interpreter
twolf	SPEC INT	place and route simulator
vortex	SPEC INT	object-oriented database transaction program
vpr	SPEC INT	circuit place and route program for FPGAs
ammp	SPEC FP	computational chemistry and molecular modeling
applu	SPEC FP	computational fluid dynamics and physics
apsi	SPEC FP	weather and pollutant prediction program
art	SPEC FP	object recognition in thermal images
equake	SPEC FP	simulation of seismic wave propagation
facerec	SPEC FP	face recognition
fma3d	SPEC FP	finite element method crash simulation
galgel	SPEC FP	computational fluid dynamics
lucas	SPEC FP	primality testing of Mersenne numbers
mesa	SPEC FP	3-D graphics library
mgrid	SPEC FP	multi-grid solver in a 3-D potential field
sixtrack	SPEC FP	high energy nuclear physics accelerator design
swim	SPEC FP	shallow water modeling for weather prediction
wupwise	SPEC FP	quantum chromodynamics
bisort	Olden	bitonic sequence sorter
health	Olden	simulation of a health care system
mst	Olden	finds the minimum spanning tree of a graph
perimeter	Olden	computes perimeters of regions in images
treeadd	Olden	sums values distributed on a tree
tsp	Olden	traveling salesman problem
voronoi	Olden	computes voronoi diagram of a set of points

Table 4.6: Evaluated benchmarks for the Alpha processor.

4.2.5 Performance Results on the Alpha Processor

The remainder of this chapter repeats and extends some of the evaluations performed for the x86 processor and shows that runahead execution's performance benefit on the Alpha processor is similar to that on the x86 processor.

4.2.5.1 Runahead Execution vs. Hardware Data Prefetcher

Figure 4.13 shows the IPC performance of four different Alpha processor models on the SPEC CPU2000 benchmarks: 1) a model with no prefetcher and no runahead (baseline without the prefetcher), 2) a model with the stream-based hardware prefetcher but no runahead (baseline), 3) a model with runahead but no prefetcher, and 4) a model with the prefetcher and runahead (baseline with runahead).

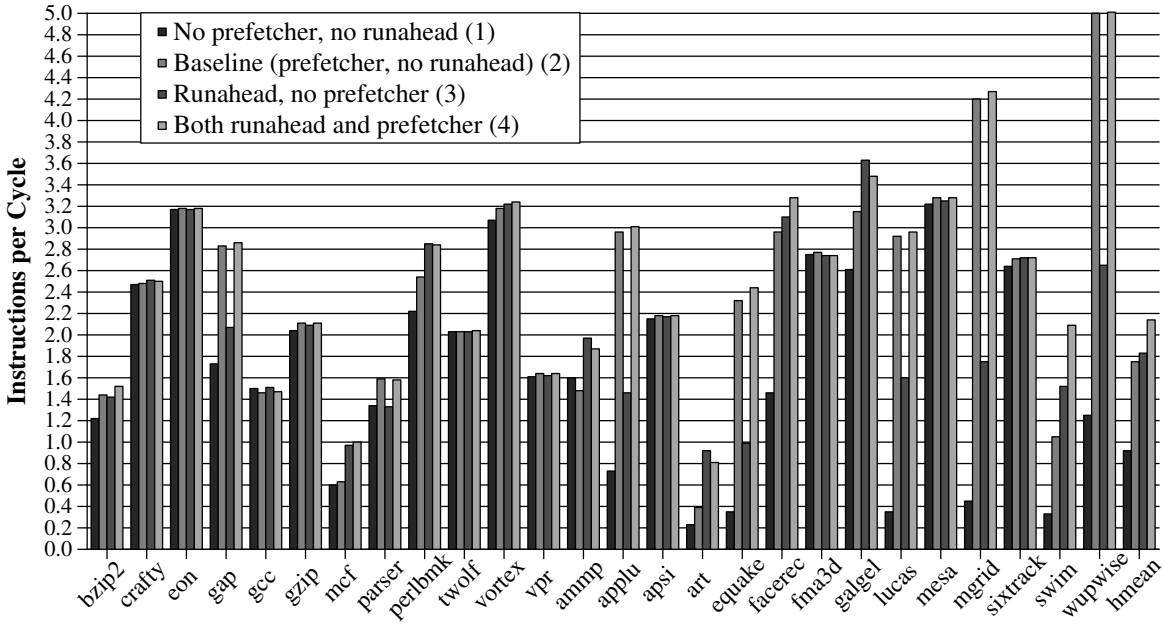


Figure 4.13: Runahead execution performance on the baseline Alpha processor.

Similarly to the results on the x86 processor, the model with only runahead outper-

forms the model with only the prefetcher for both integer and floating-point benchmarks. Again similarly to the results on the x86 processor, adding runahead execution to the baseline processor with the aggressive prefetcher improves the IPC by 22.6% on average. Table 4.7 shows the percentage IPC improvement across different models averaged separately over integer (INT) and floating-point (FP) benchmarks. These results support the following conclusions about runahead execution:

- Runahead execution is a more effective prefetching mechanism than the stream prefetcher especially on the INT benchmarks that have irregular memory access patterns.
- Runahead execution provides significant performance benefits over a baseline with an aggressive prefetcher. Thus, the benefits of runahead execution and stream prefetching are complementary.[2]
- Runahead execution is more effective on FP benchmarks than on INT benchmarks because FP benchmarks have a large amount of memory-level parallelism available and INT benchmarks have relatively low branch prediction accuracy.

Comparison	INT	FP	Average
IPC delta with only prefetcher (over no prefetcher)	8.8%	162.6%	90.5%
IPC delta with only runahead (over no prefetcher)	14.8%	172.3%	99.2%
IPC delta with runahead and prefetcher (over no prefetcher)	20.8%	254.6%	133.6%
IPC delta with perfect L2 cache (over no prefetcher)	50.7%	385.6%	204.7%
IPC delta of using runahead (over the baseline)	**10.9%**	**35.0%**	**22.6%**
IPC delta with perfect L2 cache (over the baseline)	38.6%	84.9%	59.9%

Table 4.7: IPC improvement comparisons across different prefetching models.

[2]The results presented in Table 4.7 also show that a baseline without a prefetcher performs very poorly especially on FP benchmarks. The modeled prefetcher improves the average IPC of FP benchmarks by 162.6%. If we had not used this aggressive prefetcher in our baseline model, the average IPC improvement of runahead execution would be 172.3% on FP benchmarks instead of the 35.0% on the baseline with the prefetcher. Serious studies of large instruction windows and techniques to increase memory latency tolerance must therefore include a state-of-the-art prefetcher. Otherwise, the results presented would be meaningless to the designers of aggressive processors.

4.2.5.2 Sensitivity of Runahead Execution Performance to Main Memory Latency

Runahead execution is a mechanism for tolerating long main memory latencies. As the main memory latency increases, one would expect the performance benefit of runahead execution to increase due to two reasons:

- With a longer memory latency, the performance loss due to main memory latency becomes more significant and therefore tolerating the increased latency would result in larger performance improvements.
- With a longer memory latency, the time a processor spends in runahead mode increases. Hence, the processor can execute more instructions in runahead mode further ahead in the instruction stream. This results in the discovery of L2 misses further in the instruction stream, which could not have been discovered with a shorter memory latency.

Figure 4.14 shows the average IPC performance of processors with different minimum main memory latencies with and without runahead execution. The numbers on top of each pair of bars shows the IPC improvement due to runahead execution for the corresponding memory latency. As expected, as memory latency increases, the IPC of the baseline processor decreases. Also, the performance improvement due to runahead execution increases. Runahead execution improves the IPC by 2.1% on a machine with a relatively short, 100-cycle memory latency and by 37.7% on a machine with a long, 900-cycle memory latency. The IPC performance of a runahead execution processor with 900-cycle memory latency is 1% higher than that of a non-runahead processor with 500-cycle memory latency. Thus, runahead execution can preserve the absolute performance of a non-runahead processor with a short memory latency as the memory latency gets longer.

Figure 4.15 provides a closer look at the IPC improvement of runahead execution with different memory latencies by examining each benchmark. For all benchmarks, in-

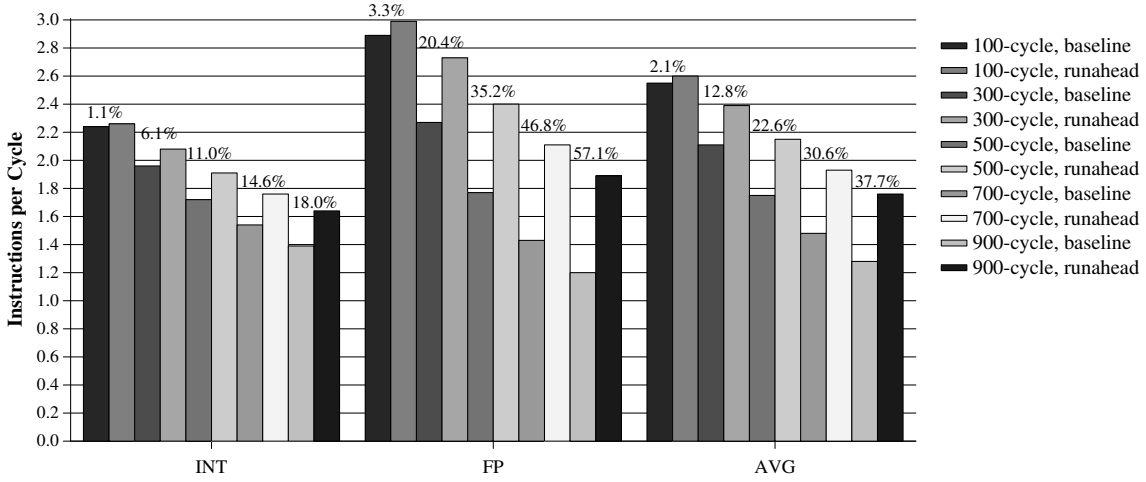


Figure 4.14: Runahead execution performance on baselines with different minimum main memory latency. Data is averaged over INT, FP, and all (AVG) benchmarks.

creased memory latency results in increased performance improvement due to runahead execution. It is worthy to note that runahead execution slightly degrades the IPC for short memory latencies in some benchmarks (notably parser). With a short memory latency the processor does not execute enough instructions to discover a new L2 miss in runahead mode. The pipeline flush at the end of runahead mode results in performance loss since runahead mode does not provide any benefits to outweigh the performance cost of the flush. Even so, runahead execution provides a performance improvement on average, albeit small, on a processor with 100-cycle memory latency.

We expect that runahead execution will become more effective in future processors. As processor and system designers continue to push for shorter cycle times and larger memory modules, and memory designers continue to push for higher bandwidth and density, main memory latencies will continue to increase in terms of processor cycles. As main memory latency increases, the performance of runahead execution would also increase, as we have shown.

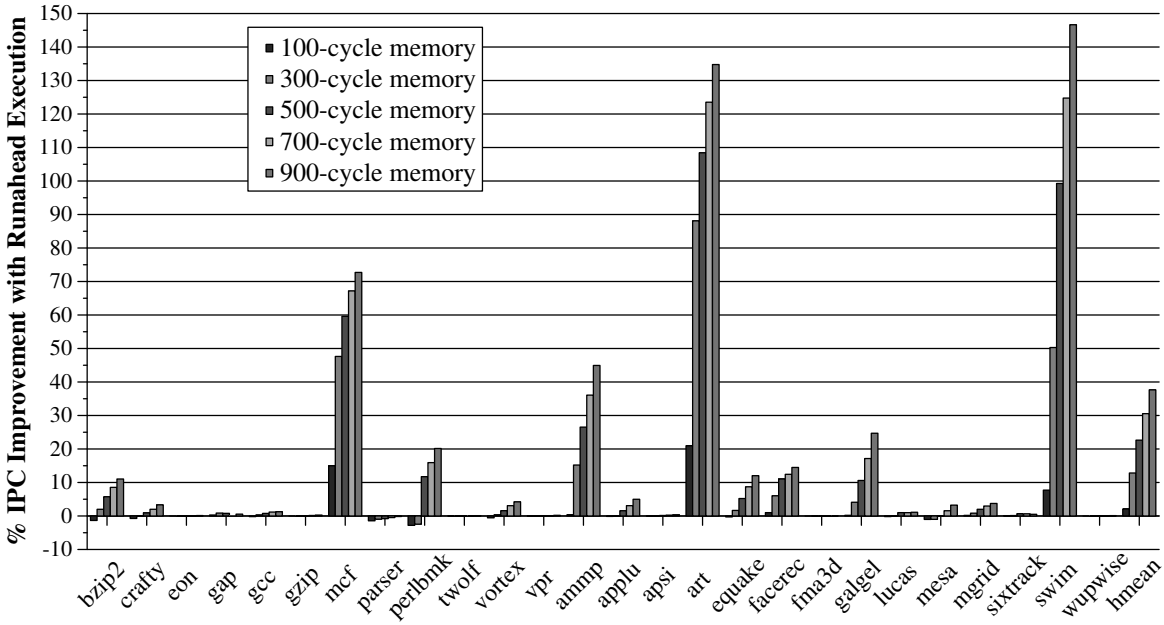


Figure 4.15: IPC improvement due to runahead execution on baselines with different minimum main memory latency.

4.2.5.3 Runahead Execution vs. Large Instruction Windows

Figure 4.16 shows the IPC of processors with instruction window sizes ranging from 64 to 8192, with and without runahead execution. The three graphs show the average IPC over INT, FP, and all (AVG) benchmarks. Runahead execution on a 128-entry instruction window achieves 1% higher performance than the conventional processor with 384-entry window. Similarly, runahead execution on a 256-entry window achieves 1% higher performance than the conventional processor with 512-entry window. Similar to the results shown for the x86 processor, runahead execution on the Alpha processor is able to achieve the performance of a larger window without requiring the implementation of large structures.

Figure 4.16 also shows that runahead execution is more effective (i.e., provides larger performance improvements) when implemented on a baseline processor with smaller

instruction windows. For example, runahead execution improves the average IPC by 38% when implemented on a 64-entry window processor whereas it degrades the average IPC by 1.4% when implemented on a 2048-entry window processor. A 2048-entry instruction window is already able to tolerate most of the main memory latency. Therefore, adding runahead execution does not provide significant performance benefits and it degrades performance due to the performance overhead of exiting from runahead mode.

We note that the runahead execution implementation proposed in this dissertation, in particular the runahead mode entry and exit policies, is optimized assuming that runahead execution is implemented on a processor with a relatively small (128-entry) window. If runahead execution is to be implemented on a large-window processor with better tolerance to memory latency, the runahead mode entry and exit policies may need to be revisited. For example, entry into runahead mode can be delayed until the instruction window becomes full on a large-window processor to avoid entering runahead mode on relatively shorter-latency L2 cache misses that can already be fully tolerated by the large instruction window. The data shown in Figure 4.16 is obtained without performing such optimizations specific to large-window processors. Even so, runahead execution improves the average IPC by 1% when implemented on a processor with 1024-entry instruction window.

Effect of a Longer Main Memory Latency

As described in Section 4.2.5.2, runahead execution becomes more effective as the main memory latency gets longer. As future processors are expected to have significantly longer main memory latencies than current processors, we evaluate the performance of runahead execution versus large instruction windows on processors with longer main memory latencies. The minimum main memory latency of the processors evaluated in this section is 1000 or 2000 cycles.[3]

[3]Note that a 1000-cycle main memory latency is not unrealistic in a near future processor. A 10 GHz processor with a 100-nanosecond main memory would face and need to tolerate a 1000-cycle memory latency.

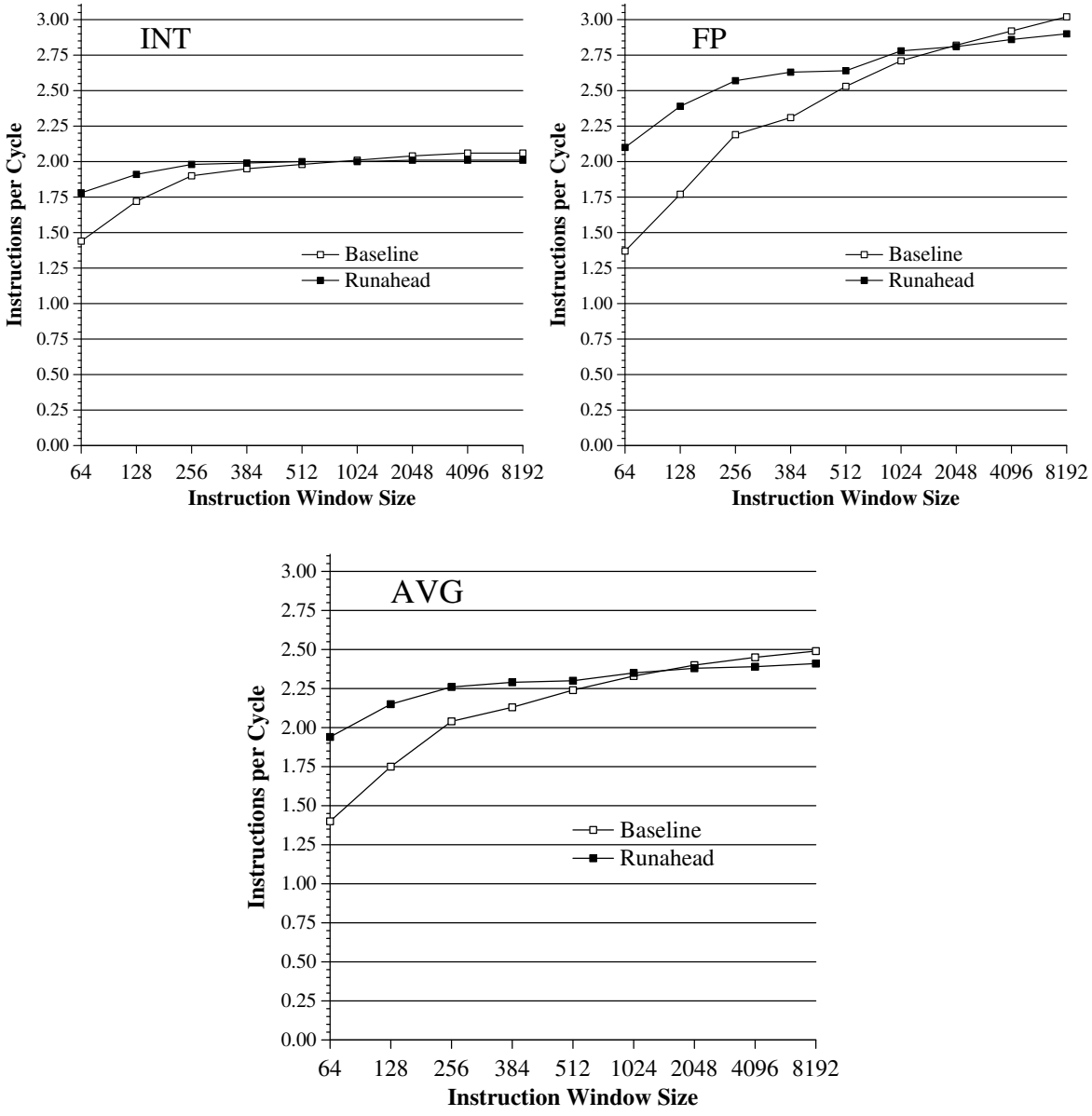


Figure 4.16: Performance of runahead execution vs. large windows (minimum main memory latency = 500 cycles).

Figure 4.17 shows the IPC of processors with instruction window sizes ranging from 64 to 8192, with and without runahead execution, for a minimum main memory latency of 1000 cycles. The sizes of the load/store buffer, register file, and scheduling windows are the same as the instruction window size for the evaluated processors and these structures are modeled as monolithic structures. Thus, the performance of a given processor is the maximum performance achievable by the modeled instruction window, since we do not consider the performance loss imposed by specific implementations.

When the memory latency is 1000 cycles, runahead execution on a 128-entry instruction window achieves 0.1% higher performance than the conventional processor with 1024-entry window. Similarly, runahead execution on a 256-entry window achieves 0.2% higher performance than the conventional processor with 2048-entry window. In other words, with a longer memory latency, runahead execution provides the same performance as a processor with 8 times the instruction window size. Given that runahead execution adds very little hardware to a state-of-the-art processor, it provides a cost- and complexity-effective alternative to building a processor with 8 times the instruction window size of today's processors.

When the memory latency is 1000 cycles, runahead execution still provides significant performance benefits if it is implemented on a large instruction window. Figure 4.17 shows that implementing runahead execution on processors with 512, 1024, 2048, and 4096-entry windows improves the average performance of the respective baseline processor by 10%, 6.5%, 2.8%, and 0.8%. Therefore, designers of processors with large instruction windows can also utilize runahead execution to increase the latency tolerance of a large window to long main memory latencies.

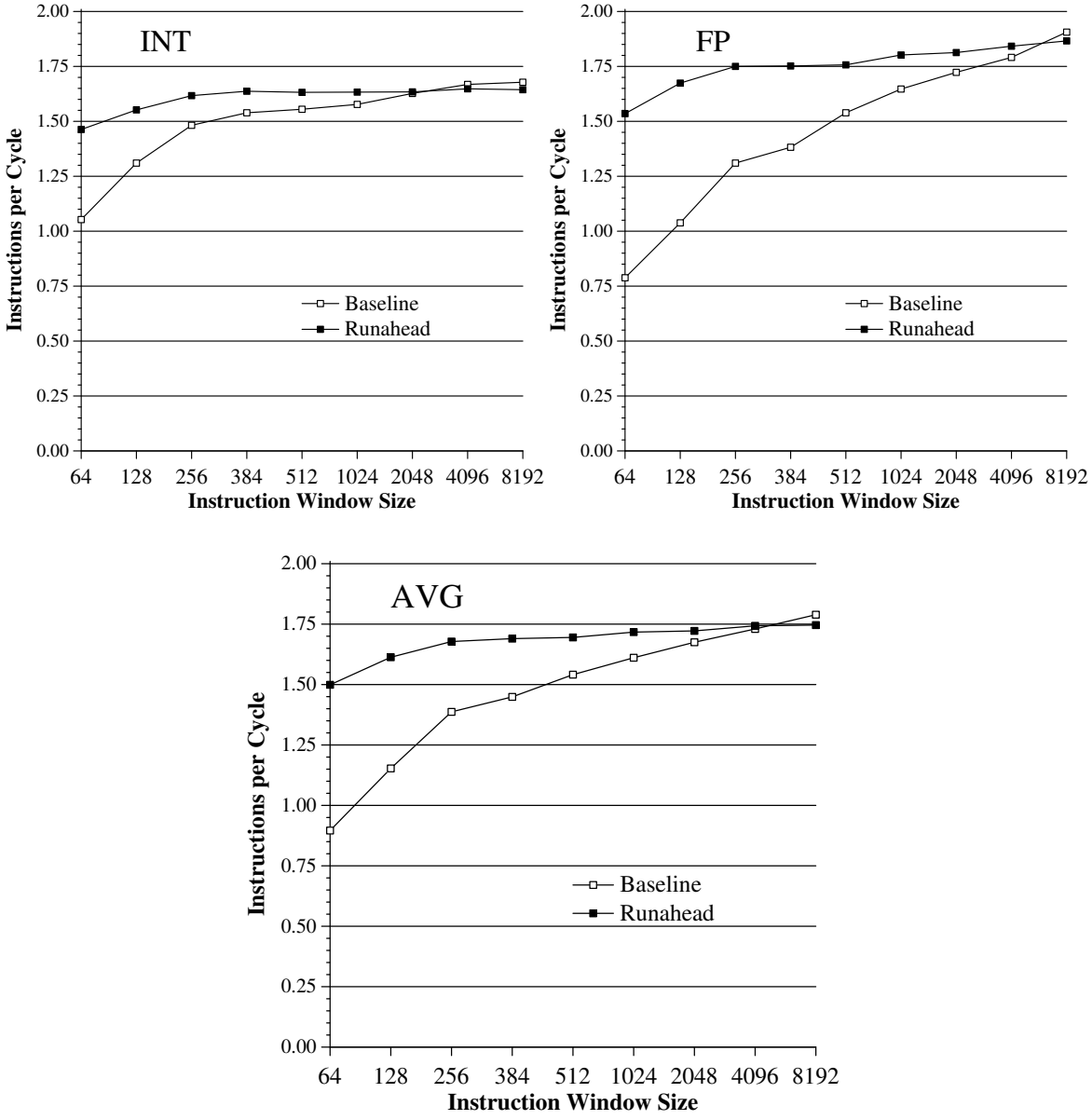


Figure 4.17: Performance of runahead execution vs. large windows (minimum main memory latency = 1000 cycles).

Figure 4.18 shows the IPC of processors with instruction window sizes ranging from 64 to 8192, with and without runahead execution, for a minimum main memory latency of 2000 cycles. With a 2000-cycle main memory latency, the performance of a runahead execution processor with 128-entry window is 3% higher than that of the conventional processor with 2048-entry window and within 1% of that of the conventional processor with 4096-entry window. Also, runahead execution on a 256-entry window processor achieves 1.7% higher performance than the conventional processor with 4096-entry window. Hence, with a 2000-cycle main memory latency, a runahead execution processor with a relatively small instruction window is able to provide higher performance than a conventional processor with 16 times the instruction window size.

When the memory latency increases to 2000 cycles, the effectiveness of runahead execution on a processor with a large instruction window also increases. With a 2000-cycle memory latency, implementing runahead execution on a processor with a 4096-entry window improves performance by 4.2%.

We conclude that, with increasing memory latencies, the performance benefit provided by runahead execution approximates and surpasses the performance benefit provided by a very large instruction window that can support thousands of in-flight instructions. Therefore, implementing runahead execution on a small instruction window without significantly increasing hardware cost and complexity becomes a more attractive alternative to building large instruction windows as main memory latencies increase.

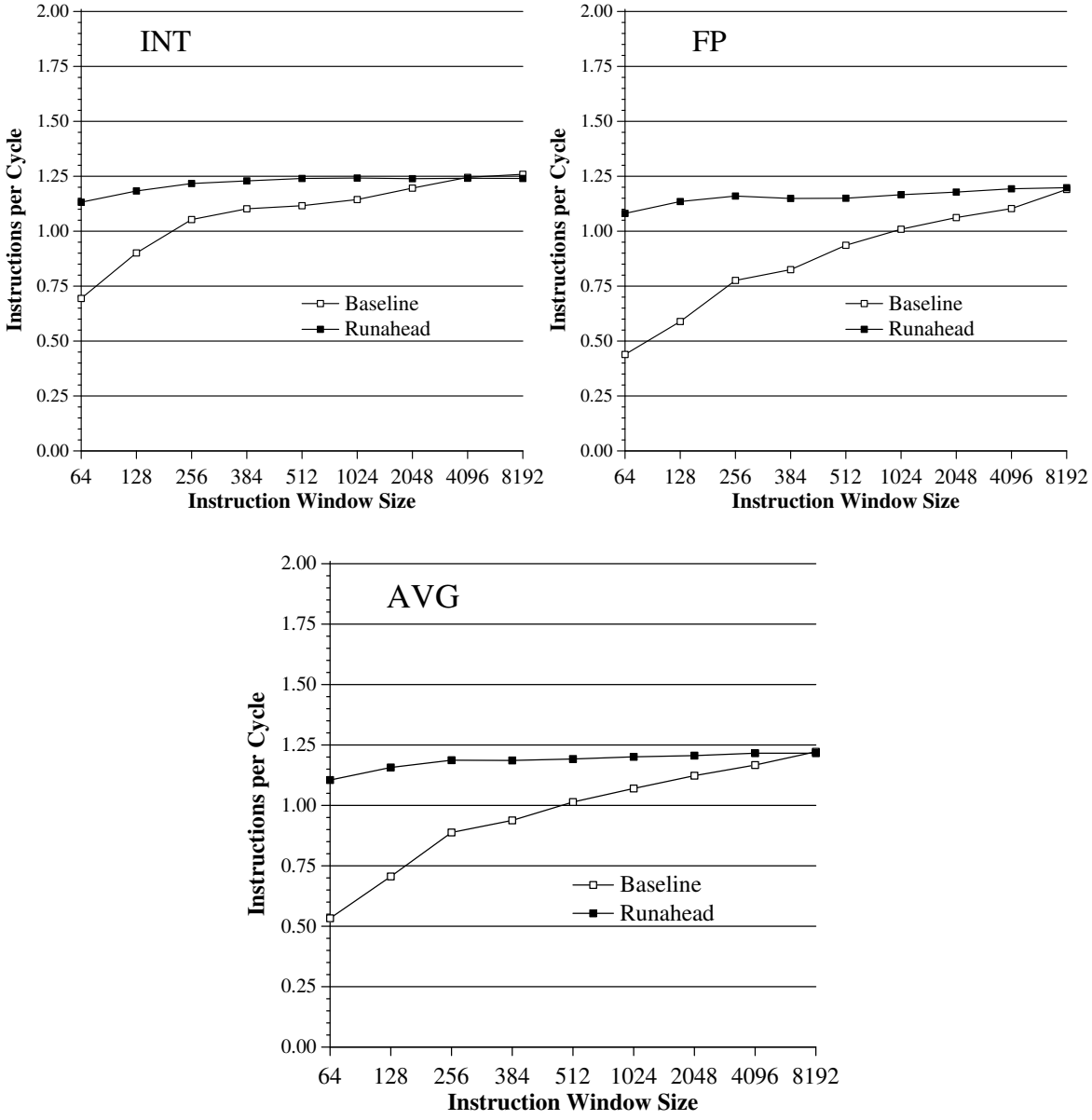


Figure 4.18: Performance of runahead execution vs. large windows (minimum main memory latency = 2000 cycles).

4.2.5.4 Runahead Execution on In-Order vs. Out-of-order Processors

Figure 4.19 shows the IPC of four processors: 1) an in-order execution processor, 2) in-order execution with runahead execution, 3) the out-of-order baseline processor, and 4) out-of-order baseline with runahead execution. The evaluated in-order processor is aggressive because it performs register renaming to eliminate stalls due to write-after-write and write-after-read dependencies. Runahead execution is initiated on an L1 data cache miss on the in-order processor because in-order execution cannot tolerate the latency of an L1 data cache miss. Furthermore, the pipeline depth of the in-order processor was modeled as 8 stages (instead of 20), assuming that an out-order processor would require extra stages to perform out-of-order scheduling. Thus, the evaluated processor models favor the in-order processor. All other parameters of the in-order processor were kept the same as the baseline out-of-order Alpha processor.

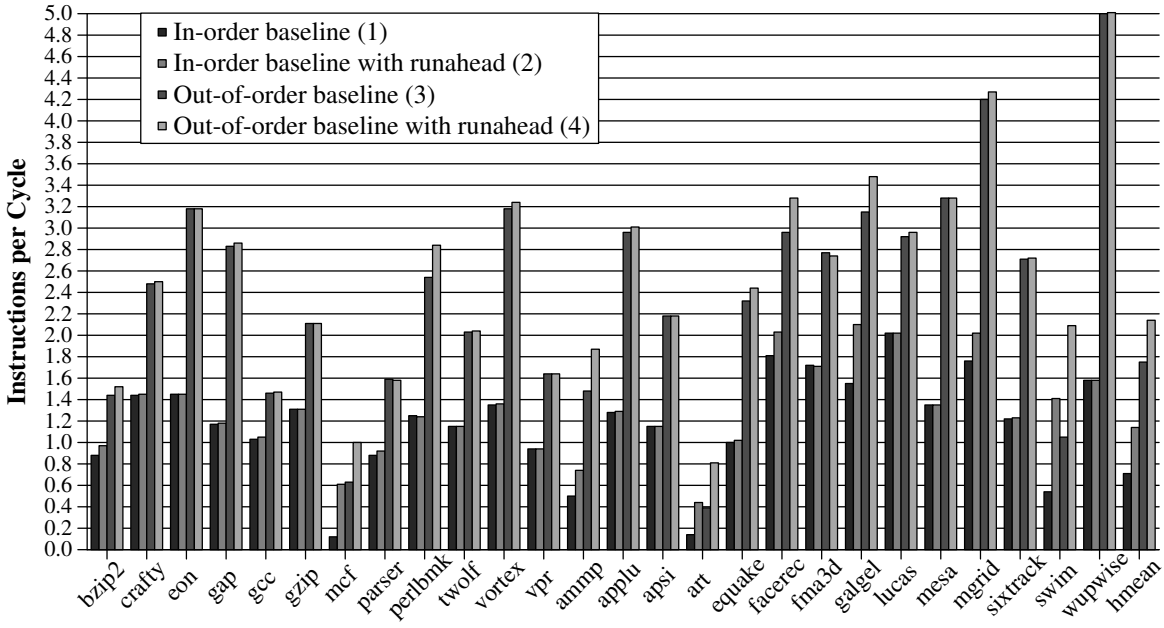


Figure 4.19: Runahead execution performance on in-order vs. out-of-order processors.

Similarly to the results on the x86 processor, runahead execution provides larger IPC improvements on an in-order processor than on an out-of-order processor. Table 4.8 summarizes the average IPC improvements across different models. Runahead execution's IPC improvement is 60.9% on the in-order processor versus 22.6% on the out-of-order processor. However, using runahead execution on an in-order processor cannot approximate the performance of out-of-order execution. An out-of-order processor outperforms an in-order processor with runahead execution by 53.7%.[4] Moreover, an out-of-order processor with runahead execution outperforms an in-order processor with runahead execution by 88.4%. These results suggest that runahead execution significantly improves the memory latency tolerance of both in-order and out-of-order processors, but a large performance gap remains between in-order and out-of-order execution when both are augmented with runahead execution.

Comparison	INT	FP	Average
IPC delta of IO+runahead (over IO)	59.9%	61.8%	60.9%
IPC delta of OOO+runahead (over OOO)	10.9%	35.0%	22.6%
IPC delta of OOO (over IO)	156.2%	139.3%	147.2%
IPC delta of OOO (over IO+runahead)	60.2%	47.9%	53.7%
IPC delta of OOO+runahead (over IO+runahead)	77.7%	99.8%	88.4%

Table 4.8: IPC improvement comparisons across in-order, out-of-order, and runahead execution models. IO stands for in-order, OOO for out-of-order.

Effect of Memory Latency

Figure 4.20 compares the performance of in-order and out-of-order execution pro-

[4]For two FP benchmarks, art and swim, an in-order processor with runahead execution outperforms a non-runahead out-of-order processor. The performance of these two benchmarks is mainly limited by L2 cache misses. Therefore, the major benefit of out-of-order execution is due to the parallelization of L2 cache misses. As runahead execution can achieve this parallelization more efficiently without being limited by the window size, an in-order processor with runahead execution is able to parallelize more L2 misses than a 128-entry window. That is why an in-order runahead processor performs better than an out-of-order processor that does not implement runahead execution.

cessors with and without runahead execution using different main memory latencies. As the main memory latency increases, the performance of the in-order processor with runahead execution gets closer to and surpasses the performance of the out-of-order processor without runahead execution, especially for FP benchmarks. For a 900-cycle minimum main memory latency, the average performance of the in-order processor with runahead execution is only 22% less than the performance of the out-of-order processor without runahead execution. For a 1900-cycle memory latency, the in-order processor with runahead execution actually outperforms the conventional out-of-order processor by 9.5%. With longer memory latencies, the 128-entry instruction window's ability to tolerate the memory latency diminishes. As runahead execution's latency tolerance is not limited by the size of any buffer, the in-order runahead processor's latency tolerance (and performance) gets close to and surpasses that of the out-of-order processor. This is especially true for FP benchmarks, which are more limited by memory performance than INT benchmarks. On integer benchmarks, a large chunk of the benefits of an out-of-order processor comes from its ability to extract instruction-level parallelism. As this cannot be achieved by an in-order runahead processor, a significant performance gap still remains between the two processors for the INT benchmarks even at longer latencies.

The performance of the out-of-order processor with runahead execution is significantly higher than that of the in-order processor with runahead execution, even though the performance gap between the two reduces as the main memory latency increases. For a 900-cycle memory latency, the out-of-order runahead processor outperforms the in-order runahead processor by 68% on average. For a 1900-cycle memory latency, the out-of-order runahead processor still outperforms the in-order runahead processor by 47%. We conclude that out-of-order execution augmented with runahead execution provides the best memory latency tolerance across a wide variety of main memory latencies.

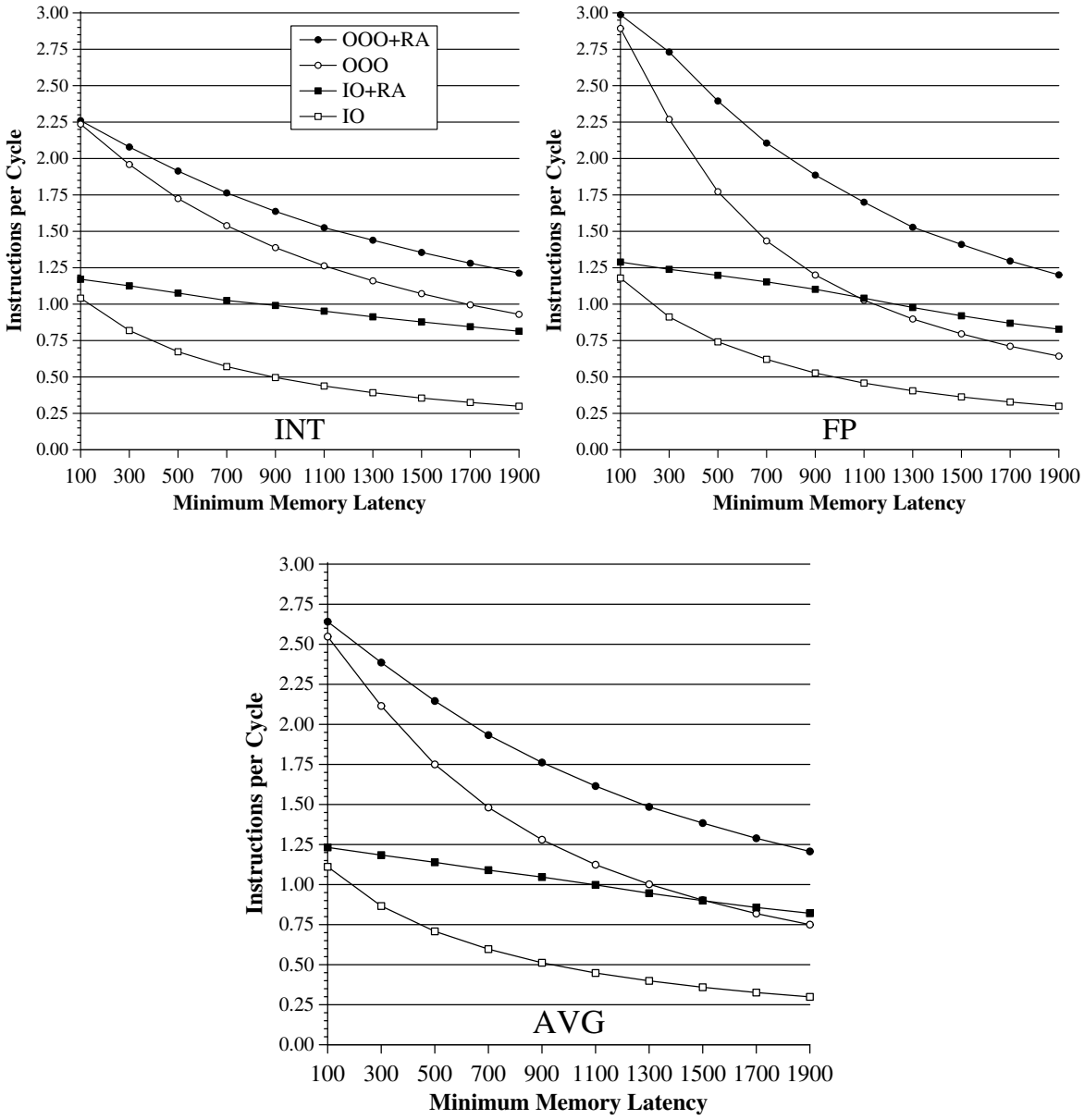


Figure 4.20: Effect of main memory latency on the runahead execution performance on in-order and out-of-order execution processors. OOO stands for out-of-order, IO for in-order, and RA for runahead execution.

4.3 Summary of Results and Conclusions

This chapter evaluated the proposed runahead execution mechanism on both x86 and Alpha processors and a wide variety of machine configurations. Our empirical analyses on a variety of processor models and a wide variety of benchmarks provide strong support for the following major conclusions:

- Runahead execution is more effective than and complementary to hardware-based prefetching which is commonly employed in current high-performance processors.
- Runahead execution provides the performance benefit of much larger instruction windows without the additional cost and complexity of the large structures needed to implement large instruction windows. With a 500-cycle memory latency, runahead execution on a 128-entry window processor achieves the performance of a conventional out-of-order execution processor with 3 times the instruction window size (384-entry window). With a 1000-cycle memory latency, it achieves the performance of a conventional processor with 8 times the instruction window size (1024-entry window).
- Although runahead execution is also very effective on in-order processors, an out-of-order processor with runahead execution by far provides the best performance.
- The performance benefit of runahead execution improves as main memory latency and the effectiveness of instruction supply increase. Since future processors will have longer memory latencies and better instruction supplies, we expect runahead execution to become more effective in future processors.

Chapter 5

Techniques for Efficient Runahead Execution

A runahead processor executes significantly more instructions than a traditional out-of-order processor in order to discover L2 cache misses in runahead mode. Because of the increase in executed instructions, a runahead processor can consume significantly more dynamic energy than a conventional processor. This section examines the causes of energy-inefficiency in runahead execution and proposes 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. In order to achieve this goal, this section seeks answers to the following questions:

- As a runahead processor speculatively 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 the performance improvement provided by runahead execution? In other words, how can the processor designer increase the *efficiency* of a runahead processor?
- As the speculative execution of instructions in runahead mode targets the discovery of useful L2 cache misses, instruction processing during runahead mode should be optimized for maximizing the number of useful L2 misses generated during runahead execution. What kind of techniques increase the probability of the generation of

useful L2 misses and hence increase the performance of a runahead processor, while reducing or not significantly increasing the number of instructions executed?

- Speculative pre-execution of the instruction stream in runahead mode generates the correct result values for some of the pre-executed instructions in addition to the L2 misses that will later be used by the application program. Our proposed runahead execution implementation discards the result values of all pre-executed instructions even though some of those result values are correct. Is it worthwhile to design a hardware mechanism to reuse the results of pre-executed instructions to improve both the performance benefit and the efficiency of runahead execution?

5.1 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 [43].

A runahead processor's efficiency can be increased in two ways:

- The number of executed instructions (the denominator) can be reduced without affecting the increase in IPC (the numerator) by eliminating the causes of inefficiency.
- 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.

This chapter proposes techniques that increase efficiency in both ways. We examine techniques that increase efficiency by reducing the *Percent Increase In Executed Instructions* in Section 5.2. Techniques that increase efficiency by increasing the *Percent Increase In IPC* are examined in Section 5.3.

Note that efficiency *by itself* is not a very meaningful metric since it does not show the performance improvement. 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.

[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 used in this dissertation, 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.

Therefore, we always report changes in these two metrics. Efficiency values can be easily computed using these two metrics.

Figure 5.1 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 $= 108.4/235.4 = 0.46$)

5.2 Eliminating the Causes of Inefficiency

We have identified three major causes of inefficiency in a runahead 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.

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%, since only those benchmarks that show significant changes in IPC or executed instructions can be affected

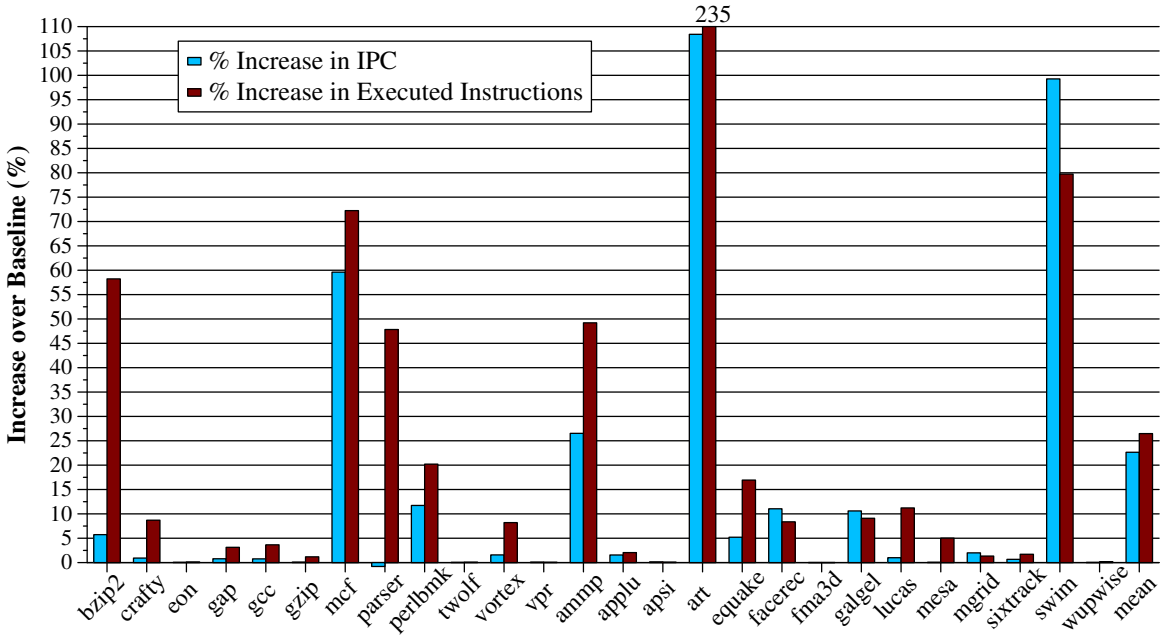


Figure 5.1: Increase in IPC and executed instructions due to runahead execution.

by techniques designed for improving efficiency. Results for all benchmarks are reported in Section 5.2.4.

5.2.1 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.

Figure 5.2 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 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 may not be able to execute enough instructions far ahead into the instruction stream and hence may not be able to discover any useful L2 cache misses during runahead mode.

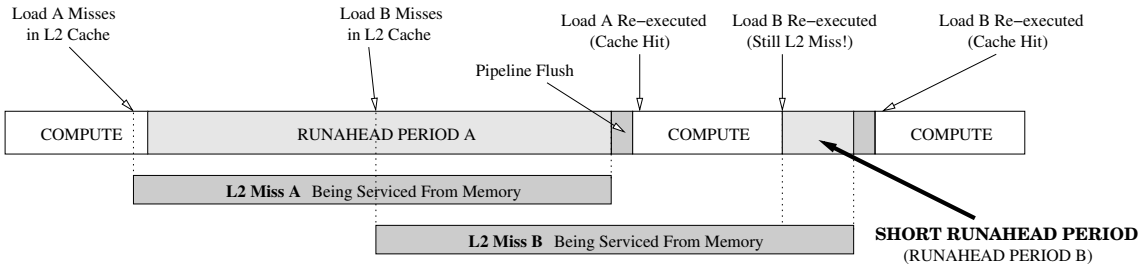


Figure 5.2: Example execution timeline illustrating the occurrence of a short runahead period.

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.[2] 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 the proposed 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

[2]While it is possible to have the memory system communicate the estimated latency of an L2 miss to the processor core after the DRAM access for that L2 miss is started, such communication would add unnecessary complexity to the memory system and buses between the core and main memory. Moreover, the estimate provided by the memory system cannot be exact due to the queueing delays the data may encounter on its way back to the processing core.

(MSHR) [62] is augmented with 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 enter 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 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.

Figures 5.3 and 5.4 show the increase in number of executed instructions and IPC over the baseline processor if the proposed thresholding mechanisms are employed. We show results for the best static and dynamic thresholds out of all we evaluated. 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.

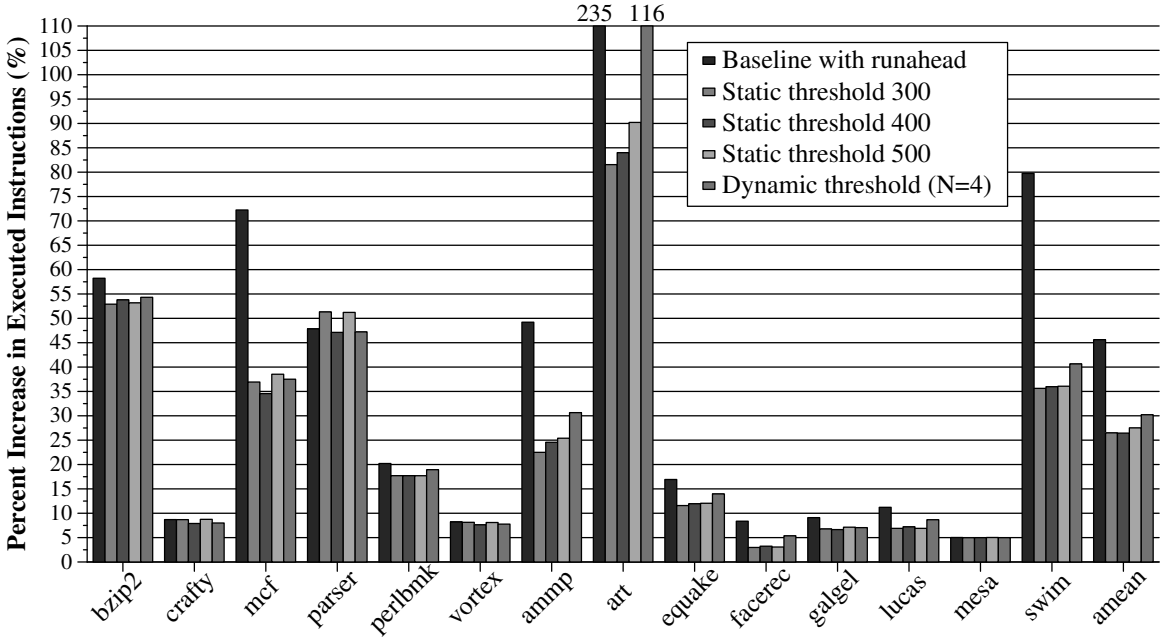


Figure 5.3: Increase in executed instructions after eliminating short runahead periods using static and dynamic thresholds.

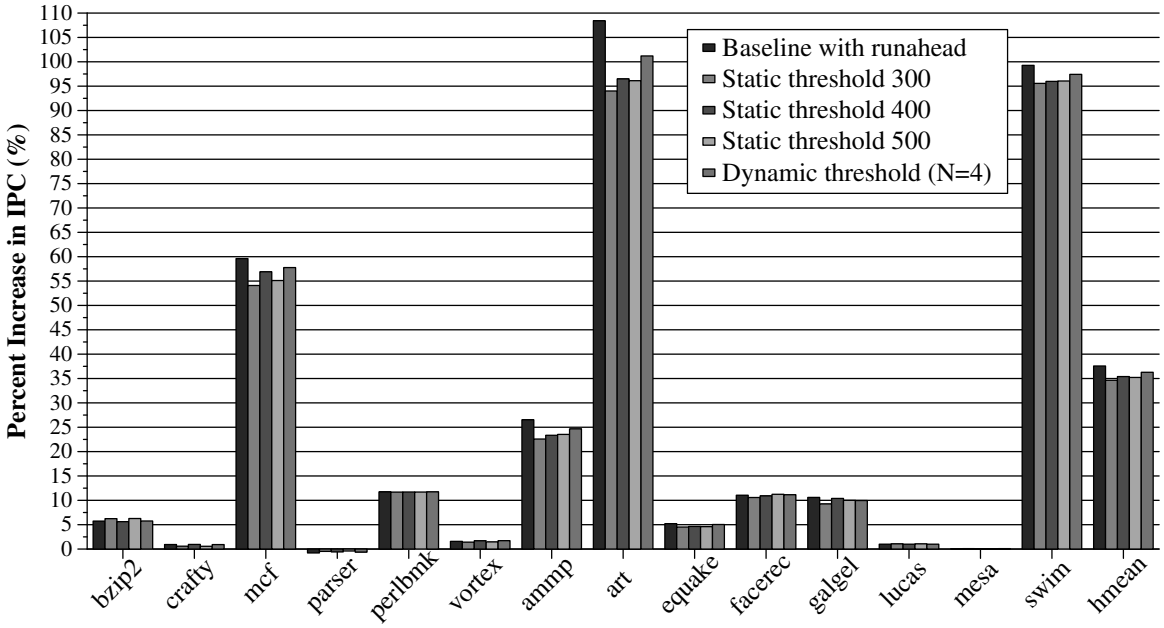


Figure 5.4: Increase in IPC after eliminating short runahead periods using static and dynamic thresholds.

Figure 5.5 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 is an average 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 (the leftmost points in the left graph in Figure 5.5). Using the *static threshold 400* mechanism eliminates all occurrences of periods that are shorter than 100 cycles, as shown in the right graph. Eliminating short periods also increases the occurrence of relatively longer periods (450-500 cycles), which are more efficient and effective in terms of the number of useful L2 misses they generate.[3] This mechanism also eliminates some of the very long runahead periods (longer than 600 cycles) that are useful, but we found that the loss in efficiency due to eliminating a smaller number of long useful periods is more than offset by the gain in efficiency due to eliminating a larger number of short useless periods.

[3]Many of the L2 misses that are discovered and started by a short runahead period are not complete when they are later needed in normal mode. Therefore, they cause entries into *more* short runahead periods. Once short runahead periods are eliminated using the thresholding mechanism, these L2 misses do not occur in runahead mode because the runahead periods that used to lead to them are eliminated. Instead, they are discovered in normal mode and cause entries into longer runahead periods. Hence the increase in the occurrence of relatively longer runahead periods.

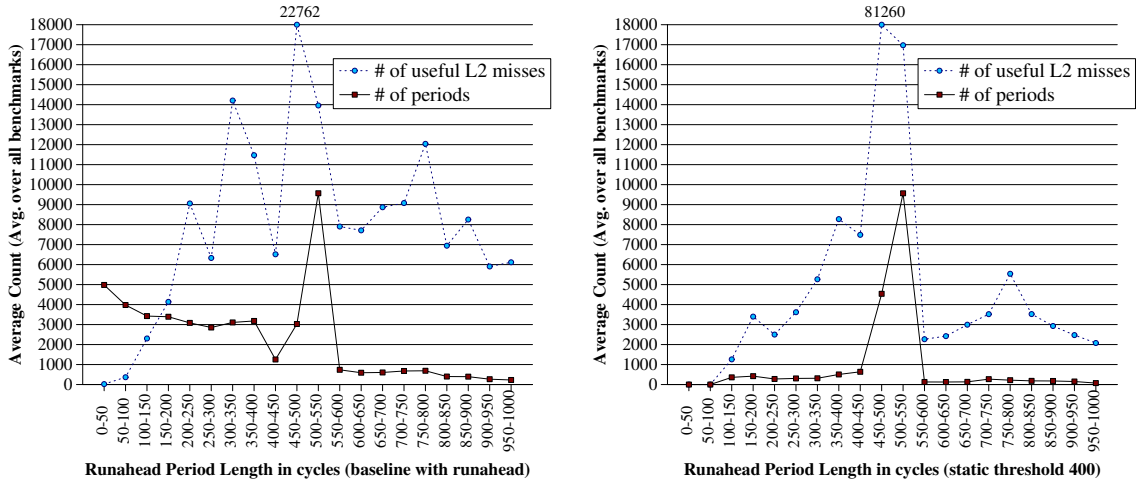


Figure 5.5: Distribution of runahead period length (in cycles) and useful misses generated for each period length.

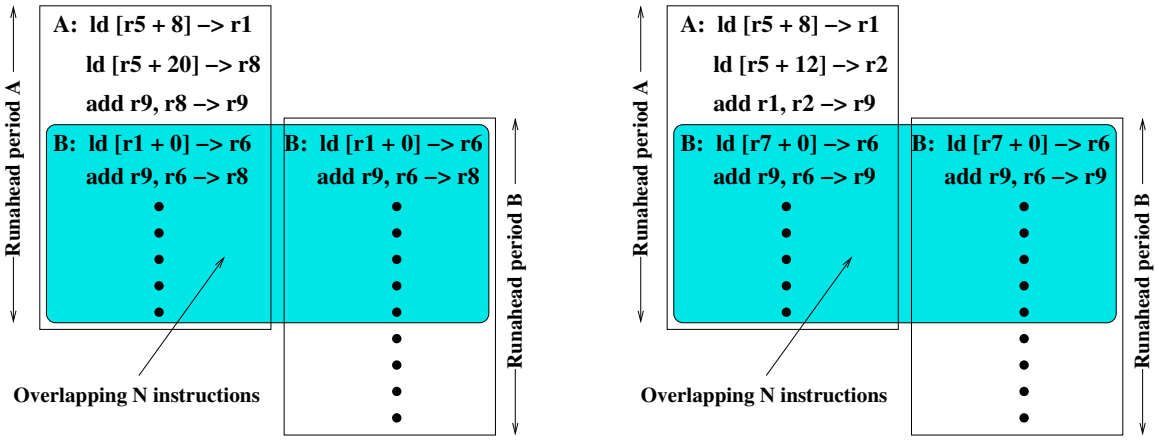
5.2.2 Eliminating Overlapping Runahead Periods

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 5.6a):* The execution timeline showing the occurrence of overlapping periods due to dependent L2 misses is given in Figure 5.7. Load A causes entry into runahead period A. During this period, the processor executes Load B and finds that it is dependent on the miss caused by Load A. Because the processor has not serviced the miss caused by Load A yet, it cannot calculate Load B's address and therefore it marks Load B as INV. The processor executes and pseudo-retires N instructions after Load B, and exits runahead period A when the miss for Load A returns from main memory. The pipeline is flushed and fetch is redirected to Load A. In normal mode, the processor re-executes Load B and finds it to be an L2 miss,

which causes runahead period B. In runahead period B, the processor executes the same N instructions that were executed in period A.

2. *Independent L2 misses with different latencies (Figure 5.6b):* This is similar to the previous case, except Load A and Load B are independent. The L2 miss caused by Load B takes longer to service than the L2 miss caused by Load A. Note that runahead period B may or may not also be a *short* period, depending on the latency of the L2 cache miss due to Load B.



(a) Due to dependent L2 misses

(b) Due to independent L2 misses

Figure 5.6: Code example showing overlapping runahead periods.

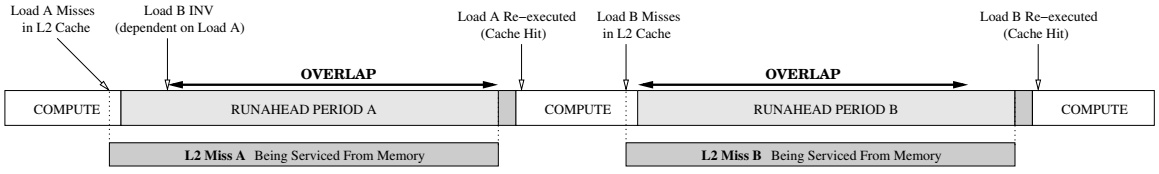


Figure 5.7: Example execution timeline illustrating the occurrence of an overlapping runahead period.

Overlapping runahead periods can benefit 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 prefetches that could not have been generated in runahead period A (because the result of Load A was not available in runahead period A). However, in the benchmark set we examined, overlapping runahead periods rarely benefited performance. 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 L2 misses, 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*). This technique can be implemented with two simple counters and a comparator.

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. 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 elimi-

nate some non-overlapping runahead periods. To reduce the probability of eliminating non-overlapping periods, we examined two other policies (*half threshold policy* and *75% threshold policy*) where the processor does not enter runahead mode if it has not fetched more than half (or 75%) of the number of instructions pseudo-retired during the last runahead period.

Figures 5.8 and 5.9 show 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 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 relatively 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 in these benchmarks 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.

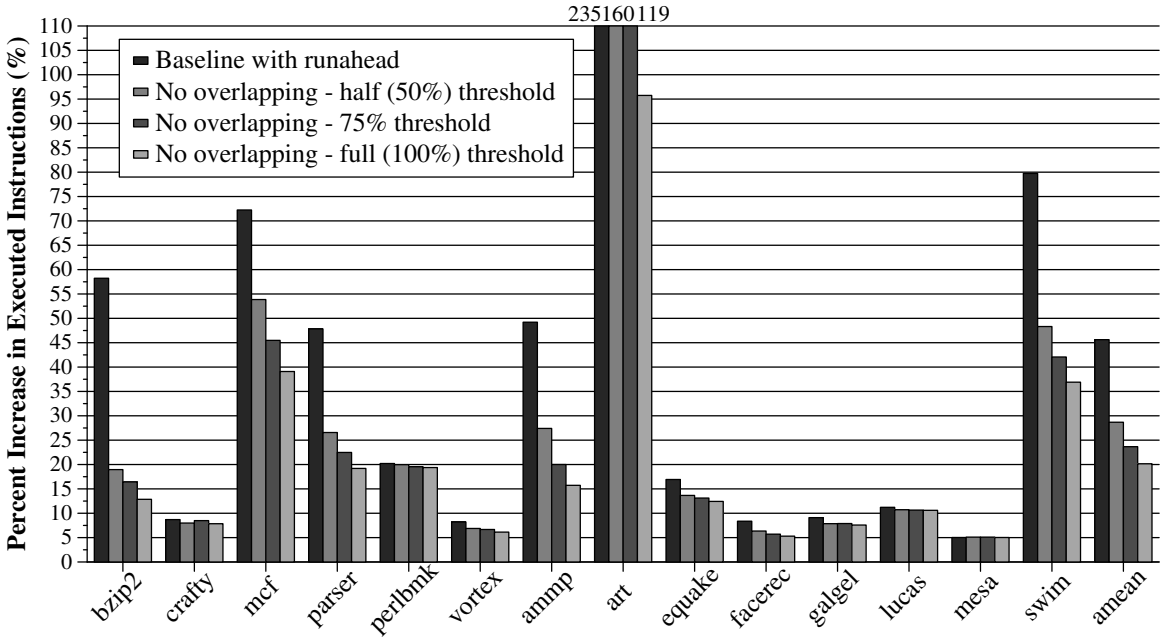


Figure 5.8: Increase in executed instructions after eliminating overlapping runahead periods using thresholding.

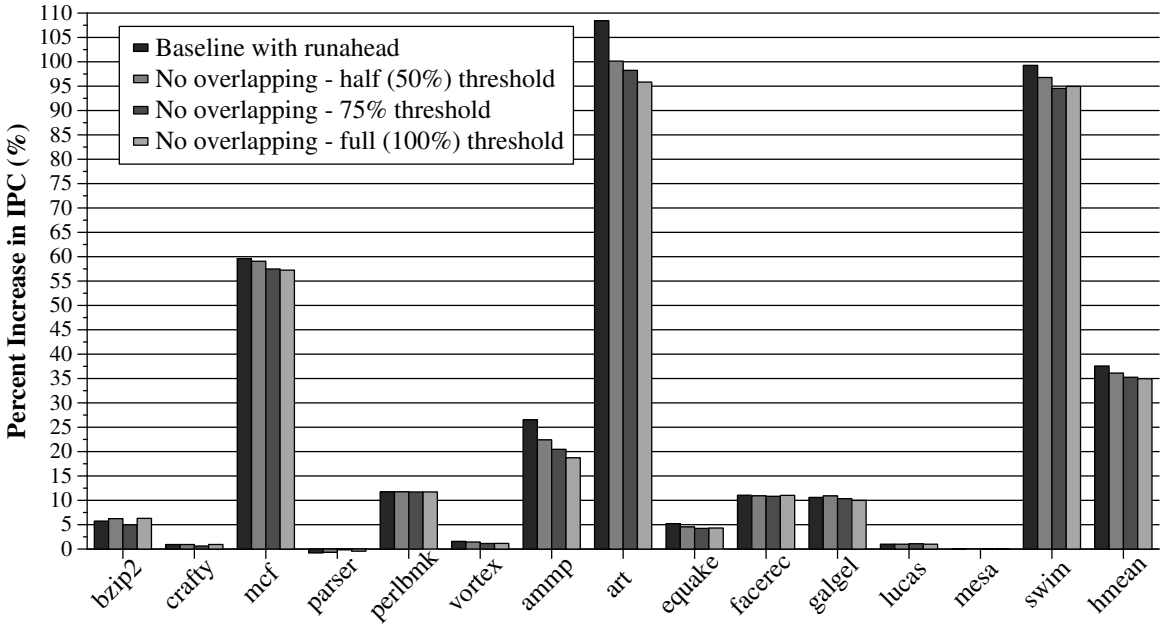


Figure 5.9: Increase in IPC after eliminating overlapping runahead periods periods using thresholding.

5.2.3 Eliminating Useless Runahead Periods

Useless runahead periods are those runahead periods in which no useful L2 misses that are needed by normal mode execution are generated, as shown in Figure 5.10. These periods exist due to the lack of memory-level parallelism [42, 23] 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).

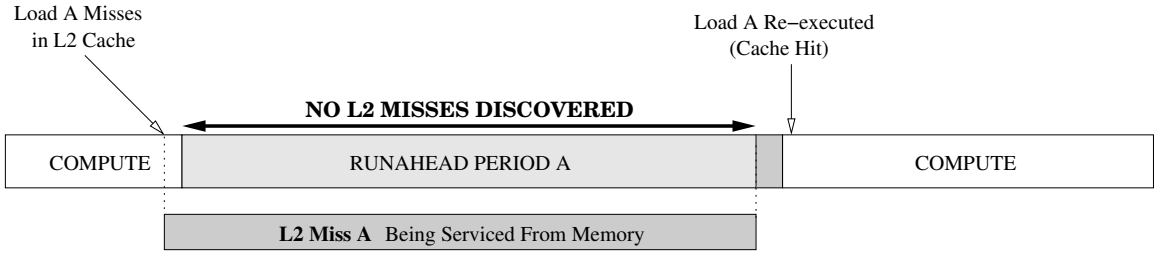


Figure 5.10: Example execution timeline illustrating a useless runahead period.

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 future. We propose four simple mechanisms for predicting whether or not a runahead period generate a useful L2 cache miss.

5.2.3.1 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 due to 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.[4] The insight behind this technique is that the usefulness of future runahead periods tend to be predictable based on the recent past behavior of runahead periods caused by the same static load. The processor uses a table of two-bit saturating counters called Runahead Cause Status Table (RCST) to collect information on the usefulness of runahead periods caused by each L2-miss load. The state diagram for an RCST entry is shown in Figure 5.11.

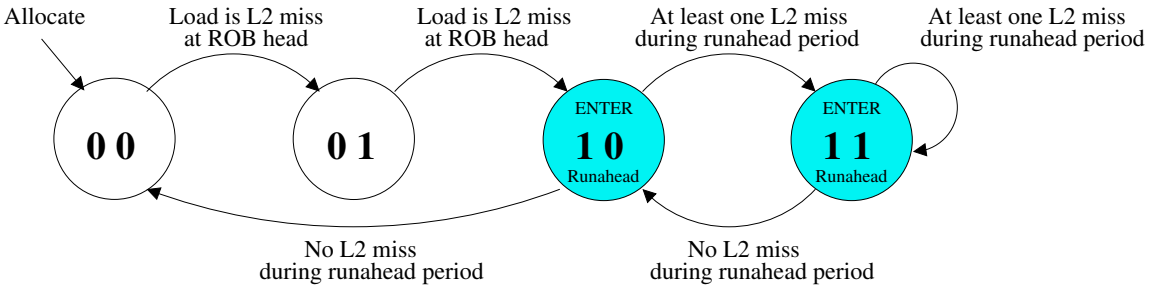


Figure 5.11: State diagram of the RCST counter.

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 mode 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.[5] Upon exit from runahead mode, if there was at least one such L2 load miss generated during runahead mode, the two-bit counter associated

[4]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. However, we found that the heuristic works well because wrong-path L2 cache misses generated in runahead mode tend to be useful for normal mode execution [77].

[5]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 size, cannot be captured by the instruction window. N=128 in our simulations.

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 the counter is not incremented, the load will never cause entry into runahead mode until its counter is evicted from RCST, which we found to be detrimental to performance because it eliminates many useful runahead periods along with useless ones. In our experiments, we used a 4-way RCST containing 64 counters.[6]

5.2.3.2 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, the processor keeps a count of the number of load instructions executed in runahead mode and a count of how many of those were INV (i.e., dependent on an L2 miss). After N cycles of execution during runahead mode, the processor starts checking the fraction INV loads out of all executed loads in the ongoing runahead mode. If the fraction 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 0.75, 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 dur-

[6]We examined other two, three, and four-bit state machines with various update policies and found that the one shown in Figure 5.11 performs the best in terms of efficiency. We also examined a tagless RCST organization. This resulted in lower performance than the tagged organization due to destructive interference between different load instructions.

ing runahead mode are INV, this is used as an indication that the current runahead period will not generate useful L2 cache misses and therefore the processor is better off exiting runahead mode. This technique is called the *INV Load Count* technique.

5.2.3.3 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 found that usefulness of runahead periods exhibits more coarse-grain, phase-like behavior. Runahead execution tends to consistently generate or not generate L2 cache misses in a large number of consecutive periods. This behavior is due to: (1) the phase behavior in benchmarks [35] where some phases show high memory-level parallelism and others do not, (2) the clustering of L2 cache misses [23].

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. 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. The pseudo-code for the sampling algorithm is given in Figure 5.12. 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 dissertation. However, this dissertation

demonstrates that even with untuned values of parameters, it is possible to significantly increase the efficiency of a runahead execution processor without significantly impacting performance.

```
When a process is loaded after context switch DO
 {
   mode = MONITOR_MODE;
   periods_monitored = 0;
   L2_misses_seen = 0;
 }
```

```
When an L2 miss load is at the head
    of the instruction window DO
{
  if (mode == MONITOR_MODE) {
    ENTER runahead;
  } else {
    DO NOT ENTER runahead;
    opportunities_seen++;

    if (opportunities_seen == M) {
      periods_monitored = 0;
      L2_misses_seen = 0;
      mode = MONITOR_MODE;
    }
  }
}
```

```
When exit from runahead mode is complete DO
{
  ASSERT(mode == MONITOR_MODE);

  periods_monitored++;
  L2_misses_seen += L2_misses_seen_in_this_period;

  if (periods_monitored == N) {
    if (L2_misses_seen >= T) {
      periods_monitored = 0
    } else {
      opportunities_seen = 0;
      mode = SLEEP_MODE;
    }
  }
}
```

Figure 5.12: Pseudo-code for the sampling algorithm.

Figures 5.13 and 5.14 show 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. The *Sampling* technique results in significant reductions in executed instructions in especially bzip2 and parser, two benchmarks for which runahead execution is very inefficient. This technique increases the IPC improvement slightly in bzip2. All three techniques together increase the efficiency more than each individual technique, indicating 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%.

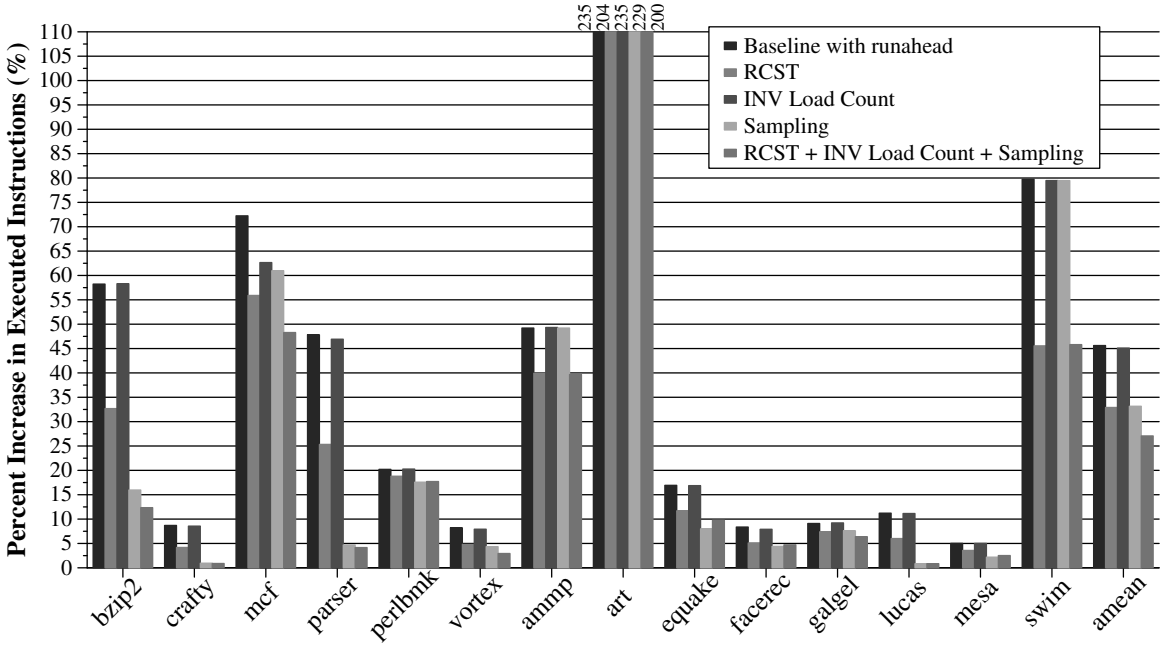


Figure 5.13: Increase in executed instructions after eliminating useless runahead periods with the dynamic uselessness prediction techniques.

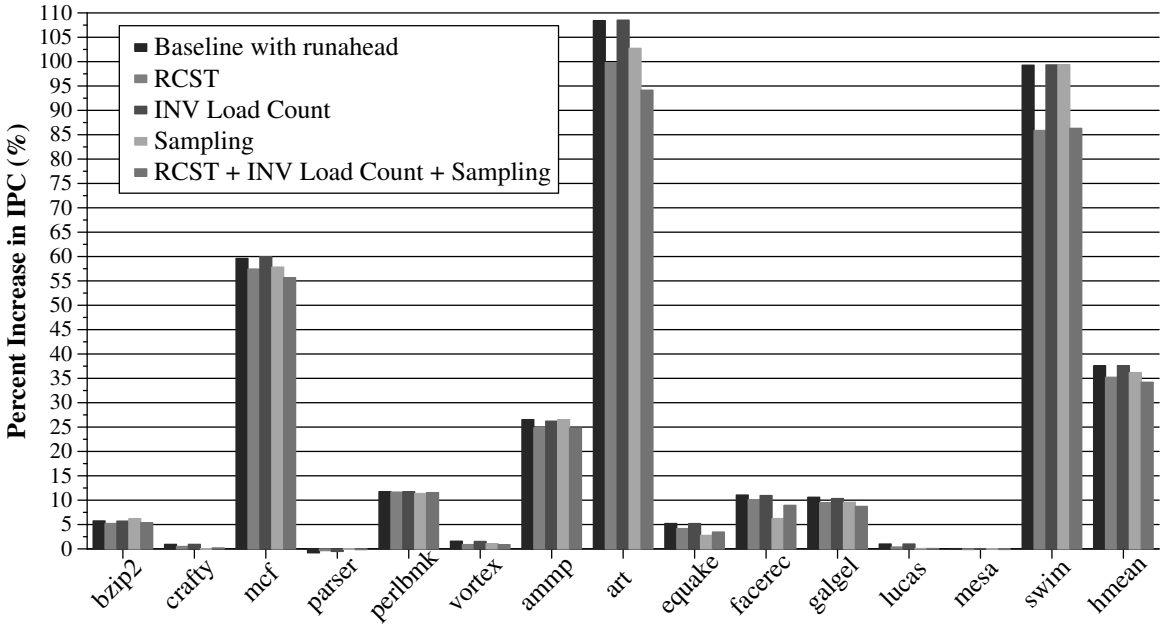


Figure 5.14: Increase in IPC after eliminating useless runahead periods with the dynamic uselessness prediction techniques.

5.2.3.4 Compile-Time Techniques to Eliminate Useless Runahead Periods Caused by a Static Load

Some static load instructions rarely lead to the generation of useful 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 useful 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 that cannot cause entry into runahead mode) by the compiler after code analysis and/or profiling runs. This section 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 that is augmented in the load instruction format of the ISA.[7] 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 enter runahead mode. If loads identified as *non-runahead* exhibit similar behavior in the profiling runs and the normal runs, the efficiency and perhaps the performance of the runahead processor would be improved. In our experiments, we examined threshold values of 0.1, 0.25, and 0.5, and found that 0.25 yields the best average efficiency value. In

[7] If a load instruction never causes entry into runahead during the profiling run, it is not marked as a *non-runahead load*.

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.

Figures 5.15 and 5.16 show 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 by itself, the processor always enters runahead mode when the L2-miss load at the head of the instruction window is not marked as *non-runahead*. 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*. In both cases, loads marked as *non-runahead* never cause entry into runahead mode. For comparison, the rightmost bar shows the effect of only using the previous three uselessness prediction techniques without profiling.

The figures show 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.[8] 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 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.

[8]In Figures 5.15 and 5.16, "same input set" means that the same input set was used for the profiling run and the simulation run; "different input set" means that different input sets were used for the two runs. For the "different input set" case, we used the train input set provided by SPEC for the profiling run.

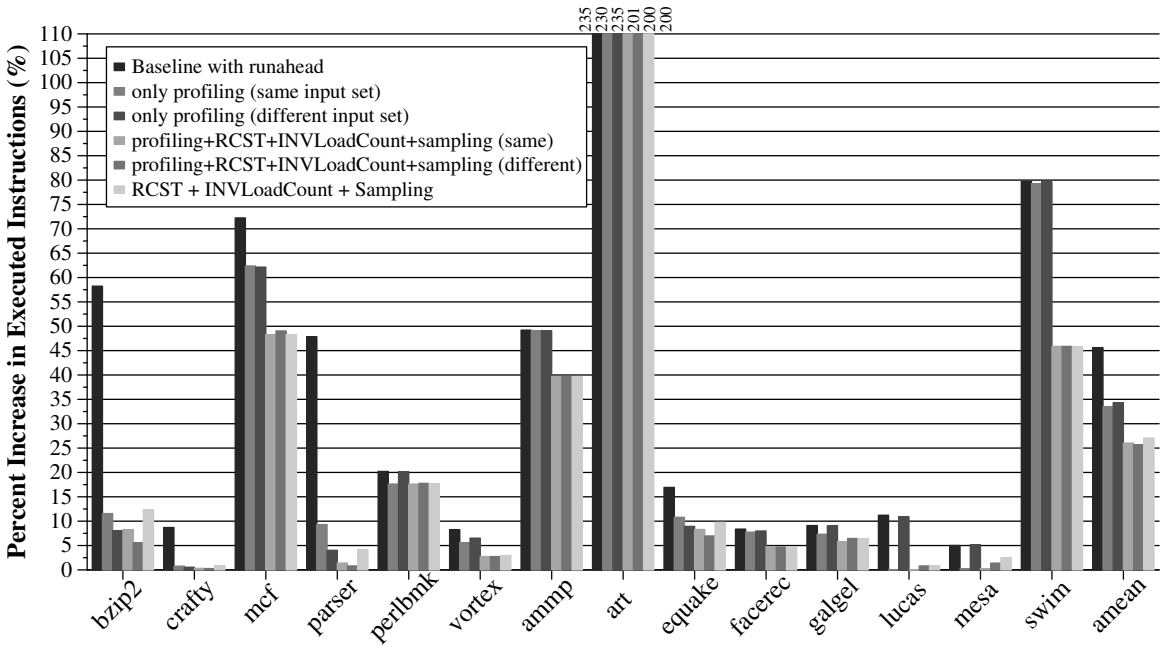


Figure 5.15: Increase in executed instructions after using compile-time profiling to eliminate useless runahead periods.

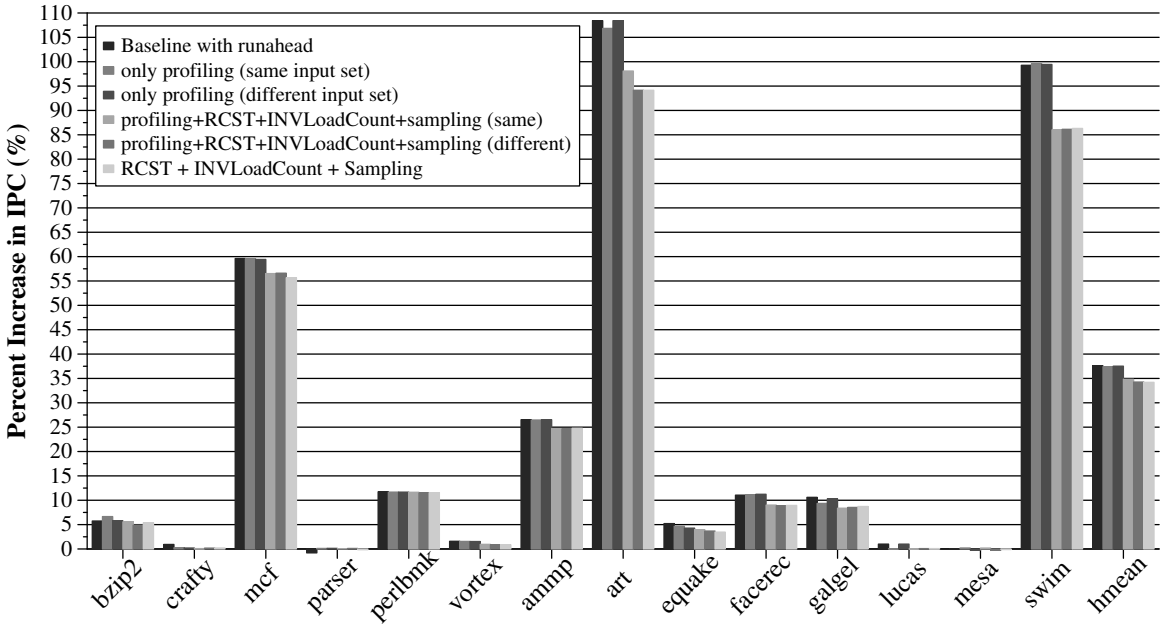


Figure 5.16: Increase in IPC after using compile-time profiling to eliminate useless runahead periods.

5.2.4 Combining the Efficiency Techniques

So far, we have considered individually each technique that aims to eliminate a particular cause of inefficiency. The causes of efficiency are sometimes not disjoint from each other. For example, a short runahead period may also be a useless runahead period because fewer instructions tend to be executed in short periods, probably resulting in the generation of no useful L2 misses during that period. Similarly, an overlapping runahead period may also be a short runahead period especially if it occurs due to independent L2 misses with different latencies (See Section 5.2.2). However, in some cases these causes of inefficiency can be disjoint from each other and therefore combining the techniques proposed in previous sections may result in increased efficiency. This section examines the effect of combining the different proposed techniques. The results show that a combination of all the proposed techniques increases efficiency more than each technique by itself.

Figures 5.17 and 5.18 show 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 CPU2000 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.[9] The average increase in the number of executed 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 but 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).

[9] We examine profiling separately since it requires modifications to the ISA or a dynamic optimization framework.

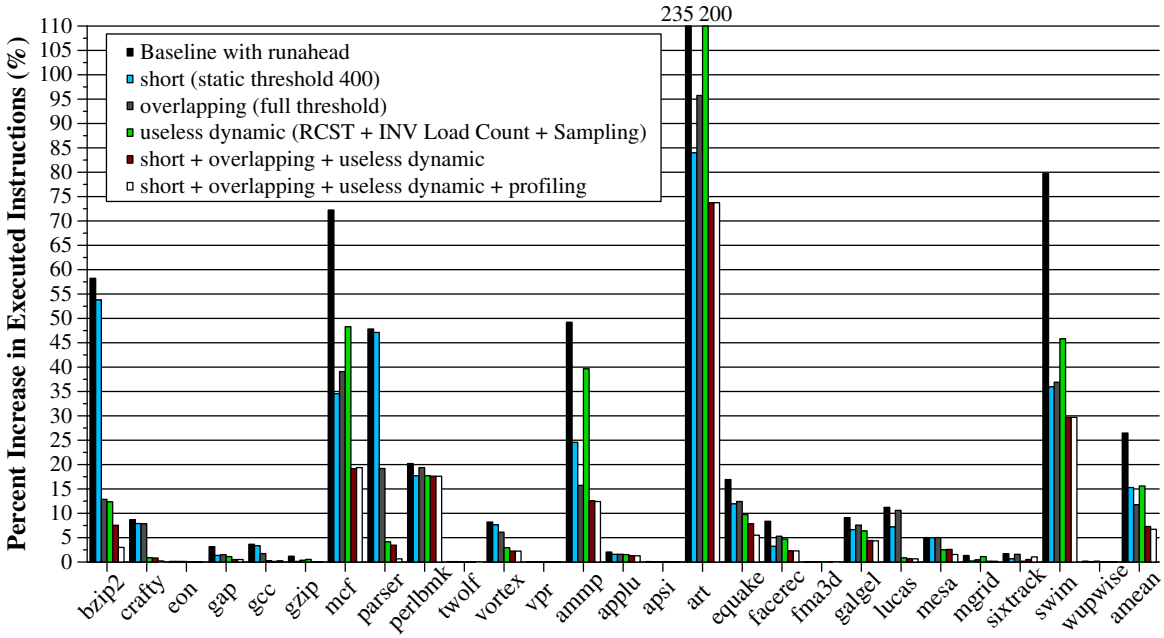


Figure 5.17: Increase in executed instructions after using the proposed efficiency techniques individually and together.

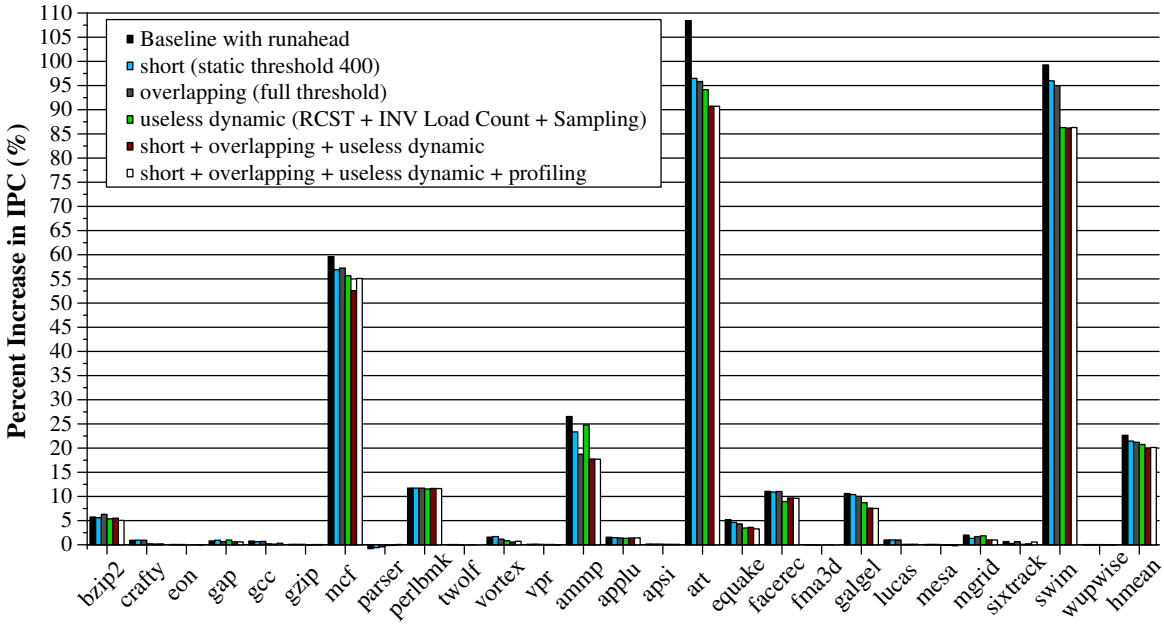


Figure 5.18: Increase in IPC after using the proposed efficiency techniques individually and together.

5.3 Increasing the Usefulness of Runahead Periods

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 [84, 23], 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.

5.3.1 Turning Off the Floating Point Unit During Runahead Mode

Since the purpose 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. 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.[10] 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.[11] 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 integer registers, are executed and are treated as prefetch instructions so that they can potentially generate cache

[10] In fact, FP operate instructions can be dropped immediately after fetch during runahead mode, if extra decode information is stored in the instruction cache indicating whether an instruction is an FP instruction.

[11] An FTOI instruction that 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. An FTOI instruction is handled this way so that its INT destination register is marked as INV.

misses. Their execution is accomplished in the load/store unit, just like in a traditional out-of-order processor. The impact of turning off the FP unit in runahead mode on performance and efficiency is evaluated in the next section.

5.3.2 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. The baseline runahead execution implementation does not take advantage of this property. In the baseline implementation, an instruction waits until all its source operands become ready before being scheduled, even if the first-arriving source operand is INV because INV bits exist only in the physical register file, which is accessed only after the instruction is scheduled. 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 L2 misses.[12] Another advantage of this mechanism is that it allows for faster processing of the INV dependence chains, which results in the fast removal of useless INV instructions from the reorder buffer. 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 the baseline runahead execution implementation, INV bits were only present in

[12]Note that this mechanism may not always result in the early wake-up of an INV instruction, since other sources of an instruction may already be ready at the time a source becomes INV.

the physical register file. In contrast, *early INV wake-up* requires INV bits to be also 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.

Figures 5.19 and 5.20 show 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. INT benchmarks do not have many FP instructions, therefore turning off the FP unit does not help their performance. *Early INV wake-up* benefits 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 INT 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 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.

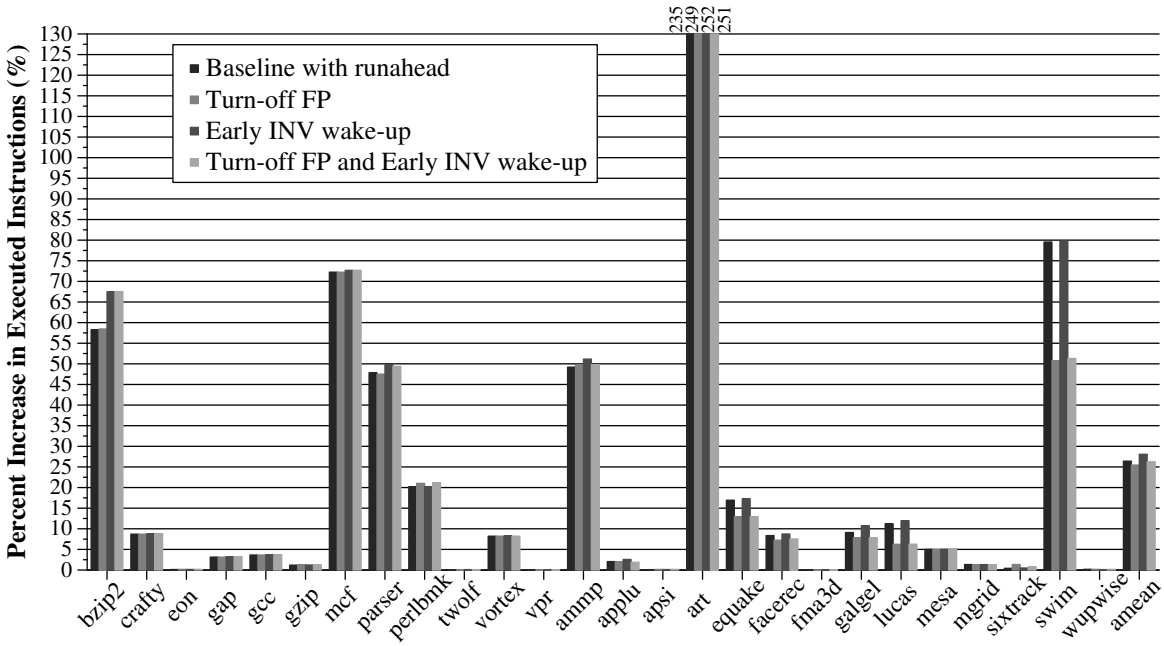


Figure 5.19: Increase in executed instructions after turning off the FP unit in runahead mode and using early INV wake-up.

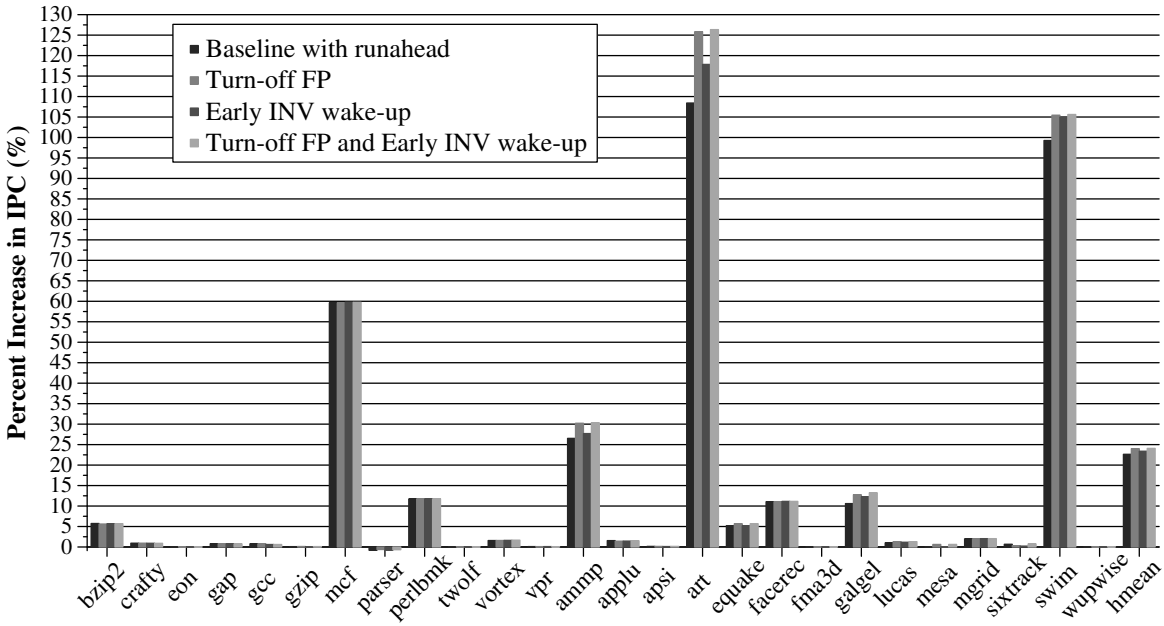


Figure 5.20: Increase in IPC after turning off the FP unit in runahead mode and using early INV wake-up.

5.3.3 Optimizing the Prefetcher Update Policy

One of the potential benefits of runahead execution is that the hardware 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. Moreover, inaccurate hardware prefetcher requests can cause resource contention for the more accurate runahead load/store memory requests during runahead mode and thus reduce runahead execution's effectiveness.

This dissertation showed that runahead execution and hardware prefetching have synergistic behavior (see Sections 4.1.3.1 and 4.2.5.1), a result that was independently verified by other researchers [51]. This section optimizes the prefetcher update policy to increase the synergy between the two prefetching mechanisms. We experiment with three different policies to determine the impact of prefetcher update policy on the performance and efficiency of a runahead processor.

The baseline runahead execution 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. That is, L2 accesses do not train the stream buffers and L2 misses do not create new streams in runahead mode (*no train, no create* policy). The last policy we examine allows the training of existing streams, but disables the creation of new streams in runahead mode (*only train* policy).

Figures 5.21 and 5.22 show 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 benchmarks 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 that 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 significantly reduces the performance improvement of runahead execution in benchmarks applu, equake, facerec, lucas, mgrid, and swim. 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 beneficial. In these benchmarks, the main benefit of useful runahead execution periods comes from increasing the timeliness of the prefetches generated by the hardware 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.

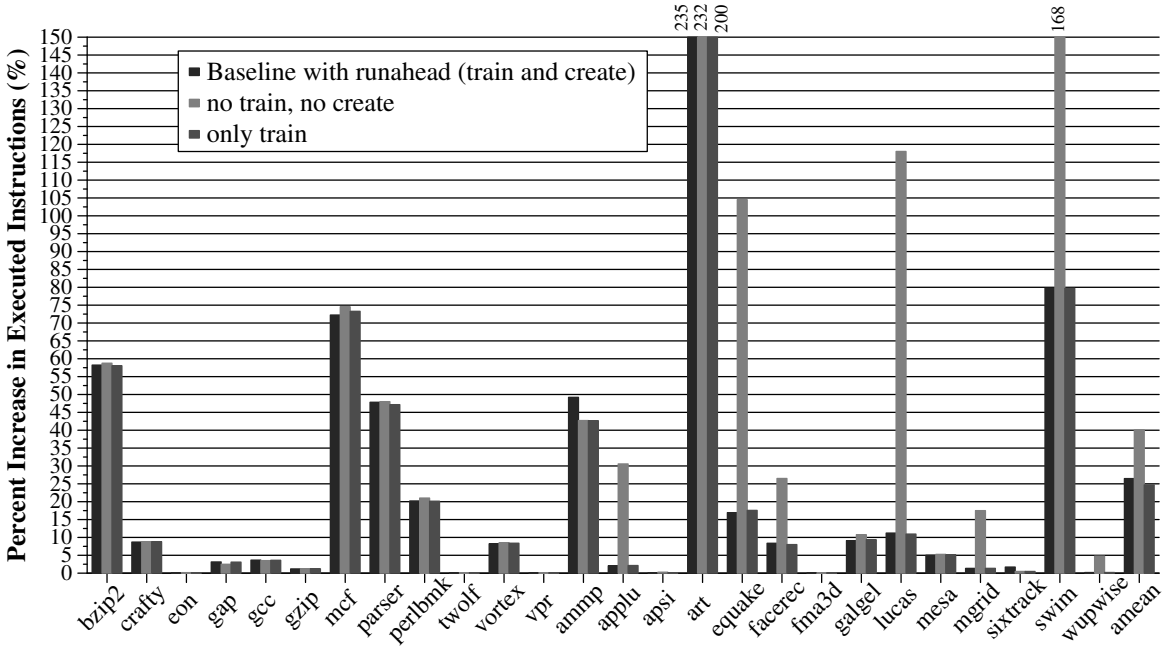


Figure 5.21: Increase in executed instructions based on prefetcher training policy during runahead mode.

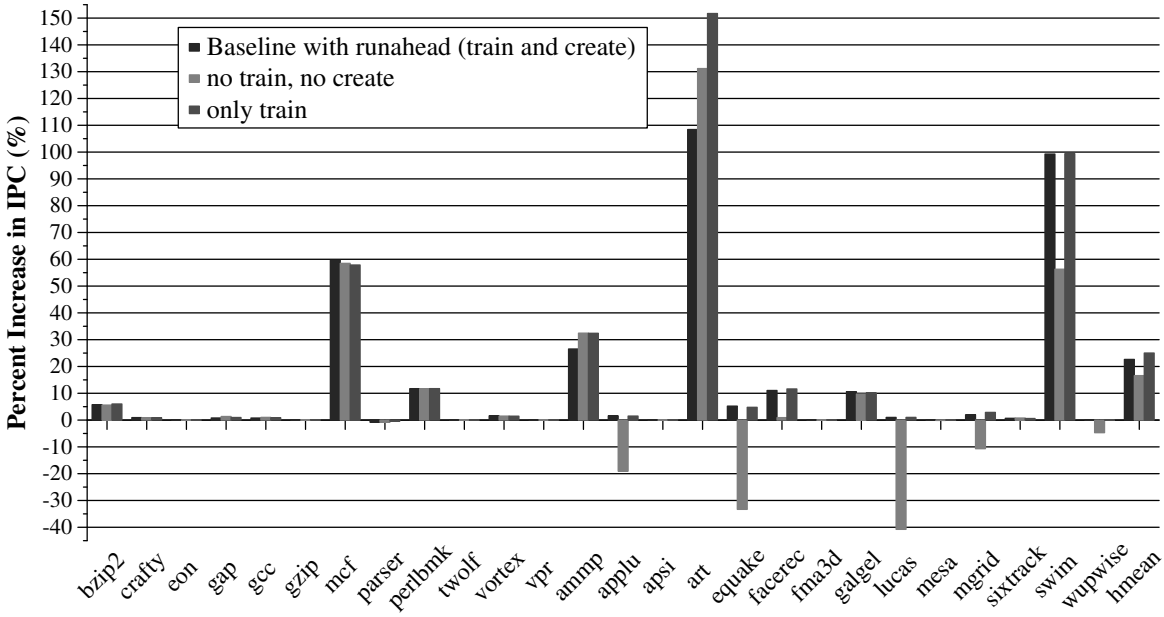


Figure 5.22: Increase in IPC based on prefetcher training policy during runahead mode.

5.4 Putting It All Together: Efficient Runahead Execution

This section evaluates the overall effect of the efficiency and performance enhancement techniques proposed for efficient runahead execution in Sections 5.2 and 5.3. We consider implementing the following efficiency and performance optimizations together on the baseline runahead execution processor:

- The *static threshold 400* (Section 5.2.1) to eliminate short runahead periods
- The *full threshold* policy (Section 5.2.2) to eliminate overlapping runahead periods
- *RCST*, *INV Load Count*, and *Sampling* (Section 5.2.3) to eliminate useless runahead periods
- Turning off the FP unit (Section 5.3.1) to increase performance and save energy
- The *only train* policy (Section 5.3.3) to increase the usefulness of runahead periods

We also evaluate the impact of using profiling to eliminate useless runahead periods (Section 5.2.3.4) in addition to these dynamic techniques.

Figures 5.23 and 5.24 show the increase in executed instructions and IPC over the baseline processor when the mentioned techniques are applied. 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 that 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.

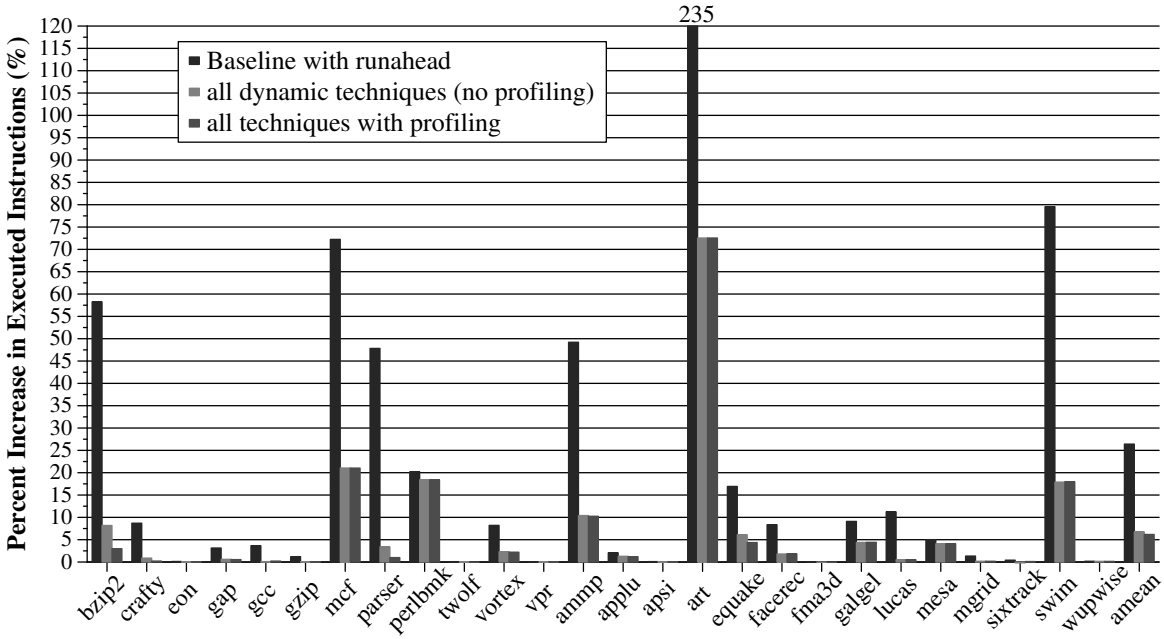


Figure 5.23: Increase in executed instructions after all efficiency and performance optimizations.

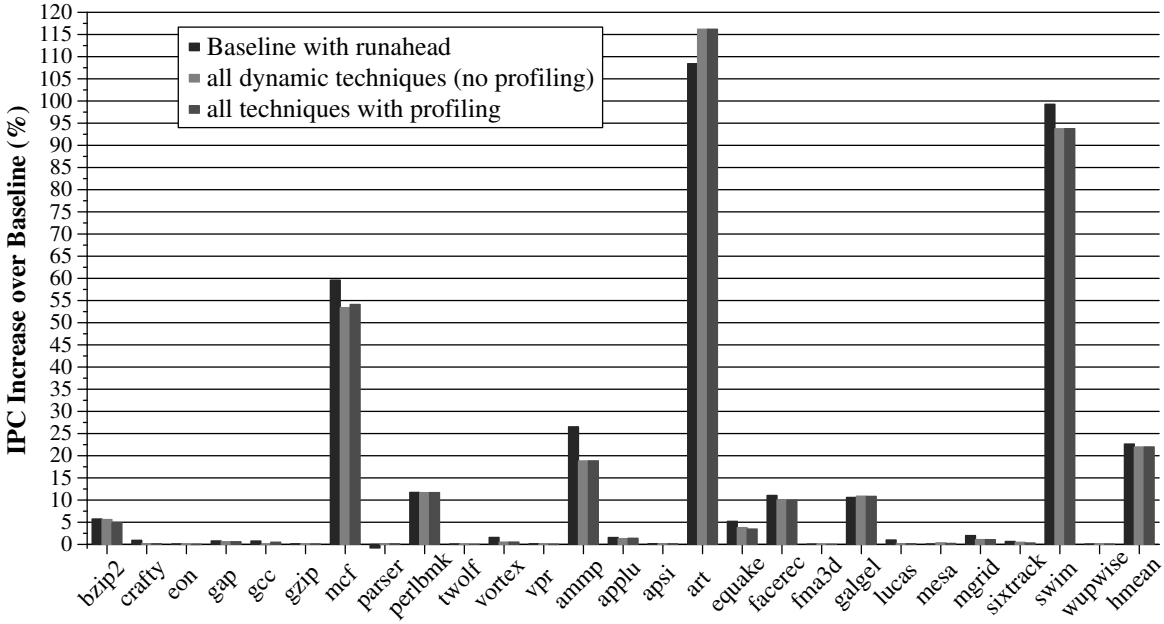


Figure 5.24: Increase in IPC after all efficiency and performance optimizations.

5.4.1 Effect of Main Memory Latency

This section examines the effectiveness of using the proposed dynamic techniques with different memory latencies. The left graph in Figure 5.25 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 5.25 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 IPC improvement and extra 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 when the parameters used in the techniques are not tuned.

5.4.2 Effect of Branch Prediction Accuracy

Section 4.1.3.4 showed that the performance improvement due to runahead execution increases significantly with a better branch predictor. Here, we examine the effectiveness of the proposed efficiency techniques on a baseline with perfect branch prediction.

Figures 5.26 and 5.27 show the increase in executed instructions and IPC due to runahead execution on a baseline with perfect branch prediction. Runahead execution on this baseline increases the IPC by 29.7% at a cost of 30.8% extra instructions. Comparing this to the IPC increase (22.6%) and extra instructions (26.5%) due to runahead on the baseline with a real branch predictor, we can conclude that runahead execution is more efficient

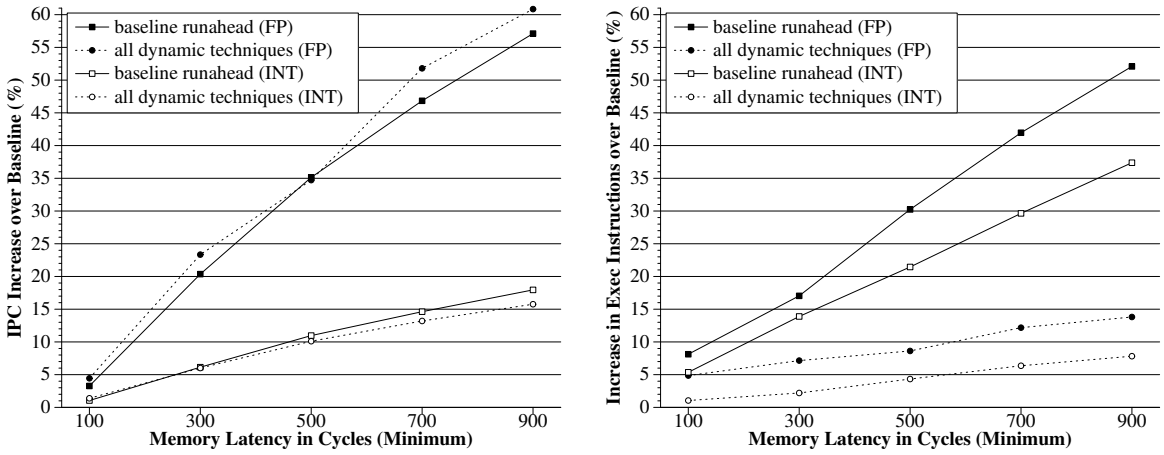


Figure 5.25: Increase in IPC and executed instructions with and without the proposed techniques (no profiling).

on a machine with improved branch prediction. This is expected because perfect branch prediction significantly increases the performance improvement of runahead execution (as shown in Section 4.1.3.4) and it also eliminates the wrong-path instructions executed in runahead mode.

Incorporating the proposed efficiency techniques on a runahead processor with perfect branch prediction significantly reduces the extra instructions down to 7.4% while slightly reducing the IPC improvement to 27.9%. We conclude that the proposed efficiency techniques remain very effective on a runahead processor with better branch prediction.

5.4.3 Effect of the Efficiency Techniques on Runahead Periods

The proposed efficiency techniques were designed to increase efficiency by reducing the number of ineffective runahead periods and increasing the usefulness of runahead periods. Figures 5.28 and 5.29 provide supporting data showing that the techniques are effective in achieving these goals.

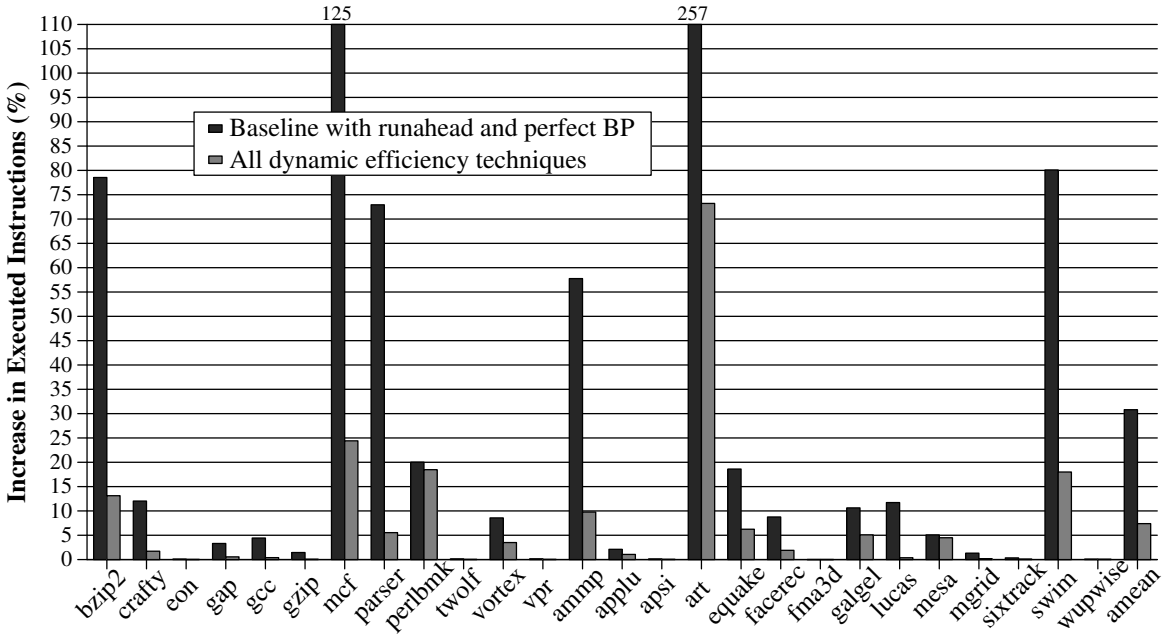


Figure 5.26: Increase in executed instructions with and without the proposed efficiency techniques on a baseline with perfect branch prediction.

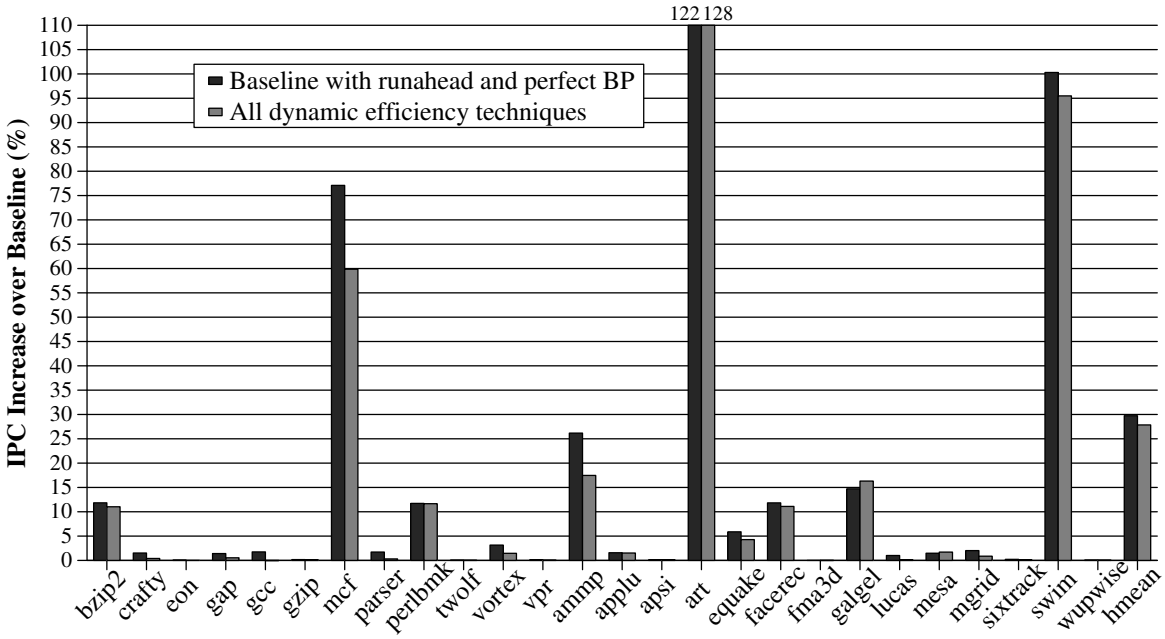


Figure 5.27: Increase in IPC with and without the proposed efficiency techniques on a baseline with perfect branch prediction.

Figure 5.28 shows the number of useful and useless runahead periods per 1000 retired instructions before and after applying the proposed dynamic efficiency techniques. The baseline runahead processor enters runahead mode 0.2 times per 1000 instructions, on average. Only 36% of all these periods are useful (i.e., they lead to the generation of at least one L2 cache miss that will be used in normal mode). When the proposed efficiency techniques are applied, the efficient runahead processor enters runahead mode 0.04 times per 1000 instructions. Hence, 79% of the runahead periods are eliminated using the efficiency techniques. Only 15% of the periods in the efficient runahead processor are useless. Therefore, the efficiency techniques are effective at eliminating useless runahead periods which do not result in the discovery of L2 misses that are beneficial for performance.

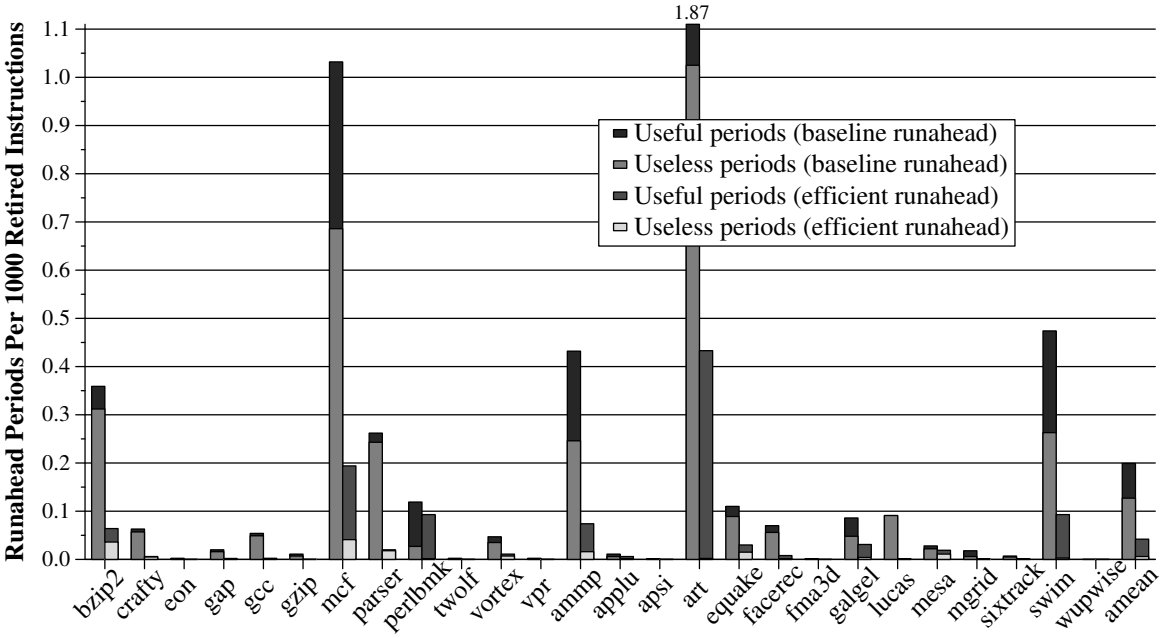


Figure 5.28: The number of useful and useless runahead periods per 1000 instructions with and without the proposed dynamic efficiency techniques.

Figure 5.29 shows the effect of the efficiency techniques on the average number of useful L2 cache misses parallelized in a useful runahead period. The baseline runa-

head processor discovers an average of 7.05 useful L2 misses in a useful runahead period. In contrast, the efficient runahead processor discovers useful 13.8 L2 misses in a useful runahead period. Thus, the effectiveness of each useful runahead period is increased with efficient runahead execution. This is due to two reasons. First, techniques that are designed to increase the usefulness of runahead periods increase the number of L2 cache misses discovered by useful runahead periods. Second, techniques that are designed to eliminate the causes of efficiency result in the elimination of some relatively less useful periods that discover only a single L2 cache miss. For example, eliminating short runahead periods eliminates not only useless short runahead periods, but also some relatively less useful short runahead periods as shown in Figure 5.5. Eliminating the relatively less useful runahead periods increases the average number of L2 misses discovered in a runahead period, but it does not impact performance because the performance loss due to it is balanced by the performance gain due to increased usefulness in other runahead periods.

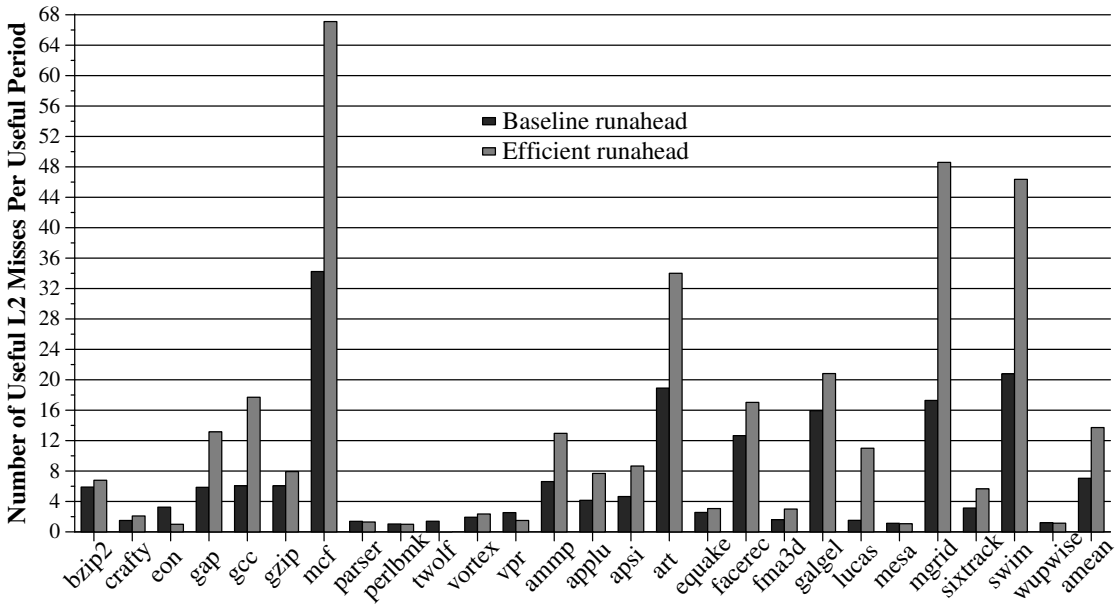


Figure 5.29: Average number of useful L2 misses discovered in a useful runahead period with and without the proposed dynamic efficiency techniques.

5.5 Other Considerations for Efficient Runahead Execution

The proposed efficiency techniques are effective at improving the efficiency of a runahead execution processor. This section describes orthogonal ways of improving runahead efficiency. We found that these techniques could potentially be useful for improving runahead efficiency, but their current implementations either are not cost-effective or they do not improve efficiency significantly.

5.5.1 Reuse of Instruction Results Generated in Runahead Mode

The evaluations in this dissertation so far took runahead execution for granted as a prefetch-only technique. Even though the results of most instructions independent of an L2 miss are correctly computed during runahead mode, the described runahead implementation discards those results instead of trying to utilize them in normal mode execution. Discarding the instruction results generated during runahead mode, as opposed to buffering them and reusing them in normal mode, is very appealing due to two reasons. First, it simplifies the implementation of a runahead processor. Any attempt at reusing the results generated during runahead mode needs to provide storage for those results and needs to provide a mechanism through which those results are incorporated into the processor state (register file and memory) during normal mode. These requirements increase the area, design complexity, and probably the peak power consumption of the runahead processor.

Second, discarding the instruction results generated during runahead mode decreases the verification complexity of a runahead processor. If results of instructions are not reused, runahead execution does not require any correctness guarantees, since it is purely speculative (i.e., it never updates the architectural state of the processor). Reusing instruction results requires correctness guarantees in the form of mechanisms that make sure that the reused values used to update the architectural state are correct. This requirement is likely to increase the design verification time of a runahead processor.

On the other hand, reuse of instruction results is also appealing in two ways. First, it has the potential to increase performance, since instructions that are valid during runahead mode do not need to be executed during normal mode and therefore they do not need to consume processor resources during normal mode. This frees up space in the processor for other instructions and possibly enables the processor to make faster progress through the instruction stream. Second, the reuse of instruction results increases the efficiency of a runahead processor. An aggressive reuse scheme reduces the total number of instructions executed by a runahead processor, thereby increasing its efficiency.[13]

To quantify the advantages of the reuse of instruction results generated during runahead mode, this section examines the effects of reuse on the performance of a runahead processor. If the reuse of results could improve the performance of a runahead processor significantly, perhaps it would be worthwhile to research techniques for implementing reuse. We examine two different models of reuse: *ideal* and *simple reuse*.

5.5.1.1 Ideal Reuse Model

We consider the hypothetical *ideal reuse* model to assess the performance potential of runahead result reuse. In this mechanism, if an instruction was valid (correctly executed) during runahead mode, it consumes no processor resources (including fetch bandwidth) during normal mode and updates the architectural state of the processor with the value it computed during runahead mode. Note that this mechanism is not implementable. We only simulate it to get an upper bound on the performance improvement that can be achieved.

In the *ideal reuse* mechanism, the instructions pseudo-retired during runahead mode write their results and INV status into a FIFO reuse queue, the size of which is sufficiently large (64K entries) in our experiments. At the end of a runahead period, the reuse queue

[13]However, such an aggressive reuse scheme may require complex control and large storage hardware whose energy consumption may offset the energy reduction gained by increased efficiency.

contains a trace of all instructions pseudo-retired in the runahead period along with their program counters, results, and INV status. During normal mode, the simulator reads instructions from this queue, starting with the first instruction pseudo-retired during runahead mode. If the processor encounters an INV instruction in the queue, it is inserted into the pipeline to be executed, since the result for an INV instruction was not generated during runahead mode. If the processor encounters a valid instruction, the instruction is not inserted into the pipeline, but the simulator makes sure that its results are correctly incorporated into the architectural state at the right time, without any latency cost. Hence, the processor is able to fetch a fetch-width worth of INV instructions from the reuse queue each cycle, regardless of the number of intervening valid instructions. If an INV branch that was fetched from the reuse queue is found out to be mispredicted after it is executed in normal mode, the reuse queue is flushed and the fetch engine is redirected to the correct next instruction after the mispredicted INV branch. An INV branch is mispredicted if the program counter of the next instruction fetched from the reuse queue does not match the address of next instruction as calculated by the execution of the branch. When the reuse queue is empty, the simulator fetches from the I-cache just like in a normal processor.[14]

As an optimization, if the processor enters runahead mode again while fetching from the reuse queue, the processor continues fetching from the reuse queue. Therefore, valid instructions that are executed in the previous runahead period do not need to be executed again in the next entry into runahead mode. A runahead period overlapping with a previous runahead period can thus reuse the instruction results generated in the previous period. This allows for further progress in the instruction-stream during runahead mode. We use this optimization in both the *ideal* and *simple reuse* models.

[14]Note that this mechanism does not allow the reuse of results generated by instructions executed on the wrong path during runahead mode. Section 5.5.1.5 examines the impact of *ideal reuse* on a processor with perfect branch prediction, which gives the ultimate upper bound on the performance achievable by reusing the results of *all* valid instructions pre-executed in runahead mode.

5.5.1.2 Simple Reuse Model

We consider the *simple reuse* model to assess the performance potential of a more realistic reuse mechanism that is similar to the previously proposed dynamic instruction reuse technique [104]. Similar to the *ideal reuse* model, the *simple reuse* model makes use of a reuse queue, which is populated by pseudo-retired runahead mode instructions. However, in the *simple reuse* model, a valid instruction that is fetched from the reuse queue actually consumes resources, including fetch bandwidth. It is inserted into the pipeline as a move instruction that moves an immediate value (the correct result produced in runahead mode) into its destination (register or memory location) in a single cycle. Hence, a valid instruction fetched from the reuse queue is independent of any other instruction and can be executed immediately after it is placed into the scheduler. This allows for the parallel execution of valid instructions during normal mode, which can result in performance improvement. However, the *simple reuse* mechanism is unlikely to reduce the number of executed instructions because reused instructions are still executed as move operations.

5.5.1.3 Performance and Efficiency Potential of Result Reuse

Figure 5.30 shows the IPC increase of the *simple* and *ideal reuse* models over the baseline processor, along with the IPC increase of runahead execution. On average, *simple reuse* increases the baseline IPC by 23.1% and *ideal reuse* by 24.7%. Hence, adding *simple reuse* to a runahead execution processor improves the average IPC by 0.38% and adding *ideal reuse* improves the average IPC by 1.64%. Unfortunately, such a small performance improvement, even with an ideal mechanism, probably does not justify the cost and complexity of adding a reuse mechanism to a runahead execution processor. We examine the reasons for the poor performance improvement of the ideal reuse mechanism in Section 5.5.1.4.

Figure 5.31 shows increase in executed instructions over the baseline processor for

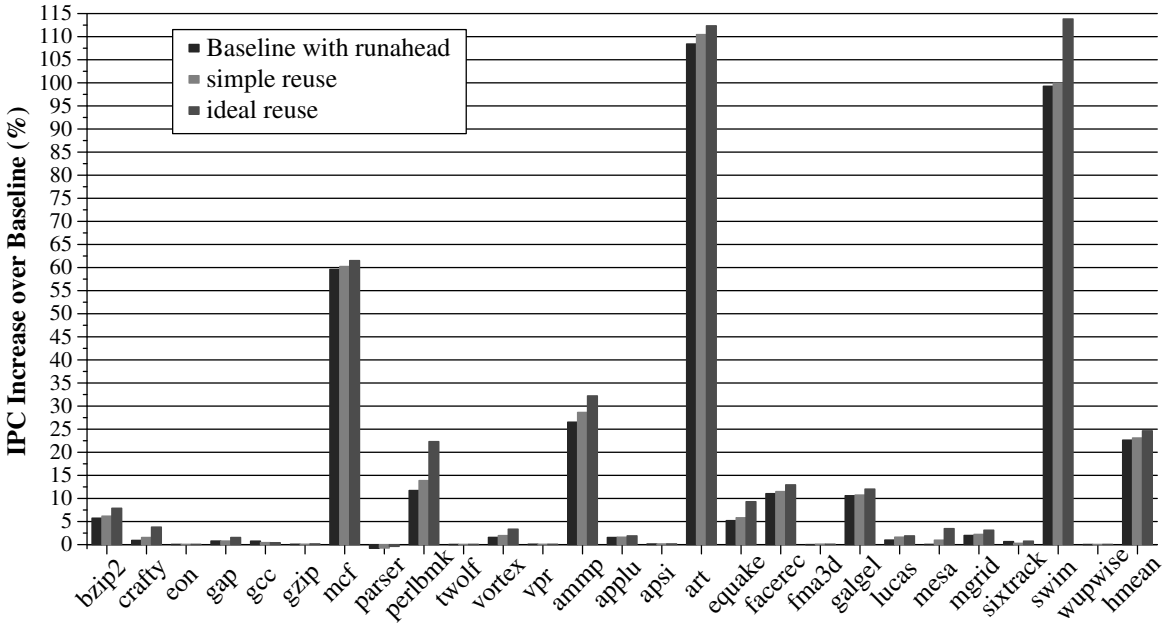


Figure 5.30: Increase in IPC with the simple and ideal reuse mechanisms.

the *simple* and *ideal reuse* models. *Simple reuse* actually increases the number of executed instructions because it does not eliminate the re-execution of reused instructions. The conversion of reused instructions to short-latency move operations enables the fast removal of reused instructions from the instruction window in runahead mode, which results in the execution of more instructions than a runahead processor without simple reuse. *Ideal reuse* reduces the increase in number of executed instructions from 26.5% to 16.6%, leading to a more efficient runahead execution processor. However, compared to the reduction in executed instructions obtained using the techniques proposed in Sections 5.2 and 5.3, this reduction is small. Therefore, employing result reuse as an efficiency-increasing technique is perhaps too complex and costly when there are simpler techniques that achieve better efficiency.

We also evaluated combining *ideal reuse* with the efficiency-increasing techniques proposed in Sections 5.2 and 5.3. Combining the dynamic efficiency techniques evaluated

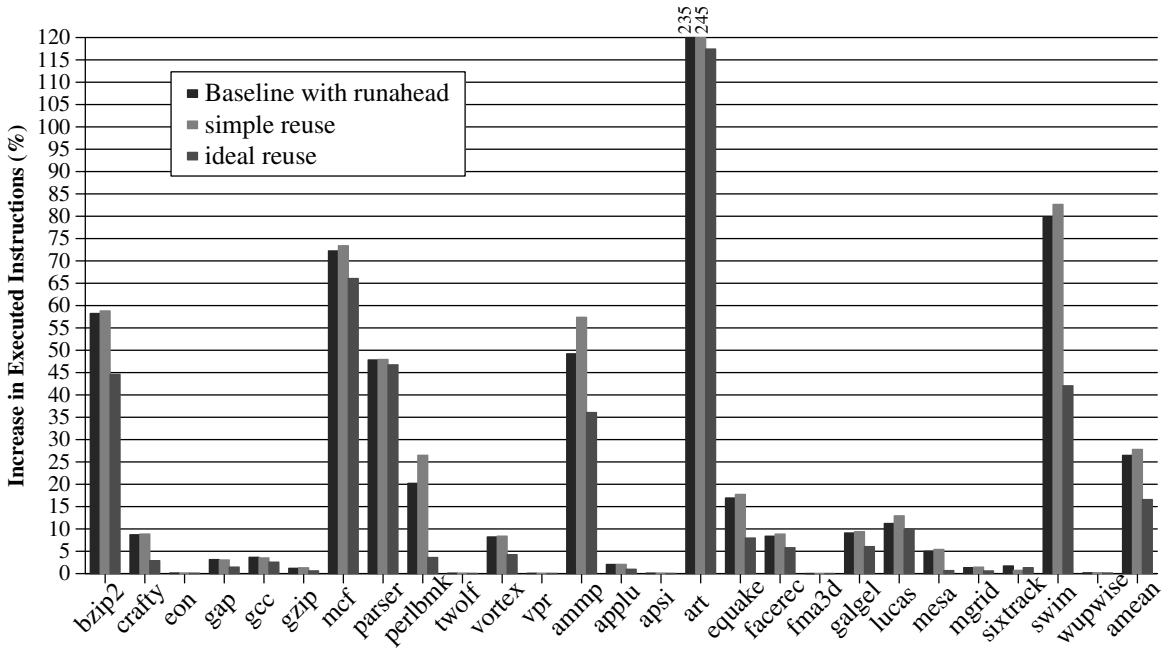


Figure 5.31: Effect of the simple and ideal reuse mechanisms on the percent increase in executed instructions.

in Section 5.4 with *ideal reuse* results in a decrease in the number of executed instructions compared to what is achieved solely by using the proposed dynamic efficiency techniques, from 6.7% (see Section 5.4) to 3.8%. This also results in an increase in the IPC improvement from 22.0% to 23.0%. Even when combined with other efficiency techniques, ideal reuse does not provide benefits significant enough to justify a costly implementation. Therefore, we conclude that runahead execution should be employed as a prefetching mechanism without reuse.

5.5.1.4 Why Does Reuse Not Increase Performance Significantly?

We analyzed why ideal result reuse does not increase the IPC of a runahead execution processor significantly. We identify four reasons as to why this is the case.

First, the percentage of retired instructions that are ideally reused is very low in many benchmarks. Result reuse can only be effective if it significantly reduces the number of retired instructions by eliminating those that are executed in runahead mode. Figure 5.32 shows the breakdown of retired instructions based on where they were fetched from. An instruction that is "Reuse queue fetch - ideally reused" does not consume processor resources in normal mode. All other instructions need to be executed as described in Section 5.5.1.1. Figure 5.32 shows that only 8.5% of the instructions are eliminated (ideally reused) during normal mode.[15] The percentage of ideally reused instructions is low because: (1) For many benchmarks, the processor spends a small amount of time during runahead mode, which means only a small number of instructions are pre-executed in runahead mode. Hence, there are not a lot of opportunities for reuse. (2) The results of instructions after a mispredicted INV branch cannot be reused. This limits the number of reused results especially in integer benchmarks, where 43% of all instructions pseudo-retired during runahead mode are after a mispredicted INV branch. Unfortunately, even in FP benchmarks, where only 5% of instructions pseudo-retired during runahead mode are after a mispredicted INV branch, ideal reuse does not result in a large IPC increase due to the reasons explained in the following paragraphs.

Second, and more importantly, eliminating an instruction does not guarantee increased performance because eliminating the instruction may not reduce the critical path of program execution. The processor still needs to execute the instructions that cannot be eliminated because they were INV in runahead execution. After eliminating the valid instructions, instructions that are INV become the critical path during normal mode execution.[16] If this INV critical path is longer than or as long as the critical path of the eliminated

[15]Only in perlbmk, ammp, art, equake, lucas, and swim does the percentage of eliminated instructions exceed 10%. With the exception of lucas, these are the benchmarks that see the largest IPC improvements over the runahead processor with the *ideal reuse* mechanism.

[16]In some cases they were already on the critical path, regardless of the elimination of valid instructions.

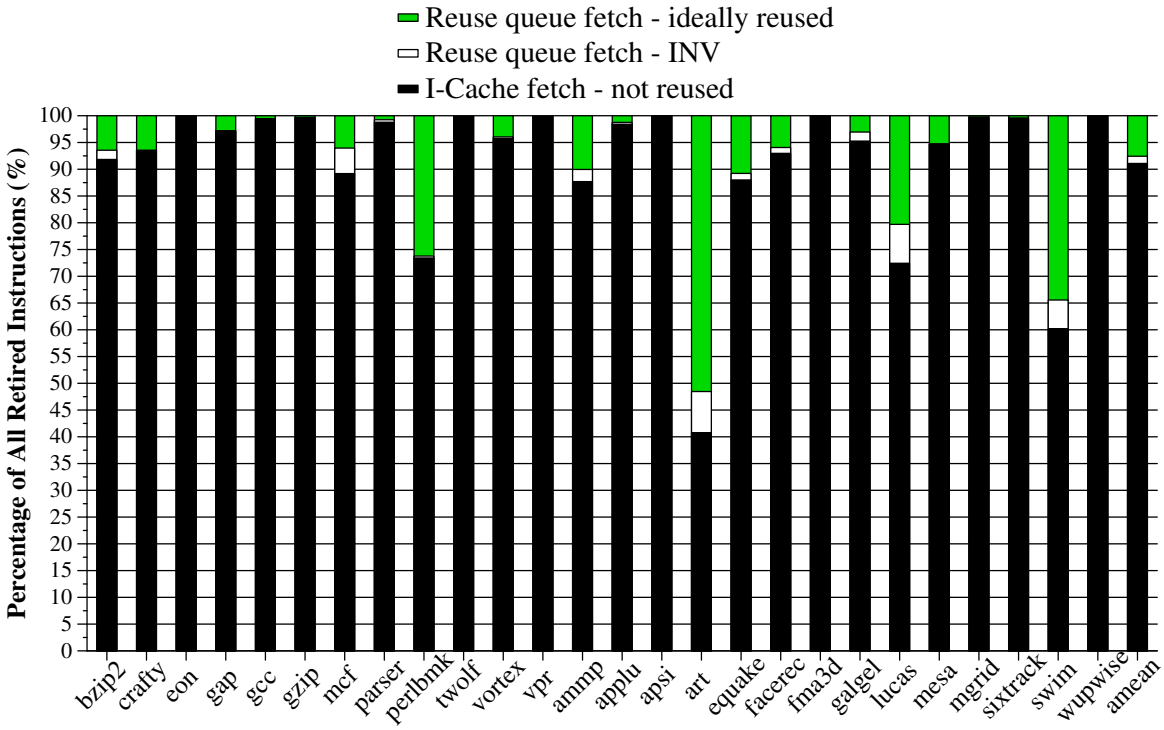


Figure 5.32: Classification of retired instructions in the ideal reuse model.

valid instructions, we should not expect much performance gain unless the processor is limited by execution bandwidth. Our analysis of the code shows that the execution of valid instructions after runahead mode is actually much faster than the execution of INV instructions, in general. Instructions that were valid in runahead mode can be executed much faster in normal mode because they do not incur any L1 or L2 cache misses in normal mode.[17] In contrast, instructions that were INV can and do incur L1 and L2 misses during normal mode, making them more expensive to execute. We found that the average execution time of a valid instruction is only 2.48 cycles after runahead mode, whereas the average execution time of an INV instruction is 14.58 cycles. Thus, ideal reuse eliminates valid

[17]Except in the rare case when the runahead period was so long that it caused thrashing in the L1 cache.

instructions that do not take long to execute. Therefore, it cannot reduce the critical path of the program significantly and results in small performance improvements.[18] Table 5.1 provides supporting data showing that although most of the instructions pre-executed in runahead mode are valid (row 6), an important percentage of INV instructions incur very long latency L2 misses and relatively long latency L1 misses in normal mode (rows 7-8) and therefore take much longer to execute than valid instructions (rows 12-13).

Row	Statistic	INT	FP	Average
1	Percent cycles in runahead mode	13.7%	27.1%	21.2%
2	Cycles per runahead (RA) period	474	436	454
3	L2 misses captured per RA period	1.45	3.4	2.5
4	Pseudo-retired instructions per RA period	1240	1169	1201
5	Correct-path instructions per RA period	712 (57%)	1105 (95%)	923 (77%)
6	INV instructions per RA period	544 (44%)	264 (23%)	393 (33%)
7	% of INV/valid inst. that are L2 miss	1.4%/0%	2.1%/0%	1.9%/0%
8	% of INV/valid inst. that are D-cache miss	3.0%/0%	2.8%/0%	2.9%/0%
9	% of INV/valid inst. that are D-cache hit	38%/38%	36%/44%	36%/43%
10	% of INV/valid inst. that are FP	0.8%/0.4%	60%/25%	43%/20%
11	% of INV/valid inst. that are 1-cycle	59%/61%	2%/31%	18%/37%
12	Avg. latency of valid instructions (after RA)	1.78 cycles	2.66 cycles	2.48 cycles
13	Avg. latency of INV instructions (after RA)	10.16 cycles	16.31 cycles	14.58 cycles
14	IPC improvement of ideal reuse	0.90%	2.45%	1.64%

Table 5.1: Runahead execution statistics related to ideal reuse.

Figure 5.33 shows an example data-flow graph from the mcf benchmark demonstrating why reuse may not increase performance. In this example, shaded instructions were INV during runahead mode. Numbers by the data-flow arcs show the latency of the instructions. After exit from runahead mode, the second INV load takes 13 cycles because it incurs an L1 data cache miss. Therefore, the INV dependence chain takes 17 cycles to

[18]Note that ideal reuse is more effective in FP benchmarks than in INT benchmarks because it eliminates some FP dependence chains that take a long time to execute due to the relatively long-latency FP instructions.

execute. If there is no result reuse, valid instructions take only 6 cycles to execute because there are enough execution resources in the processor core. Hence, eliminating these valid instructions via ideal reuse does not affect the critical path and does not increase performance, even though the number of valid instructions is larger than the number of INV ones.

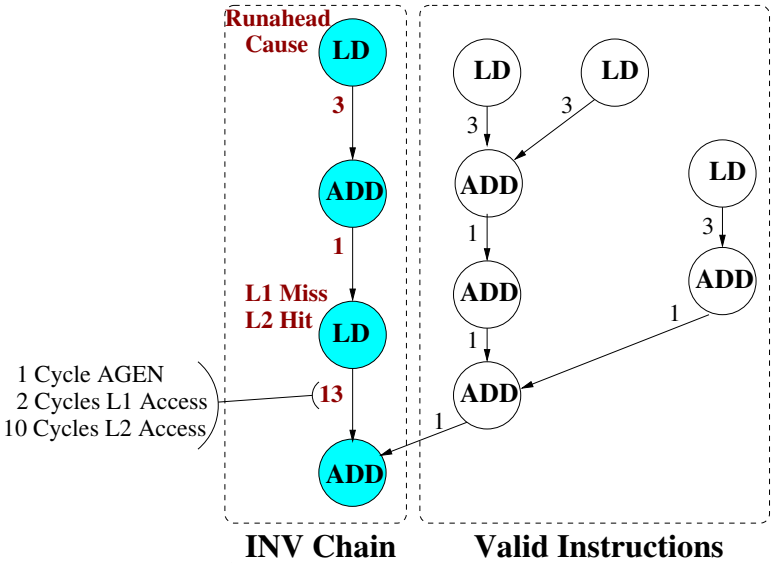


Figure 5.33: Example showing why reuse does not increase performance.

Third, if the execution time of a benchmark is dominated by long-latency memory accesses (memory-bound), the performance potential of result reuse is not very high since reuse can only speed up the smaller execution-bound portion. That is, the ideal reuse of valid instructions cannot reduce the number of cycles spent on long-latency memory accesses. Note that this is a special case of Amdahl's Law [4]. If the optimization we apply (reuse) speeds up only a small fraction of the program (execution-bound portion of the program), the performance gain due to that optimization will be small. Unfortunately, memory-bound benchmarks are also the benchmarks that execute many instructions in runahead mode. Art is a very good example of a memory-bound benchmark. We find

that, in art, 86% of all cycles are spent waiting for long-latency L2 load misses to be serviced. Hence, even if all other execution cycles are magically eliminated, only a 16% IPC improvement can be achieved. This constitutes a loose upper bound on the IPC improvement that can be attained by the *ideal reuse* scheme. In our simulations, the *ideal reuse* scheme only increases the IPC of art on the runahead processor by 3.3%, even though it eliminates 51% of all retired instructions.

Fourth, a processor that implements ideal reuse enters some performance-degrading short runahead periods that do not exist in the baseline runahead processor. The *ideal reuse* processor reaches an L2-miss load faster than the baseline runahead processor because of the elimination of valid instructions in normal mode. If there is a prefetch pending for this load, the ideal reuse processor provides less time than the baseline runahead processor for the prefetch to complete before the load is executed (since it reaches the load quicker), which makes the ideal reuse processor more likely to enter runahead mode for the load than the baseline runahead processor. This runahead period is usually very short and results in a pipeline flush without any performance benefit. Such periods, which do not exist in the baseline runahead processor, reduce the performance benefit due to ideal reuse.

5.5.1.5 Effect of Main Memory Latency and Branch Prediction Accuracy

Memory latency is an important factor on the performance impact of result reuse, since it affects the number of instructions executed during runahead mode (hence, the number of instructions that can possibly be reused) and it affects how much of a program's total execution time is spent on servicing L2 cache misses. Branch prediction accuracy is also an important factor because it affects the number of *useful* instructions that can be reused (by affecting the number of mispredicted INV branches during runahead mode or by affecting when they occur during runahead mode).

The left graph in Figure 5.34 shows the average IPC improvement of runahead exe-

cution and runahead with ideal reuse over the respective baseline processor for four different memory latencies. As the memory latency becomes shorter, the processor spends less time and executes fewer instructions during runahead mode. This reduces the percentage of retired instructions that are ideally reused in a program. For all four memory latencies, the IPC improvement of ideal reuse is not very promising.

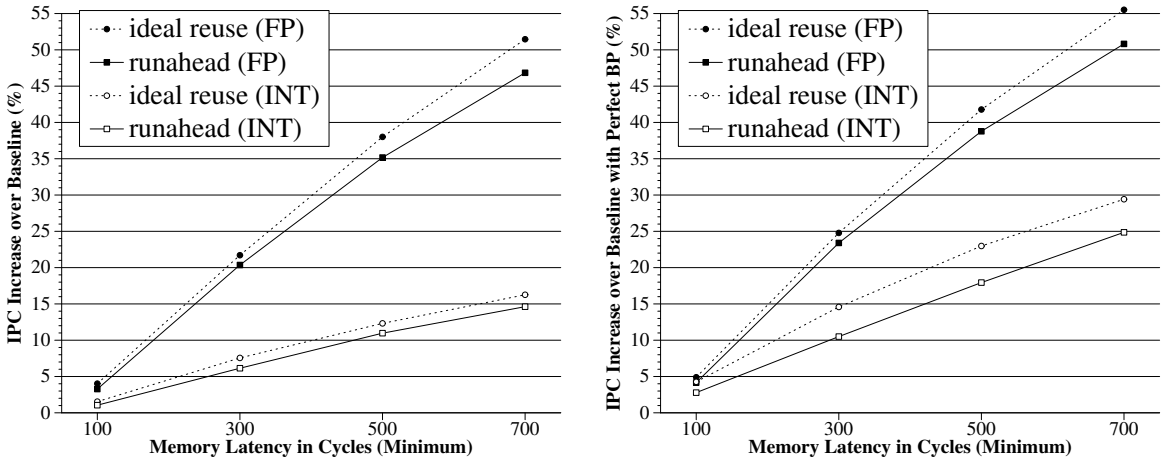


Figure 5.34: Effect of memory latency and branch prediction on reuse performance.

The right graph in Figure 5.34 shows the average IPC improvement of runahead execution and runahead with ideal reuse over the respective baseline processor with perfect branch prediction for the same four memory latencies. Ideal reuse on a processor with perfect branch prediction shows more performance potential for INT benchmarks than ideal reuse on a processor with realistic branch prediction. On INT benchmarks, with perfect branch prediction, runahead execution by itself improves the IPC of the baseline processor with 500-cycle main memory latency by 18%, whereas ideal reuse improves the IPC of the same processor by 23%. Therefore, even if an ideal reuse mechanism is implemented in a runahead processor with perfect branch prediction, the prefetching benefit of runahead execution would provide most of the performance improvement obtained over the baseline processor. The additional performance improvement due to ideal reuse is still perhaps

not large enough to warrant the implementation of a realistic reuse mechanism, even with perfect branch prediction.

5.5.2 Value Prediction of L2-miss Load Instructions in Runahead Mode

The baseline runahead execution implementation marks the results of L2-miss load instructions as INV during runahead mode. An alternative option is to predict the results of such instructions. There are two major advantages to this if the value of the instruction is correctly predicted. First, mispredicted branch instructions dependent on the result can correctly compute their outcomes and can initiate misprediction recovery. Second, load and store instructions whose addresses depend on the result can correctly compute their addresses and possibly generate useful cache misses in runahead mode. Therefore, correct prediction of the result values of L2-miss instructions can improve performance. Furthermore, the discovery of new L2 misses with value prediction can eliminate runahead periods that would otherwise be caused by those misses, thereby eliminating extra instructions that would otherwise be executed.

On the other hand, if the result of an L2-miss instruction is mispredicted, the performance of a runahead processor can be degraded. Correctly predicted branches dependent on the result can be overturned and they can result in inaccurate misprediction recoveries. Load and stores dependent on the incorrect value can generate inaccurate memory requests, possibly causing bandwidth contention and cache pollution.

Even if the value of an L2-miss is correctly predicted, performance may not be improved because of two reasons. First, the predicted value may not lead to the resolution of any mispredicted branches or generation of cache misses. In such cases, the performance of a runahead processor can actually be degraded because the prediction of the value causes slower execution of the L2-miss dependent instructions as also noted by Kirman et al. [59]. For example, if the value of an L2-miss FP load is predicted, the dependent FP operate

instructions are executed. Their execution is slower than the propagation of the INV status among those instructions. This results in slower progress in runahead mode and can reduce the number of L2 misses discovered. Second, perfect value prediction results in the resolution of mispredicted L2-miss dependent branches. This causes the runahead processor to flush its pipeline and return to the correct path instead of continuing to execute on the wrong path. If wrong-path instructions provide useful prefetching benefits and the processor is already at a control-flow independent point on the wrong path, then recovering to the correct path may actually degrade performance compared to staying on the wrong path and making further progress to discover wrong-path L2 cache misses. We found that this effect is very significant in the mcf benchmark where wrong-path instructions generate very useful L2 cache misses [77].[19]

We evaluated the effect of using value prediction for L2-miss instructions on the performance and efficiency of runahead execution using different value predictors for L2-miss instructions. Four value predictors were modeled: 1) *zero VP* always predicts the value as 0, which requires no extra storage, 2) *last VP* [66] predicts the value of the current instance of the instruction to be the same as the value in its previous instance (if the same value was produced by the instruction consecutively at least in the last two dynamic instances of the instruction), 3) *stride VP* [97] predicts the value of the instruction if the values produced by the instruction have shown a striding pattern, and 4) *perfect VP* always predicts the value correctly using oracle information. Figures 5.35 and 5.36 show the increase in executed instructions and IPC using these predictors.

A perfect value predictor improves the performance benefit of runahead execution from 22.6% to 24% while decreasing the extra instructions from 26.5% to 22.1%. Realistic 4K-entry stride and last value predictors provide even smaller benefits. In some cases, even

[19]For this very reason, mcf loses performance when the values of all L2-miss instructions in runahead mode are correctly predicted (See Figure 5.36).

a perfect value predictor degrades performance because of the reasons described above. This suggests that using value prediction has limited potential to improve runahead performance and efficiency on the benchmark set we examined. One of the reasons for this is the lack of L2-miss dependent cache misses and L2-miss dependent branches in the examined benchmarks (except for parser and bzip2 which see large IPC improvements with perfect value prediction). In Chapter 6, we focus on a pointer-intensive benchmark set and show that a new value prediction mechanism designed for predicting pointer values in runahead mode can significantly improve both performance and efficiency.

We also note that the performance and efficiency of a runahead processor using the *zero VP* is very similar to those of the baseline runahead processor, which uses INV bits. Using the *zero VP* slightly reduces the performance improvement of runahead execution from 22.6% to 21.9% but it also reduces the extra instructions from 26.5% to 25.1%. Since predicting the values of L2 miss load instructions as 0 does not require any hardware storage or any INV bits, a runahead processor using a zero value predictor for L2-miss load instructions is simpler than the baseline runahead implementation and therefore should be considered as a viable option by the designers of a runahead processor.

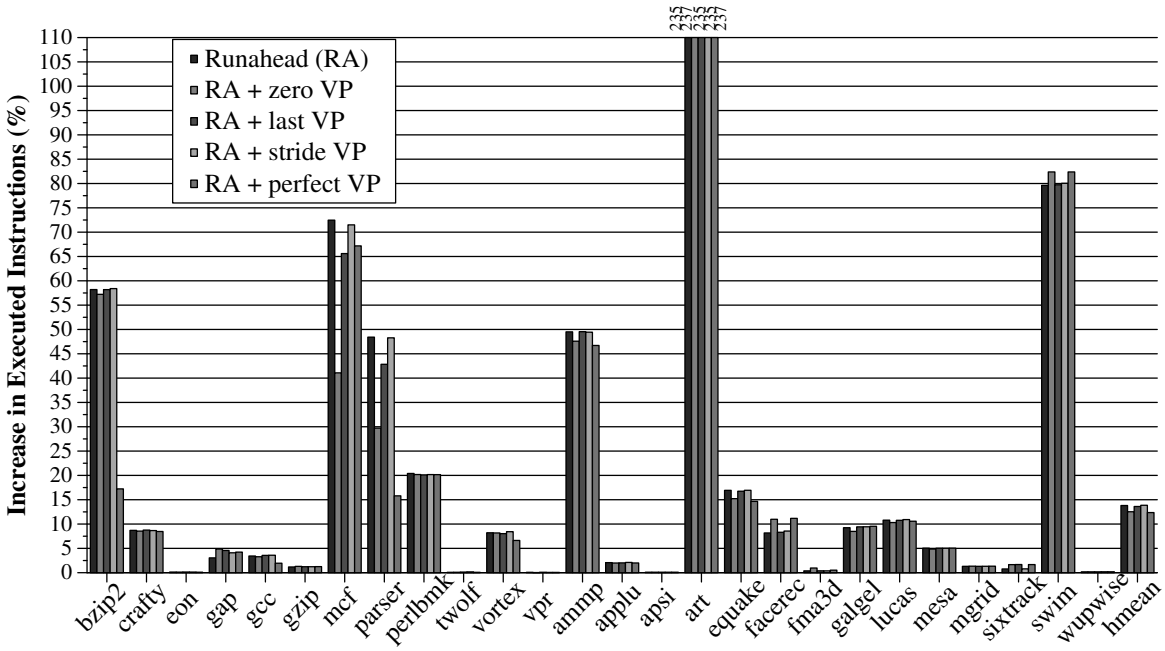


Figure 5.35: Increase in executed instructions with different value prediction mechanisms for L2-miss instructions.

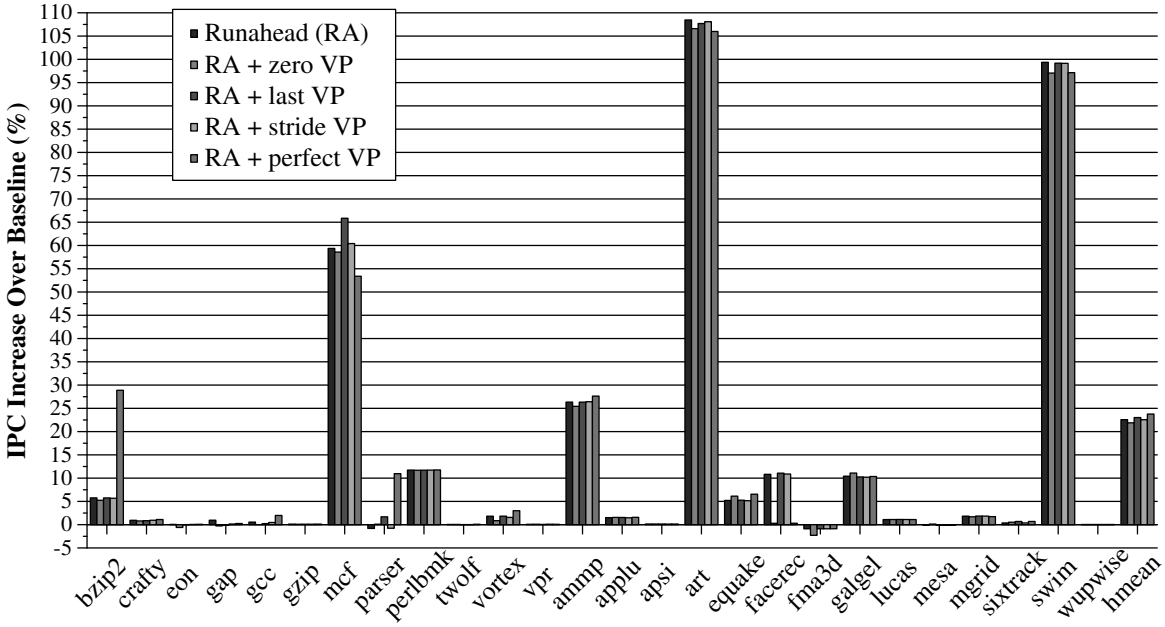


Figure 5.36: Increase in IPC with different value prediction mechanisms for L2-miss instructions.

5.5.3 Optimizing the Exit Policy from Runahead Mode

The baseline runahead implementation exits runahead mode when the runahead-causing L2 cache miss is serviced. This exit policy is not optimal for performance or efficiency. If the processor stays in runahead mode a number of cycles after the runahead-causing L2 miss is serviced, it may discover an L2 cache miss down in the instruction stream. The performance benefit of initiating that L2 cache miss may outweigh the performance cost of staying in runahead mode at the expense of normal mode execution. Therefore, extending runahead mode execution after the return of the L2 miss can yield performance benefits. Moreover, extending runahead mode might also reduce the number of instructions executed in a runahead processor by reducing the number of entries into runahead mode and eliminating short runahead periods that would otherwise be entered if a previous period were not extended.

On the other hand, if no L2 cache misses are discovered during the extended portion of the runahead period, performance will degrade because the processor stays in runahead mode without any benefit instead of doing useful work in normal mode. Also, if the extension of a runahead period does not eliminate other runahead periods, the number of executed instructions will increase. Therefore, the decision of whether or not to extend a runahead period should be made very carefully.

We developed a simple mechanism that predicts whether or not extending the ongoing runahead period would be beneficial for performance based on the usefulness of the ongoing runahead period. This mechanism counts the number of L2 cache misses generated in a runahead period. If this number is greater than a threshold M when the runahead-causing L2 miss is serviced, the processor stays in runahead mode for C more cycles. Otherwise, the processor exits runahead mode. If the runahead period is extended, the processor counts the L2 misses generated during the extra C cycles in runahead mode. At the end of C cycles, the processor again checks if the number of L2 misses generated

during C cycles is greater than M. If so, runahead mode is extended for C more cycles. This process is repeated until either the number of generated L2 misses is less than M or the ongoing runahead period is already extended E times (E was fixed at 10 in our experiments). We impose a limit on the number of extensions to prevent the starvation of the running program and ensure its forward progress.

Figures 5.37 and 5.38 show the increase in executed instructions and IPC when this mechanism is used with varying values for M and C. Extending the runahead periods using this simple prediction mechanism increases the performance improvement of runahead execution from 22.6% to 24.4% while also reducing the extra instructions from 26.5% to 21.1% (with M=3 and C=500). Extending the runahead periods is especially effective in improving the IPC of art and swim. These two benchmarks have very large numbers of L2 misses clustered close together in program execution. Staying in runahead mode longer enables the runahead processor to initiate many of the misses that are clustered without significant performance cost because normal mode execution would not have made significant progress due to the occurrence of large number of L2 misses in clusters.

We conclude that predictive mechanisms for extending runahead mode have potential to improve both the performance and efficiency of runahead execution. The mechanism described here is a simple and implementable technique that provides some benefits through the use of simple heuristics. It may be possible to design even more effective techniques by developing an aggressive cost-benefit analysis scheme in hardware. However, determining the exact or even approximate costs and benefits of events that happen in the future may prove to be a difficult task.

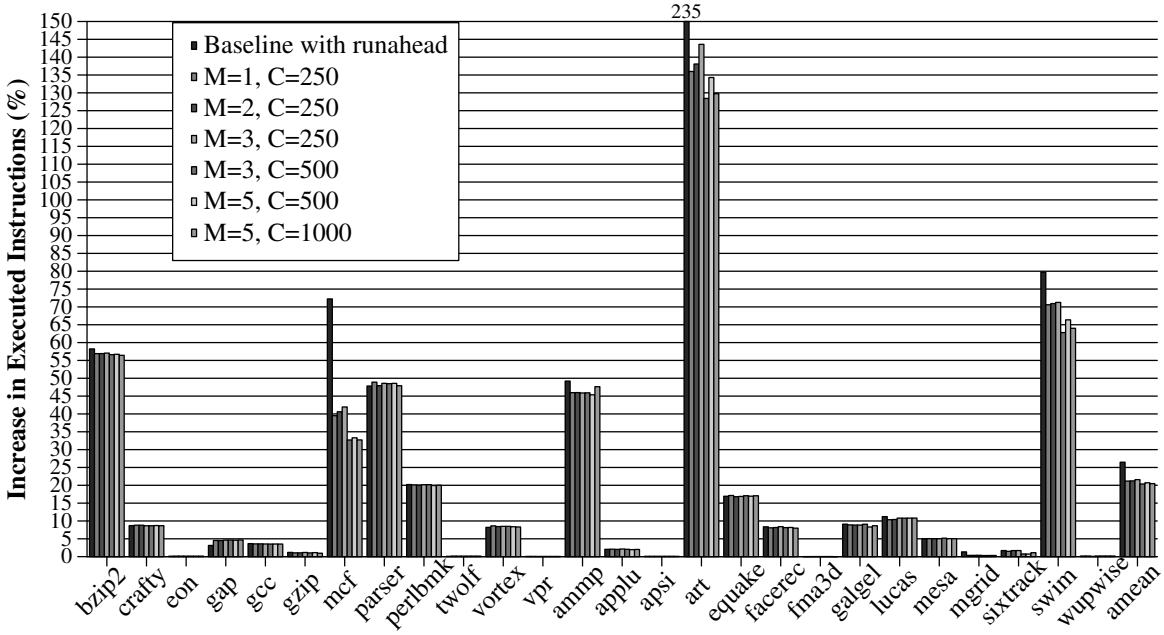


Figure 5.37: Increase in executed instructions after extending the runahead periods.

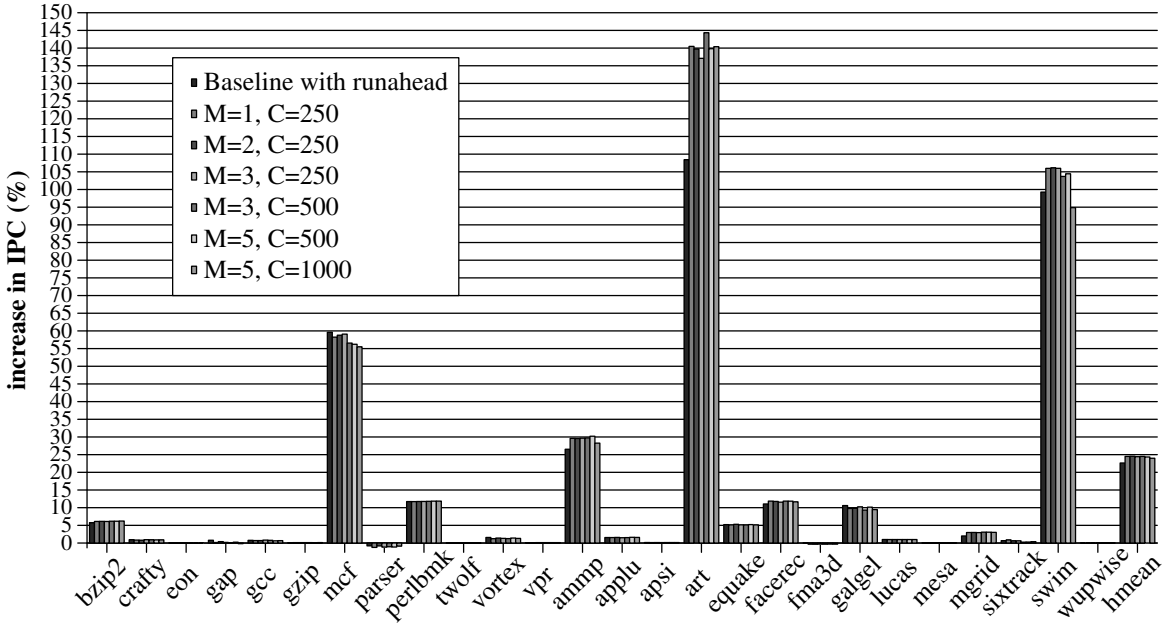


Figure 5.38: Increase in IPC after extending the runahead periods.

5.6 Summary and Conclusions

Runahead execution can significantly increase energy consumption because it increases the number of speculatively executed instructions. This chapter examined the efficiency of runahead execution and described new and simple techniques to make a runahead processor more efficient.

Three major causes of inefficiency were identified: *short*, *overlapping*, and *useless* runahead periods. Techniques that reduce the occurrence of these causes significantly reduce the extra instructions due to runahead execution without significantly affecting performance. Techniques that increase the usefulness of runahead periods by increasing the number of useful L2 misses generated in runahead mode increase the performance of runahead execution without significantly increasing and sometimes decreasing the executed instructions. Incorporating all the proposed efficiency techniques in a runahead processor reduces the extra instructions from 26.5% to 6.2% while only slightly reducing runahead execution's IPC improvement from 22.6% to 22.1%.

Reuse of runahead instruction results, value prediction of runahead L2-miss instructions, and extension of runahead periods were examined as possible techniques to improve performance and efficiency. Augmenting runahead execution with aggressive reuse and value prediction mechanisms does not significantly improve performance or efficiency while it likely adds significant hardware cost and complexity. Predicting the values of L2-miss load instructions as zero instead of marking them as INV eliminates the INV bits needed by the baseline runahead execution without significantly degrading performance and therefore can reduce the complexity of a runahead processor. Finally, extending runahead periods with a simple prediction mechanism has potential to improve both the performance and efficiency of runahead execution.

Chapter 6

Address-Value Delta (AVD) Prediction

The proposed runahead execution mechanism cannot parallelize *dependent* long-latency cache misses. A runahead processor cannot execute instructions that are *dependent on the pending long-latency cache misses* during runahead mode, since the data values they are dependent on are not available. These instructions are designated as INV and they mark their destination registers as INV so that the registers they produce are not used by instructions dependent on them. Hence, runahead execution is not able to parallelize two long-latency cache misses if the load instruction generating the second miss is dependent on the load instruction that generated the first miss.[1] These two misses need to be serviced serially. Therefore, the full-latency of each miss is exposed and the latency tolerance of the processor cannot be improved by runahead execution. Applications and program segments that heavily utilize linked data structures (where many load instructions are dependent on previous loads) therefore cannot significantly benefit from runahead execution. In fact, for some pointer-chasing applications, runahead execution reduces performance due to its overheads and significantly increases energy consumption due to the increased activity caused by the runahead mode pre-processing of useless instructions.

In order to overcome the serialization of dependent long-latency cache misses, techniques to parallelize dependent load instructions are needed. These techniques need to focus on predicting the values loaded by *address (pointer) loads*, i.e. load instructions that

[1] Two dependent load misses cannot be serviced in parallel in a conventional out-of-order processor either.

load an address that is later dereferenced. Previous researchers have proposed several dynamic techniques to predict the values of address loads [66, 97, 11, 26] or to prefetch the addresses generated by them [94, 95, 26]. Unfortunately, to be effective, these techniques require a large amount of storage and complex hardware control. As energy/power consumption becomes more pressing with each processor generation, simple techniques that require small storage cost become desirable and necessary. The purpose of this chapter is to devise a technique that reduces the serialization of dependent long-latency misses *without significantly increasing the hardware cost and complexity*.

This chapter describes a new, simple, implementable mechanism, *address-value delta (AVD) prediction*, that allows the parallelization of dependent long-latency cache misses in runahead mode. The proposed technique learns the *arithmetic difference (delta)* between the effective address and the data value of an *address load* instruction based on the previous executions of that load instruction. Stable *address-value deltas* are stored in a prediction buffer. When a load instruction incurs a long-latency cache miss in runahead mode, if it has a stable *address-value delta* in the prediction buffer, its data value is predicted by subtracting the stored delta from its effective address. This predicted value enables the pre-execution of dependent instructions, including load instructions that incur long-latency cache misses. We provide source-code examples showing the common code structures that cause stable *address-value deltas*, describe the implementation of a simple *address-value delta* predictor, and evaluate its performance benefits on a runahead execution processor. We show that augmenting a runahead processor with a simple, 16-entry (102-byte) AVD predictor improves the execution time of a set of pointer-intensive applications by 14.3% *and* reduces the number of executed instructions by 15.5%.

6.1 Motivation for Parallelizing Dependent Cache Misses

The goal of this chapter is to increase the effectiveness of runahead execution with a simple prediction mechanism that overcomes the inability to parallelize dependent long-latency cache misses during runahead mode. We demonstrate that focusing on this limitation of runahead execution has potential to improve processor performance. Figure 6.1 shows the potential performance improvement possible if runahead execution were able to parallelize all the *dependent long-latency cache misses* that can be generated by instructions that are pre-processed during runahead mode. This graph shows the execution time for four processors on memory- and pointer-intensive benchmarks from Olden and SPEC INT 2000 benchmark suites:[2] from left to right, (1) a processor with no runahead execution, (2) the baseline processor, which employs runahead execution, (3) an ideal runahead processor that can parallelize dependent L2 cache misses (This processor is simulated by obtaining the correct effective address of *all* L2-miss load instructions using oracle information during runahead mode. Thus, L2 misses dependent on previous L2 misses can be generated during runahead mode using oracle information. This processor is not implementable, but it is intended to demonstrate the performance potential of parallelizing dependent L2 cache misses.), (4) a processor with perfect L2 cache. Execution times are normalized to the baseline processor. The baseline runahead processor improves the average execution time of the processor with no runahead execution by 27%. The ideal runahead processor improves the average execution time of the baseline runahead processor by 25%, showing that significant performance potential exists for techniques that enable the parallelization of dependent L2 misses. Table 6.1, which shows the average number of L2 cache misses initiated during runahead mode, provides insight into the performance improvement possible with the ideal runahead processor. This table shows that the ideal runahead processor

[2] Section 6.5 describes the benchmarks used in this chapter.

significantly increases the memory-level parallelism (the number of useful L2 cache misses parallelized[3]) in a runahead period.

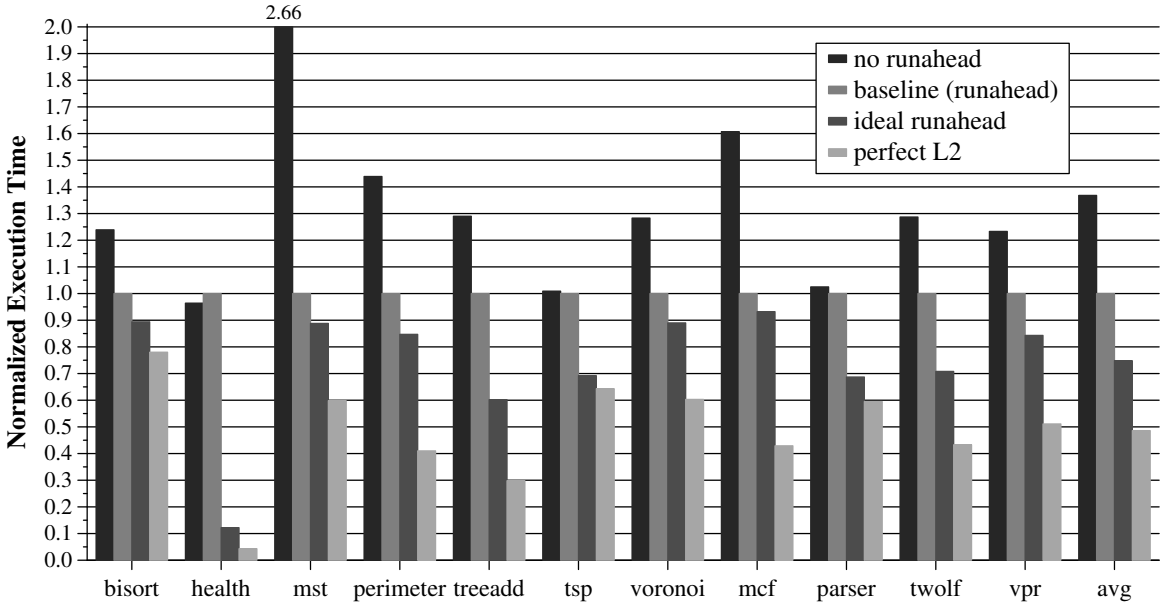


Figure 6.1: Performance potential of parallelizing dependent L2 cache misses in a runahead execution processor.

	bisort	health	mst	perimeter	treeadd	tsp	voronoi	mcf	parser	twolf	vpr	avg
baseline runahead	2.01	0.03	7.93	1.45	1.02	0.19	0.81	11.51	0.12	0.84	0.94	2.44
ideal runahead	4.58	8.43	8.77	2.06	2.87	4.42	1.43	12.75	1.56	2.79	1.19	4.62

Table 6.1: Average number of useful L2 cache misses generated (parallelized) during a runahead period. Only L2 line (block) misses that cannot already be generated by the processor's fixed-size instruction window are counted.

Figure 6.1 also shows that for two benchmarks (`health` and `tsp`) runahead execution is ineffective. These two benchmarks have particularly low levels of memory-level

[3]A useful L2 cache miss is an L2 cache miss generated during runahead mode that is later needed by a correct-path instruction in normal mode. Only L2 line (block) misses that are needed by load instructions and that cannot already be generated by the processor's fixed-size instruction window are counted.

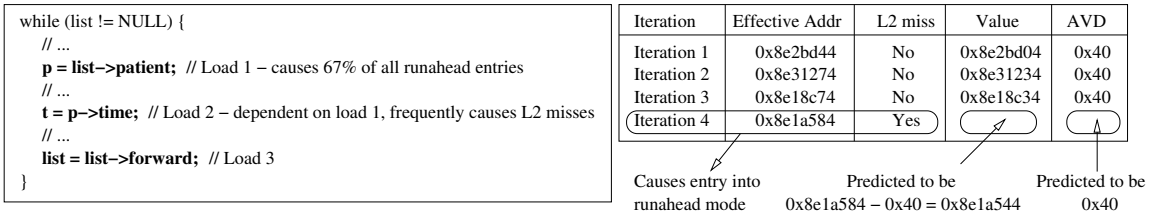
parallelism, since their core algorithms consist of traversals of linked data structures in which almost all load instructions are dependent on previous load instructions. Due to the scarcity of independent long-latency cache misses (as shown in Table 6.1), conventional runahead execution cannot significantly improve the performance of `health` and `tsp`. In fact, the overhead of runahead execution results in 4% performance loss on `health`. In contrast, the ideal runahead processor provides significant performance improvement on these two benchmarks (88% on `health` and 32% on `tsp`), alleviating the ineffectiveness of conventional runahead execution.

6.2 AVD Prediction: The Basic Idea

We have observed that some load instructions exhibit stable relationships between their effective addresses and the data values they load. We call this stable relationship the *address-value deltas (AVDs)*. We define the *address-value delta* of a dynamic instance of a load instruction L as:

$$AVD(L) = Effective\ Address\ of\ L - Data\ Value\ of\ L$$

Figure 6.2 shows an example load instruction that has a stable AVD and how we can utilize AVD prediction to predict the value of that load in order to enable the execution of a dependent load instruction. The code example in this figure is taken from the `health` benchmark. Load 1 frequently misses in the L2 cache and causes the processor to enter runahead mode. When Load 1 initiates entry into runahead mode in a conventional runahead execution processor, it marks its destination register as INV. Load 2, which is dependent on Load 1, therefore cannot be executed during runahead mode. Unfortunately, Load 2 is also an important load that frequently misses in the L2 cache. If it were possible to correctly predict the value of Load 1, Load 2 could be executed and the L2 miss it causes would be serviced in parallel with the L2 miss caused by Load 1.



(a) Code example

(b) Execution history of Load 1

Figure 6.2: Source code example showing a load instruction with a stable AVD (Load 1) and its execution history.

Figure 6.2b shows how the value of Load 1 can be accurately predicted using an AVD predictor. In the first three executions of Load 1, the processor calculates the AVD of the instruction. The AVD of Load 1 turns out to be stable and it is recorded in the AVD predictor. In the fourth execution, Load 1 misses in the L2 cache and causes entry into runahead mode. Instead of marking the destination register of Load 1 as INV, the processor accesses the AVD predictor with the program counter of Load 1. The predictor returns the stable AVD corresponding to Load 1. The value of Load 1 is predicted by subtracting the AVD returned by the predictor from the effective address of Load 1 such that:

$$Predicted\,Value = Effective\,Address - Predicted\,AVD$$

The predicted value is written into the destination register of Load 1. The dependent instruction, Load 2, reads this value and is able to calculate its address. Load 2 accesses the cache hierarchy with its calculated address and it may generate an L2 cache miss that would be serviced in parallel with the L2 cache miss generated by Load 1.

Note that Load 1 in Figure 6.2 is an *address (pointer) load*. We distinguish between *address loads* and *data loads*. An *address load* is a load instruction that loads an address into its destination register that is later used to calculate the effective address of itself or another load instruction (Load 3 is also an address load). A *data load* is a load whose

destination register is not used to calculate the effective address of another load instruction (Load 2 is a data load). We are interested in predicting the values of only *address loads*, not *data loads*, since address loads -by definition- are the only load instructions that can lead to the generation of dependent long-latency cache misses. In order to distinguish address loads from data loads in hardware, we bound the values AVD can take. We only consider predicting the values of load instructions that have -in the past- satisfied the equation:

$$-MaxAVD \leq AVD(L) \leq MaxAVD$$

where $MaxAVD$ is a constant set at the design time of the AVD predictor. In other words, in order to be identified as an address load, the data value of a load instruction needs to be *close enough* to its effective address. If the AVD is too large, it is likely that the value that is being loaded by the load instruction is not an address.[4] Note that this mechanism is similar to the mechanism proposed by Cooksey et al. [29] to identify address loads in hardware. Their mechanism identifies a load as an address load if the upper N bits of the effective address of the load match the upper N bits of the value being loaded.

6.3 Why Do Stable AVDs Occur?

Stable AVDs occur due to the regularity in the way data structures are allocated in memory by the program, which is sometimes accompanied by the regularity in the input data to the program. We examine the common code constructs in application programs that give rise to regular memory allocation patterns that result in stable AVDs for some address loads. For our analysis, we distinguish between what we call *traversal address loads* and

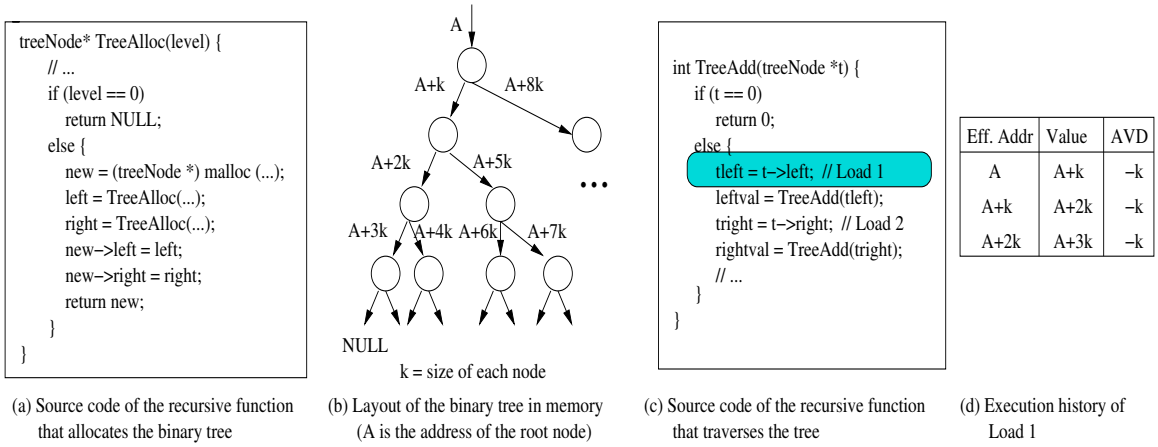
[4] An alternative mechanism is to have the compiler designate the address loads with a single bit augmented in the load instruction format of the ISA. We do not explore this option since our goal is to design a simple purely-hardware mechanism that requires no software or ISA support.

leaf address loads. A traversal address load is a static load instruction that produces an address that is later consumed by itself or another address load, such as in a linked list or tree traversal, `p = p->next` (e.g., Load 3 in Figure 6.2 is a traversal address load). A leaf address load produces an address that is later consumed by a data load (e.g., Load 1 in Figure 6.2 is a leaf address load).

6.3.1 Stable AVDs in Traversal Address Loads

A traversal address load may have a stable AVD if there is a pattern to the allocation and linking of the nodes of a linked data structure. If the allocation of the nodes is performed in a regular fashion, the nodes will have a constant distance in memory from one another. If a traversal load instruction later traverses the linked data structure nodes that have the same distance from one another, the traversal load can have a stable AVD.

Figure 6.3 shows an example from `treeadd`, a benchmark whose main data structure is a binary tree. In this benchmark, a binary tree is allocated in a regular fashion using a recursive function in which a node is allocated first and its left child is allocated next (Figure 6.3a). Each node of the tree is of the same size. The layout of an example resulting binary tree is shown in Figure 6.3b. Due to the regularity in the allocation of the nodes, the distance in memory between each node and its left child is constant. The binary tree is later traversed using another recursive function (Figure 6.3c). Load 1 in the traversal function traverses the nodes by loading the pointer to the left child of each node. This load instruction has a stable AVD as can be seen from its example execution history (Figure 6.3d). Load 1 has a stable AVD because the distance in memory between a node and its left child is constant. We found that this load causes 64% of all entries into runahead mode and predicting its value correctly enables the generation of dependent L2 misses (generated by the same instruction) during runahead mode. Similar traversal loads with stable AVDs exist in `twolf`, `mst`, and `vpr`, which employ linked lists, and `bisort`, `perimeter`, `tsp`, and



(a) Source code of the recursive function that allocates the binary tree

(b) Layout of the binary tree in memory (A is the address of the root node)

(c) Source code of the recursive function that traverses the tree

(d) Execution history of Load 1

Figure 6.3: An example from the `treeadd` benchmark showing how stable AVDs can occur for traversal address loads.

`voronoi`, which employ binary- or quad-trees.

As evident from this example, the stability of AVDs in traversal address loads is dependent not only on the way the programmer sizes the data structures and allocates memory for them but also on the behavior of the memory allocator. If the memory allocator allocates chunks of memory in a regular fashion (e.g., allocating fixed-size chunks from a contiguous section of memory), the likelihood of the occurrence of stable AVDs increases. On the other hand, if the behavior of the memory allocator is irregular, the distance in memory between a node and the node(s) it is linked to may be totally unpredictable; hence, the resulting AVDs would not be stable.

We also note that stable AVDs occurring due to the regularity in the allocation and linking of the nodes can disappear if the linked data structure is significantly re-organized (e.g., sorted) during run-time, unless the re-organization of the data structure is performed in a regular fashion. Therefore, AVD prediction may not work for traversal address loads in applications that require extensive modifications to the linkages in linked data structures.

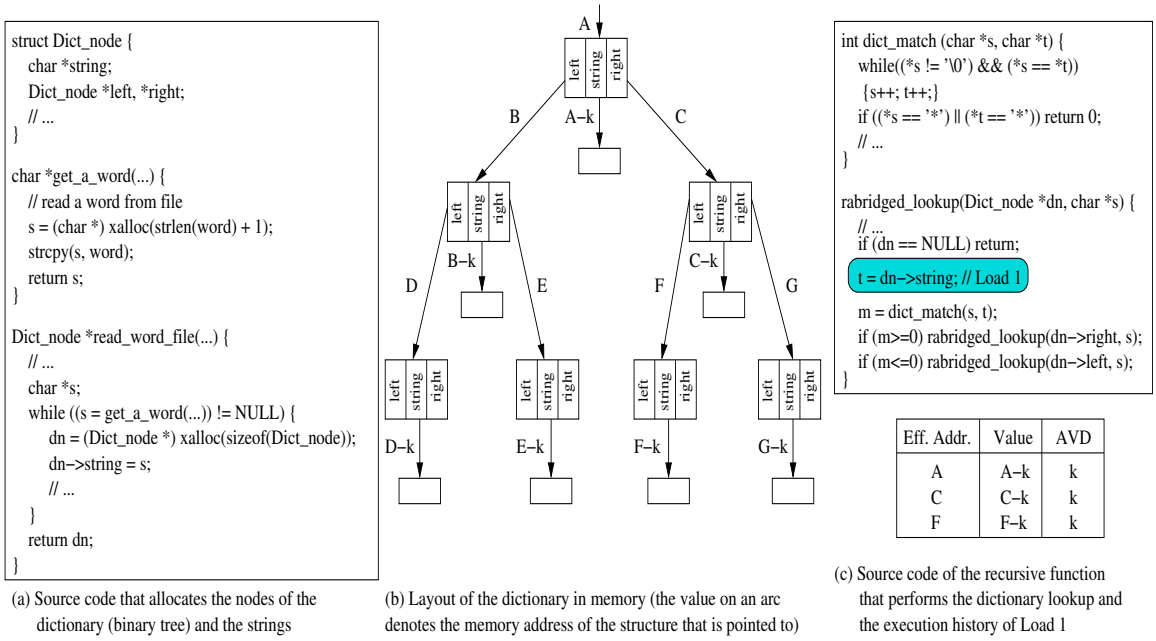
6.3.2 Stable AVDs in Leaf Address Loads

A leaf address load may have a stable AVD if the allocation of a data structure node and the allocation of a field that is linked to the node via a pointer are performed in a regular fashion. We show two examples to illustrate this behavior.

Figure 6.4 shows an example from `parser`, a benchmark that parses an input file and looks up the parsed words in a dictionary. The dictionary is constructed at the startup of the program. It is stored as a sorted binary tree. Each node of the tree is a `Dict_node` structure that contains a pointer to the `string` corresponding to it as one of its fields. Both `Dict_node` and `string` are allocated dynamically as shown in Figure 6.4a. First, memory space for `string` is allocated. Then, memory space for `Dict_node` is allocated and it is linked to the memory space of `string` via a pointer. The layout of an example dictionary is shown in Figure 6.4b. In contrast to the binary tree example from `treeadd`, the distance between the nodes of the dictionary in `parser` is not constant because the allocation of the dictionary nodes is performed in a somewhat irregular fashion (not shown in Figure 6.4) and because the dictionary is kept sorted. However, the distance in memory between each node and its associated `string` is constant. This is due to the behavior of the `xalloc` function that is used to allocate the `string`s in combination with regularity in input data. We found that `xalloc` allocates a fixed-size block of memory for the `string`, if the length of the string is within a certain range. As the length of most `strings` falls into that range (i.e., the input data has regular behavior), the memory spaces allocated for them are of the same size.[5]

Words are later looked up in the dictionary using the `rabridged_lookup` func-

[5]The code shown in Figure 6.4a can be re-written such that memory space for a `Dict_node` is allocated first and the memory space for its associated `string` is allocated next. In this case, even though the input data may not be regular, the distance in memory between each node and its associated string would be constant. We did not perform this optimization in our baseline evaluations. However, the effect of this optimization is evaluated separately in Section 6.7.2.



(a) Source code that allocates the nodes of the dictionary (binary tree) and the strings

(b) Layout of the dictionary in memory (the value on an arc denotes the memory address of the structure that is pointed to)

(c) Source code of the recursive function that performs the dictionary lookup and the execution history of Load 1

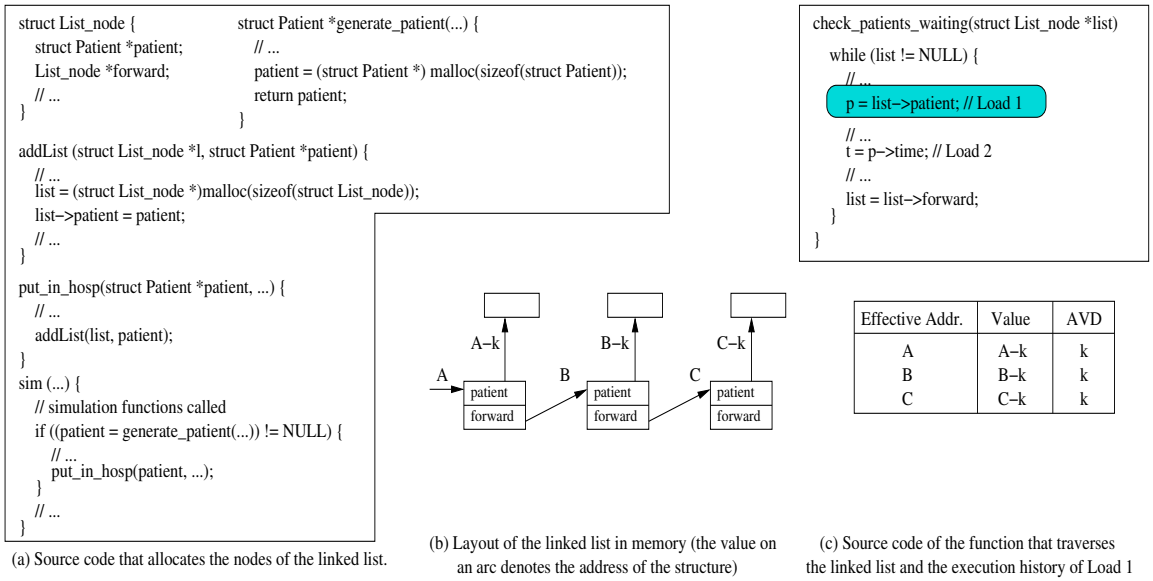
Figure 6.4: An example from the `parser` benchmark showing how stable AVDs can occur for leaf address loads.

tion (Figure 6.4c). This function recursively searches the binary tree and checks whether the `string` of each node is the same as the input word `s`. The `string` in each node is loaded by Load 1 (`dn->string`), which is a leaf address load that loads an address that is later dereferenced by data loads in the `dict_match` function. This load has a stable AVD, as shown in its example execution history, since the distance between a node and its associated string is constant. The values generated by Load 1 are hard to predict using a traditional value predictor because they do not follow a pattern. In contrast, the AVDs of Load 1 are quite easy to predict. We found that this load causes 36% of the entries into runahead mode and correctly predicting its value enables the execution of the dependent load instructions and the dependent conditional branch instructions in the `dict_match` function. Enabling the correct execution of dependent load instructions result in the generation of cache misses that could otherwise not be generated if Load 1's result were marked as INV.

Enabling the correct execution of dependent branch instructions results in the initiation of misprediction recovery for dependent mispredicted branches. This allows the processor to stay on the correct program path (i.e., to traverse the correct nodes in the dictionary) during runahead mode.

Note that stable AVDs occurring in leaf address loads continue to be stable even if the linked data structure is significantly re-organized at run-time. This is because such AVDs are caused by the regularity in the links between nodes and their fields rather than the regularity in the links between nodes and other nodes. The re-organization of the linked data structure changes the links between nodes and other nodes, but leaves intact the links between nodes and their fields.

Figure 6.5 shows an example from `health`, demonstrating the occurrence of stable AVDs in a linked list. This benchmark simulates a health care system in which a list of patients waiting to be serviced is maintained in a linked list. Each node of the linked list contains a pointer to the `patient` structure it is associated with. Each node and the `patient` structure are allocated dynamically as shown in Figure 6.5a. The allocation of these structures is performed in a regular fashion. First, memory space for a `patient` is allocated. Right after that, memory space for a `List_node` is allocated and it is linked to the `patient` via a pointer. Since `List_node` and `Patient` structures are of fixed size, the distance in memory between a node and its associated `patient` is constant as shown in the layout of the resulting linked list (Figure 6.5b). The linked list is later traversed in the `check_patients_waiting` function (Figure 6.5c). The `patient` associated with each node is loaded by Load 1 (`p = list->patient`), which is a leaf address load that is later dereferenced by a data load, Load 2 (`t = p->time`). Load 1 has a stable AVD as shown in its execution history. It causes 67% of the entries into runahead mode and predicting its value correctly enables the servicing of dependent L2 cache misses caused by Load 2.



(a) Source code that allocates the nodes of the linked list.

(b) Layout of the linked list in memory (the value on an arc denotes the address of the structure)

(c) Source code of the function that traverses the linked list and the execution history of Load 1

Figure 6.5: An example from the `health` benchmark showing how stable AVDs can occur for leaf address loads.

6.4 Design and Operation of a Recovery-Free AVD Predictor

An AVD predictor records the AVDs and information about the stability of the AVDs for address load instructions. The predictor is updated when an address load is retired. The predictor is accessed when a load misses in the L2 cache during runahead mode. If a stable AVD associated with the load is found in the predictor, the predicted value for the load is calculated using its effective address and the stable AVD. The predicted value is then returned to the processor to be written into the register file. The high-level organization of a processor employing an AVD predictor is shown in Figure 6.6.

Figure 6.7 shows the organization of the AVD predictor along with the hardware support needed to update/train it (Figure 6.7a) and the hardware support needed to make a prediction (Figure 6.7b). Each entry of the predictor consists of three fields: *Tag*, the upper bits of the program counter of the load that allocated the entry; *AVD*, the address-

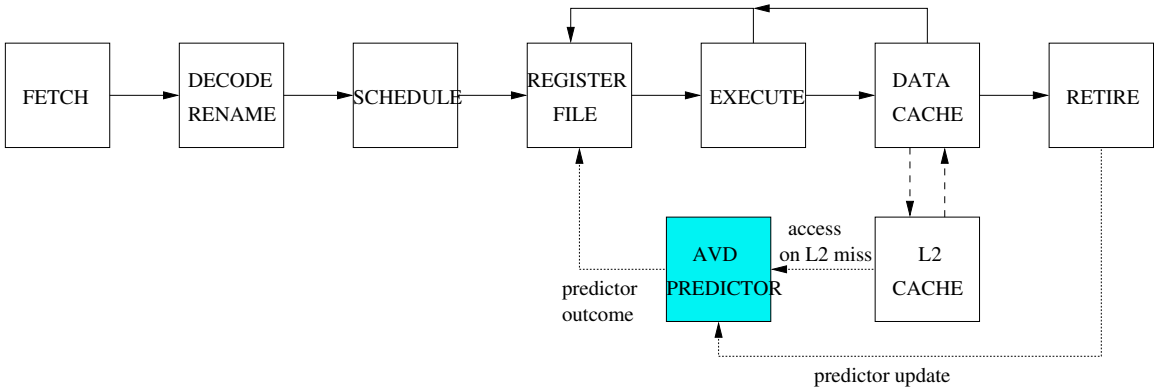
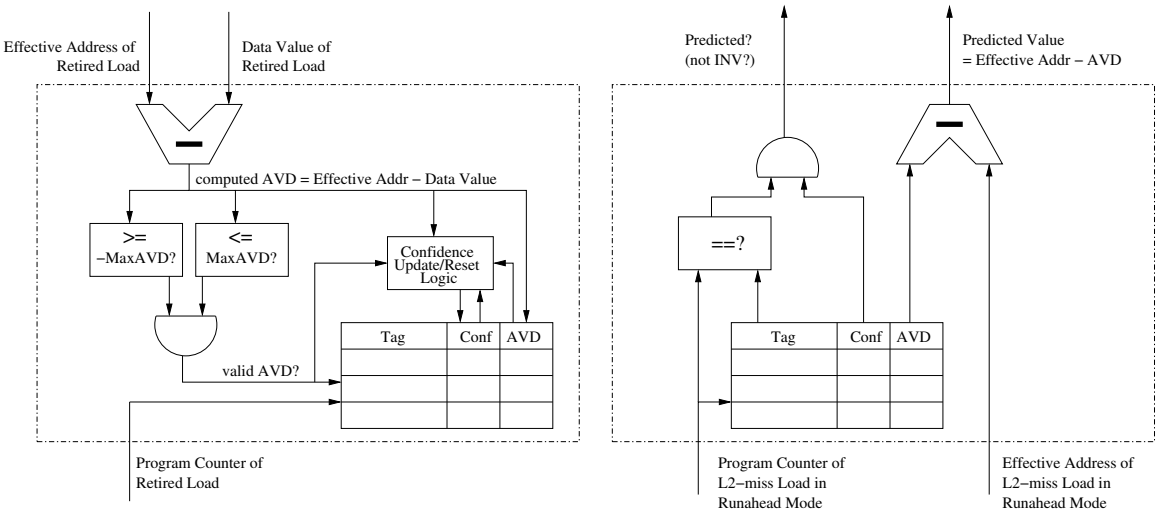


Figure 6.6: Organization of a processor employing an AVD predictor.

value delta that was recorded for the last retired load associated with the entry; *Confidence (Conf)*, a saturating counter that records the confidence of the recorded AVD (i.e., how many times the recorded AVD was seen consecutively). The confidence field is used to eliminate incorrect predictions for loads with unstable AVDs.



(a) Update Logic for the AVD Predictor

(b) Prediction Logic for the AVD Predictor

Figure 6.7: Organization of the AVD predictor and the hardware support needed for updating/accessing the predictor.

6.4.1 Operation

At initialization, the confidence counters in all the predictor entries are reset to zero. There are two major operations performed on the AVD predictor: update and prediction.

The predictor is updated when a load instruction is retired during normal mode. The predictor is accessed with the program counter of the retired load. If an entry does not already exist for the load in the predictor and if the load has a valid AVD, a new entry is allocated. To determine if the load has a valid AVD, the AVD of the instruction is computed and compared to the minimum and the maximum allowed AVD. If the computed AVD is within bounds [`-MaxAVD`, `MaxAVD`], the AVD is considered valid. On the allocation of a new entry, the computed AVD is written into the predictor and the confidence counter is set to one. If an entry already exists for the retired load, the computed AVD is compared with the AVD that is stored in the existing entry. If the two match, the confidence counter is incremented. If the AVDs do not match and the computed AVD is valid, the computed AVD is stored in the predictor entry and the confidence counter is set to one. If the computed AVD is not valid and the load instruction has an associated entry in the predictor, the confidence counter is reset to zero, but the stored AVD is not updated.[6]

The predictor is accessed when a load instruction misses in the L2 cache during runahead mode. The predictor is accessed with the program counter of an L2-miss load. If an entry exists for the load and if the confidence counter is saturated (i.e., above a certain confidence threshold), the value of the load is predicted. The predicted value is computed by subtracting the AVD stored in the predictor entry from the effective virtual address of the L2-miss load. If an entry does not exist for the load in the predictor, the value of the

[6] As an optimization, it is possible to *not update* the AVD predictor state, including the confidence counters, if the data value of the retired load is zero. A data value of zero has a special meaning for address loads, i.e., NULL pointer. This optimization reduces the training time or eliminates the need to re-train the predictor and thus helps benchmarks where loads that perform short traversals are common. The effect of this optimization on AVD predictor performance is evaluated in Section 6.7.1.

load is not predicted. Two outputs are generated by the AVD predictor: a *predicted* bit that informs the processor whether or not a prediction is generated for the load and the *predicted value*. If the *predicted* bit is set, the *predicted value* is written into the destination register of the load so that its dependent instructions read it and are executed. If the *predicted* bit is not set, the processor discards the *predicted value* and marks the destination register of the load as INV in the register file (as in conventional runahead execution) so that dependent instructions are marked as INV and their results are not used.

The AVD predictor does not require any hardware for state recovery on AVD or branch mispredictions. Branch mispredictions do not affect the state of the AVD predictor since the predictor is updated only by retired load instructions (i.e., there are no wrong-path updates). The correctness of the AVD prediction cannot be determined until the L2 miss that triggered the prediction returns back from main memory. We found that it is not worth updating the state of the predictor on an AVD misprediction detected when the L2 cache miss returns back from main memory, since the predictor will anyway be updated when the load is re-executed and retired in normal execution mode after the processor exits from runahead mode.

An AVD misprediction can occur only in runahead mode. When it occurs, instructions that are dependent on the predicted L2-miss load can produce incorrect results. This may result in the generation of incorrect prefetches or the overturning of correct branch predictions. However, since runahead mode is purely speculative, there is no need to recover the processor state on an AVD misprediction. We found that an incorrect AVD prediction is not necessarily harmful for performance. If the predicted AVD is close enough to the actual AVD of the load, dependent instructions sometimes still generate useful L2 cache misses that are later needed by the processor in normal mode. Hence, we do not initiate state recovery on AVD mispredictions that are resolved during runahead mode.

6.4.2 Hardware Cost and Complexity

Our goal in the design of the AVD predictor is to avoid high hardware complexity and large storage requirements, but to still improve performance by focusing on predicting the addresses of an important subset of address loads. Since the AVD predictor filters out the loads for which the absolute value of the AVD is too large (using the `MaxAVD` threshold), the number of entries required in the predictor does not need to be large. In fact, Section 6.6 shows that a 4-entry AVD predictor is sufficient to get most of the performance benefit of the described mechanism. The storage cost required for a 4-entry predictor is very small (212 bits[7]). The logic required to implement the AVD predictor is also relatively simple as shown in Figure 6.7. Furthermore, neither the update nor the access of the AVD predictor is on the critical path of the processor. The update is performed after retirement, which is not on the critical path. The access (prediction) is performed only for load instructions that miss in the L2 cache and it does not affect the critical L1 or L2 cache access times. Therefore, the complexity of the processor or the memory system is not significantly increased with the addition of an AVD predictor.

6.5 Performance Evaluation Methodology

We evaluate AVD prediction on eleven pointer-intensive and memory-intensive benchmarks from Olden [93] and SPEC CPU2000 integer benchmark suites. We examine seven memory-intensive benchmarks from the Olden suite which gain at least 10% performance improvement with a perfect L2 cache and the four relatively pointer-intensive benchmarks (`mcf`, `parser`, `twolf`, `vpr`) from the SPEC CPU2000 integer suite. All benchmarks were compiled for the Alpha EV6 ISA with the `-O3` optimization level. `Twolf`

[7] Assuming a 4-entry, 4-way AVD predictor with 53 bits per entry: 32 bits for the tag, 17 bits for the AVD (i.e. `MaxAVD`=65535), 2 bits for confidence, and 2 bits to support a True LRU (Least Recently Used) replacement policy.

and `vpr` benchmarks are simulated for 250 million instructions after skipping the program initialization code using a tool developed to identify a representative simulation interval in the benchmark (similar to SimPoint [100]). To reduce simulation time, `mcf` is simulated using the MinneSPEC reduced input set [61]. `Parser` is simulated using the test input set provided by SPEC. We used the simple, general-purpose memory allocator (malloc) provided by the standard C library on an Alpha OSF1 V5.1 system. We did not consider a specialized memory allocator that would further benefit AVD prediction.

Table 6.2 shows information relevant to our studies about the simulated benchmarks. Unless otherwise noted, performance improvements are reported in terms of execution time normalized to the baseline processor throughout this chapter. IPCs of the evaluated processors, if needed, can be computed using the baseline IPC performance numbers provided in Table 6.2 and the normalized execution times. In addition, the fraction of L2 misses that are due to address loads is shown for each benchmark since our mechanism aims to predict the addresses loaded by address loads. We note that in all benchmarks except `vpr`, at least 25% of the L2 cache data misses are caused by address loads. Benchmarks from the Olden suite are more address-load intensive than the set of pointer-intensive benchmarks in the SPEC CPU2000 integer suite. Hence, we expect AVD prediction to perform better on Olden applications.

	bisort	health	mst	perimeter	treeadd	tsp	voronoi	mcf	parser	twolf	vpr
Input Data Set	250,000 integers	5 levels 500 iters	512 nodes	4K x 4K image	1024K nodes	100,000 cities	20,000 points	small red	test.in	ref	ref
Simulated instruction count	468M	197M	88M	46M	191M	1050M	139M	110M	412M	250M	250M
Baseline IPC	1.07	0.05	1.67	0.92	0.90	1.45	1.31	0.97	1.33	0.73	0.88
L2 misses per 1K instructions	1.03	41.59	5.60	4.27	4.33	0.67	2.41	29.60	1.05	2.37	1.69
% misses due to address loads	72.1%	73.5%	33.8%	62.9%	57.6%	46.2%	78.6%	50.3%	30.1%	26.3%	2.1%

Table 6.2: Relevant information about the benchmarks with which AVD prediction is evaluated. IPC and L2 miss rates are shown for the baseline runahead processor.

The baseline microarchitecture model used for the evaluation of AVD prediction is an Alpha processor (described in Section 4.2) that implements runahead execution. The machine model of the processor simulated is the same as detailed in Table 4.5 except the processor does not employ the stream-based prefetcher. Our initial evaluations of AVD prediction excludes the prefetcher from the model because we would like to isolate and analyze the benefits of AVD prediction. Interaction between AVD prediction and stream-based prefetching is later examined in detail in Section 6.8.2.

6.6 Performance of the Baseline AVD Prediction Mechanism

Figure 6.8 shows the performance improvement obtained if the baseline runahead execution processor is augmented with the AVD prediction mechanism. We model an AVD predictor with a `MaxAVD` of 64K. A prediction is made if the confidence counter has a value of 2 (i.e., if the same AVD was seen consecutively in the last two executions of the load). On average, the execution time is improved by 12.6% (5.5% when `health` is excluded) with the use of an infinite-entry AVD predictor. No performance degradation is observed on any benchmark. Benchmarks that have a very high L2 cache miss rate, most of which is caused by address loads (`health`, `perimeter`, and `treeadd` as seen in Table 6.2), see the largest improvements in performance. Benchmarks with few L2 misses caused by address loads (e.g. `vpr`) do not benefit from AVD prediction.

A 32-entry, 4-way AVD predictor improves the execution time as much as an infinite-entry predictor for all benchmarks except `twolf`. In general, as the predictor size decreases, the performance improvement provided by the predictor also decreases. However, even a 4-entry AVD predictor improves the average execution time by 11.0% (4.0% without `health`). Because AVD prediction aims to predict the values produced by a regular subset of address loads, it does not need to keep track of data loads or address loads with very large AVDs. Thus, the number of load instructions competing for entries in the AVD predictor is fairly small, and a small predictor is good at capturing them.

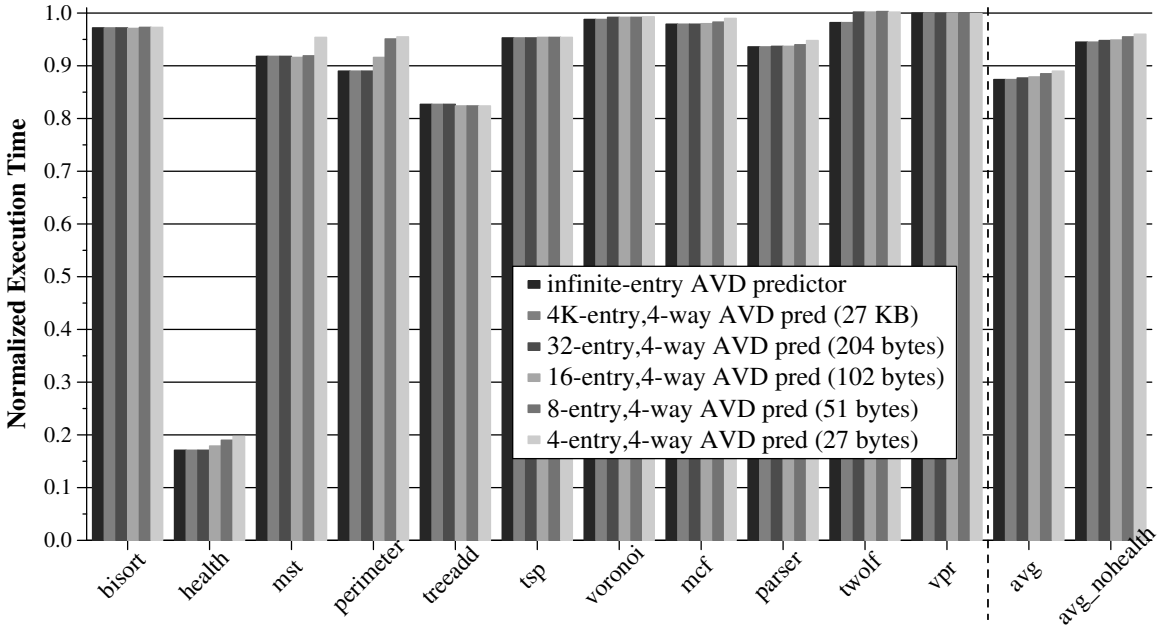


Figure 6.8: AVD prediction performance on a runahead processor.

6.6.1 Effect of MaxAVD

As explained in Section 6.2, `MaxAVD` is used to dynamically determine which loads are address loads. Choosing a larger `MaxAVD` results in more loads being identified - perhaps incorrectly- as address loads and may increase the contention for entries in the AVD predictor. A smaller `MaxAVD` reduces the number of loads identified as address loads and thus reduces contention for predictor entries, but it may eliminate some address loads with stable AVDs from being considered for AVD prediction. The choice of `MaxAVD` also affects the size of the AVD predictor since the number of bits needed to store the AVD is determined by `MaxAVD`. Figures 6.9 and 6.10 show the effect of a number of `MaxAVD` choices on the performance improvement provided by, respectively, 16-entry and 4-entry AVD predictors.

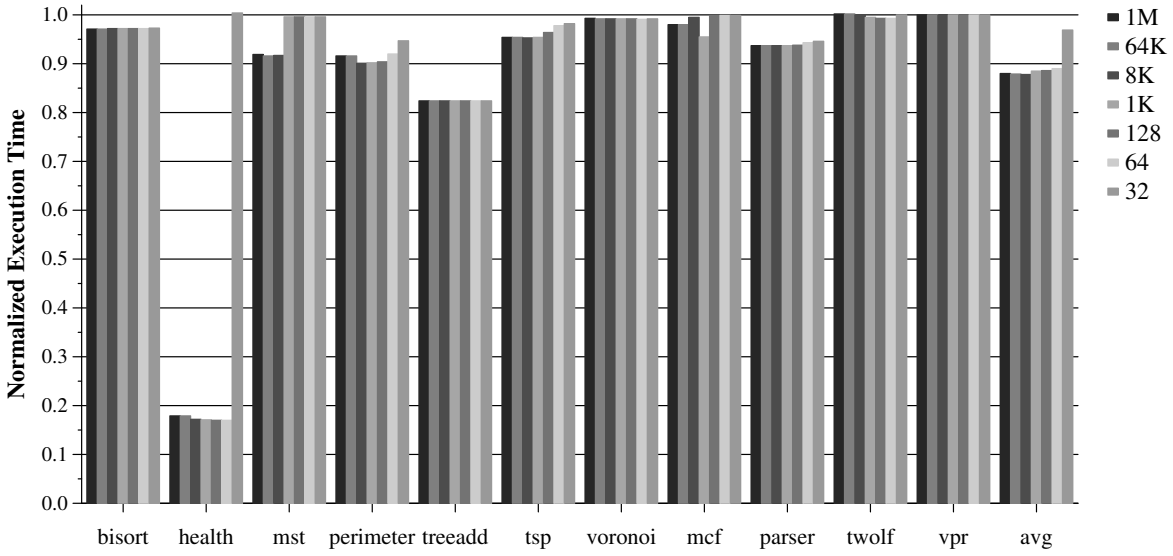


Figure 6.9: Effect of `MaxAVD` on execution time (16-entry AVD predictor).

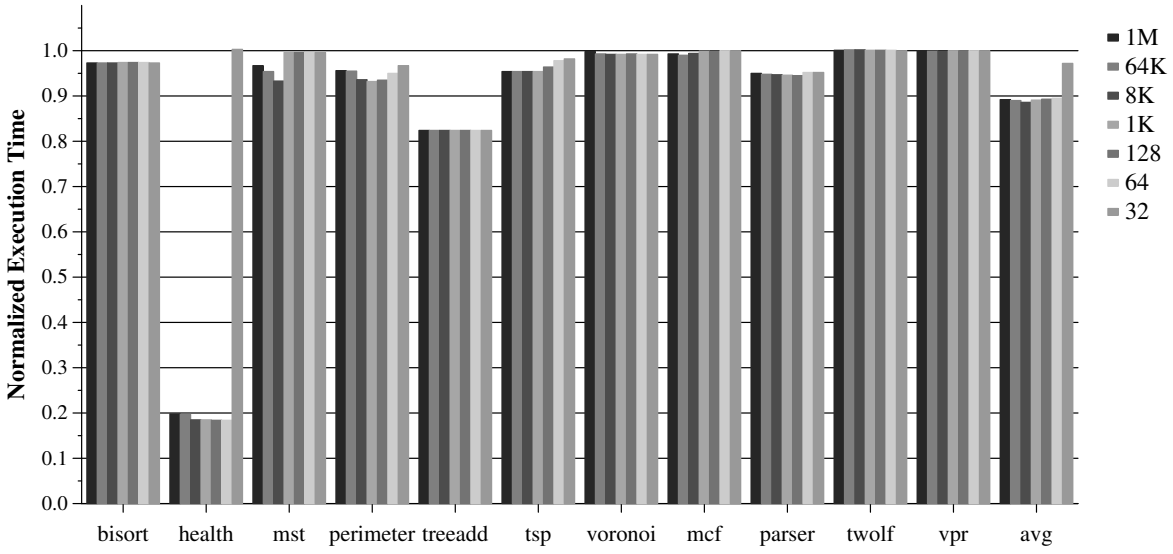


Figure 6.10: Effect of `MaxAVD` on execution time (4-entry AVD predictor).

The best performing MaxAVD value is 64K for the 16-entry predictor and 8K for the 4-entry predictor. Unless the AVD is too small (in which case very few address loads are actually identified as address loads), performance is not significantly affected by MaxAVD. However, with a 4-entry predictor, a large (1M or 64K) MaxAVD provides less performance benefit than smaller MaxAVD values in some benchmarks due to the increased contention for predictor entries. We found that most address loads with stable AVDs have AVDs that are within 0-8K range (except for some loads that have stable AVDs within 32K-64K range in mcf). This behavior is expected because, as shown in code examples in Section 6.3, stable AVDs usually occur due to regular memory allocation patterns that happen close together in time. Therefore, addresses that are linked in data structures are close together in memory, resulting in small, stable AVDs in loads that manipulate them.

6.6.2 Effect of Confidence

Figure 6.11 shows the effect of the confidence threshold needed to make an AVD prediction on performance. A confidence threshold of 2 provides the largest performance improvement for the 16-entry AVD predictor. Not using confidence (i.e., a confidence threshold of 0) in an AVD predictor significantly reduces the performance of the runahead processor because it results in the incorrect prediction of the values of many address loads that do not have stable AVDs. For example, in bisort most of the L2-miss address loads are traversal address loads. Since the binary tree traversed by these loads is heavily modified (sorted) during run-time, these traversal address loads do not have stable AVDs. A 16-entry AVD predictor that does not use confidence generates predictions for all these loads but increases the execution time by 180% since almost all the predictions are incorrect. Large confidence values (7 or 15) are also undesirable because they significantly reduce the prediction coverage for address loads with stable AVDs and hence reduce the performance improvement.

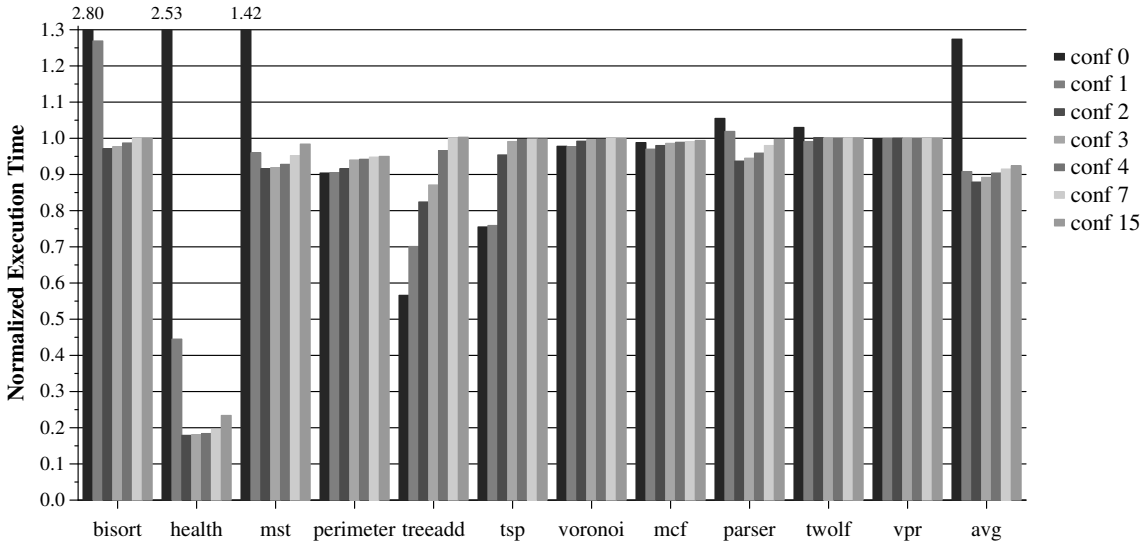


Figure 6.11: Effect of confidence threshold on execution time.

6.6.3 Coverage, Accuracy, and MLP Improvement

Figures 6.12 and 6.13 show the effect of the confidence threshold on the coverage and accuracy of the predictor. Coverage is computed as the percentage of L2-miss address loads executed in runahead mode whose values are predicted by the AVD predictor. Accuracy is the percentage of predictions where the predicted value is the same as the actual value. With a confidence threshold of two, about 30% of the L2-miss address loads are predicted and about one half of the predictions are correct, on average. We found that incorrect predictions are not necessarily harmful for performance. Since runahead mode does not have any correctness requirements, incorrect predictions do not result in any recovery overhead. In some cases, even though the predicted AVD is not exactly correct, it is close enough to the correct AVD that it leads to the pre-execution of dependent instructions that generate cache misses that are later needed by correct execution. An example showing how inaccurate AVD predictions can result in useful prefetch requests is provided in Section 6.7.1.

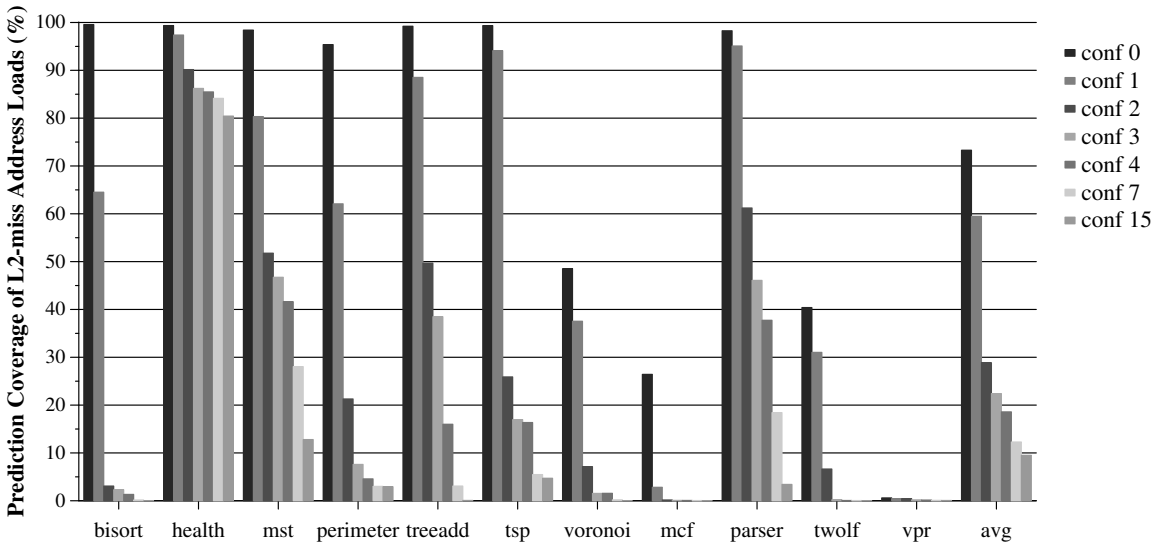


Figure 6.12: AVD prediction coverage for a 16-entry predictor.

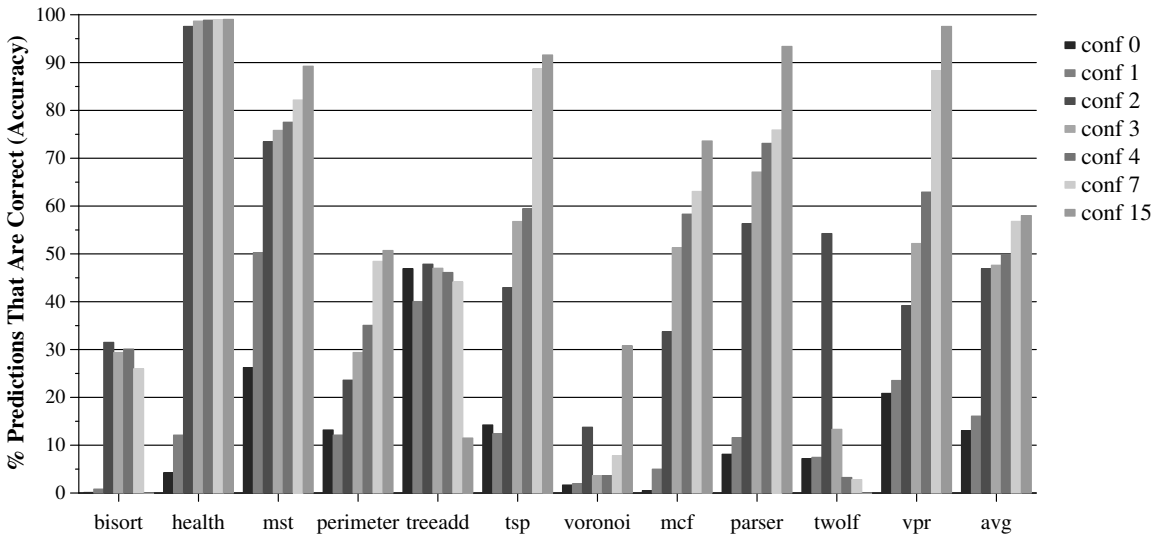


Figure 6.13: AVD prediction accuracy for a 16-entry AVD predictor.

Since AVD prediction can provide prefetching benefits even though the predicted AVD is not accurate, a more relevant metric for measuring the goodness of the AVD predictor is the *improvement in the memory-level parallelism* [23]. Table 6.3 shows the increase in memory-level parallelism achieved with a 16-entry AVD predictor by showing the average number of useful L2 cache misses generated in a runahead period with and without AVD prediction. Note that benchmarks that show large increases in the average number of useful L2 misses with an AVD predictor also show large increases in performance.

	bisort	health	mst	perimeter	treeadd	tsp	voronoi	mcf	parser	twolf	vpr	avg
L2 misses - baseline runahead	2.01	0.03	7.93	1.45	1.02	0.19	0.81	11.51	0.12	0.84	0.94	2.44
L2 misses - 16-entry AVD pred	2.40	6.36	8.51	1.67	1.53	0.25	0.90	12.05	0.50	0.87	0.94	3.27
% reduction in execution time	2.9%	82.1%	8.4%	8.4%	17.6%	4.5%	0.8%	2.1%	6.3%	0.0%	0.0%	12.1%

Table 6.3: Average number of useful L2 cache misses generated during a runahead period with a 16-entry AVD predictor.

6.6.4 AVD Prediction and Runahead Efficiency

Efficiency is an important concern in designing a runahead execution processor as described in Chapter 5. AVD prediction improves efficiency because it both increases performance and decreases the number of executed instructions in a runahead processor. Figure 6.14 shows that employing AVD prediction reduces the number of instructions processed in a runahead processor by 13.3% with a 16-entry predictor and by 11.8% with a 4-entry predictor. AVD prediction reduces the number of executed instructions because it is able to parallelize and service dependent L2 cache misses during a single runahead period. In a runahead processor without AVD prediction, two dependent L2 misses would cause two separate runahead periods, which are overlapping, and hence they would result in the execution of many more instructions than can be executed in a single runahead period.[8]

[8]In fact, an extreme case of inefficiency caused by dependent L2 misses can be seen in health. In this benchmark, using runahead execution increases the number of executed instructions by 27 times, but results in a 4% *increase* in execution time! Using AVD prediction greatly reduces this inefficiency.

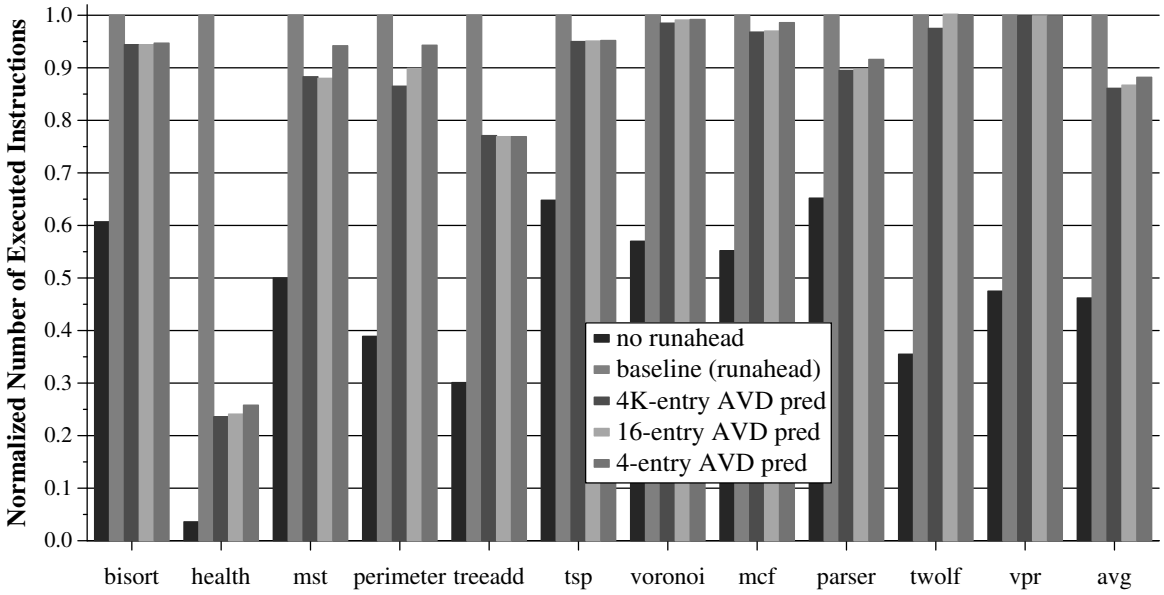


Figure 6.14: Effect of AVD prediction on the number of executed instructions.

Figure 6.15 shows the normalized number of useful and useless runahead periods in the baseline runahead processor and the runahead processor with a 16-entry AVD predictor. Remember that a runahead period is defined to be useful if it results in the generation of at least one L2 cache miss that cannot be generated by the processor's fixed size instruction window and that is later needed by a correct-path instruction in normal mode. On average, AVD prediction reduces the number of runahead periods by 18%. A significant fraction of the useless runahead periods is eliminated with AVD prediction. In some benchmarks, like `mst` and `treeadd`, the number of useful runahead periods is also reduced with AVD prediction. This is because AVD prediction increases the degree of usefulness of useful runahead periods. With AVD prediction, an otherwise useful runahead period results in the parallelization of more L2 cache misses, which eliminates later runahead periods (useful or useless) that would have occurred in the baseline runahead processor. The average number of useful L2 cache misses discovered in a useful runahead period is 6.5 without AVD

prediction and 10.3 with AVD prediction.

We note that, even with AVD prediction, a large fraction (on average 61%) of the runahead periods remain useless. Hence, AVD prediction cannot eliminate *all* useless runahead periods. However, a processor employing AVD prediction can be augmented with the previously proposed techniques for efficient runahead processing (described in Chapter 5) to further improve the efficiency of runahead execution. We examine the effect of combining these techniques and AVD prediction in Section 6.8.1.

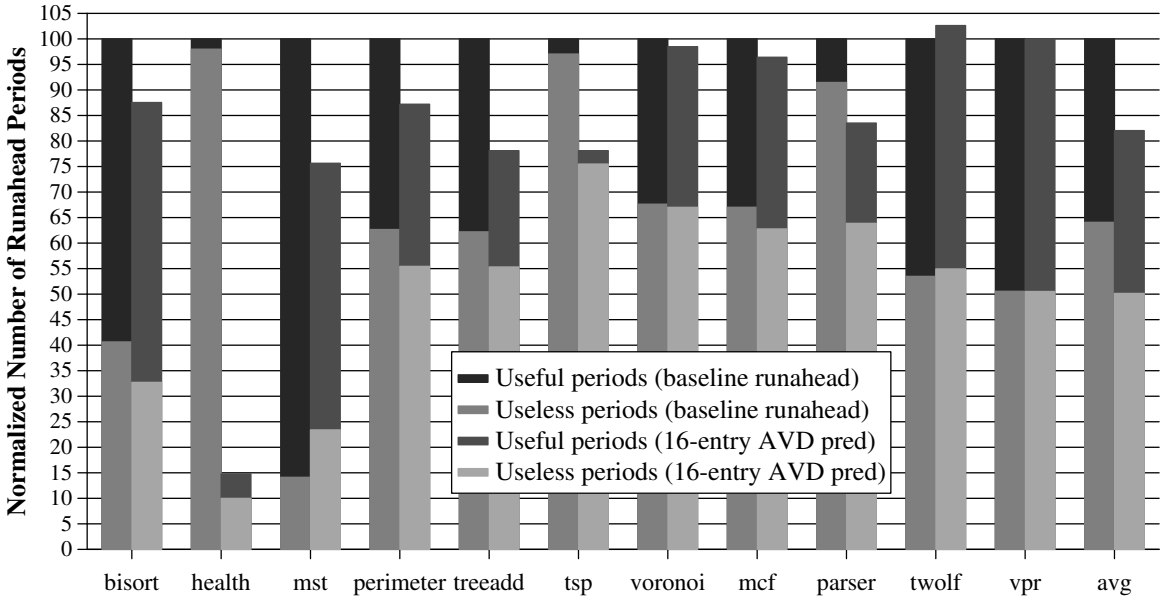


Figure 6.15: Effect of AVD prediction on the number of runahead periods.

6.6.5 Effect of Memory Latency

Figure 6.16 shows the normalized average execution time with and without AVD prediction for five processors with different memory latencies. In this figure, execution time is normalized to the baseline runahead processor independently for each memory latency. Average execution time improvement provided by a 16-entry AVD predictor ranges

from 8.0% for a relatively short 100-cycle memory latency to 13.5% for a 1000-cycle memory latency. AVD prediction consistently improves the effectiveness of runahead execution on processors with different memory latencies, including the one with a short, 100-cycle memory latency where runahead execution is very ineffective and actually increases the execution time by 2%. We conclude that a low-cost AVD predictor is beneficial for runahead processors with both relatively short and long memory latencies.

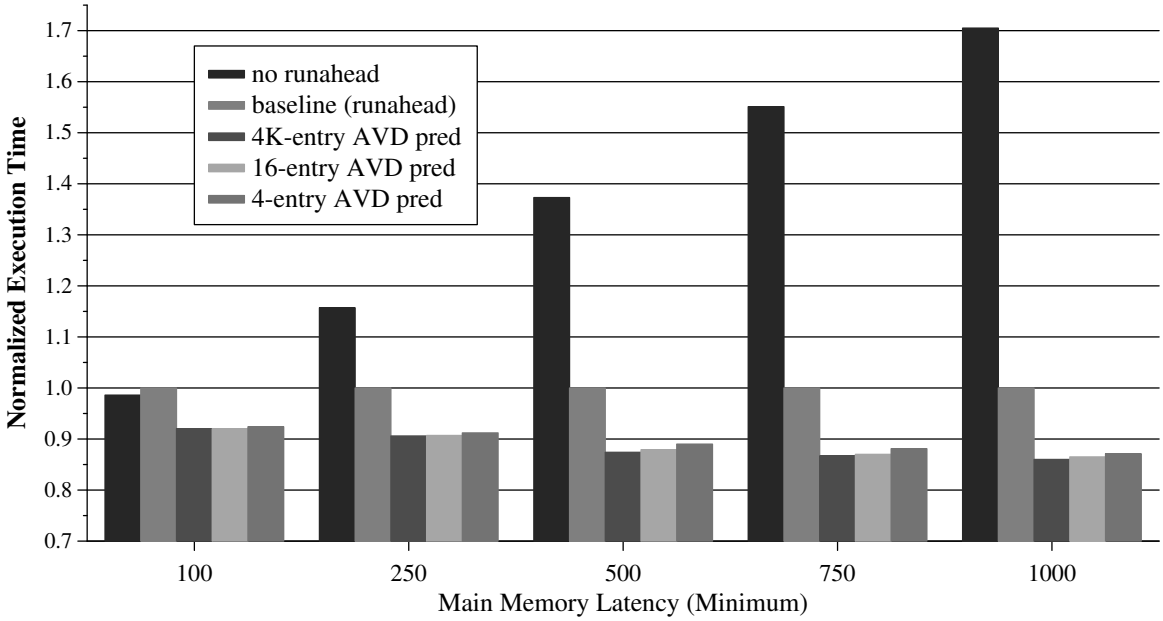


Figure 6.16: Effect of memory latency on AVD predictor performance.

6.6.6 AVD Prediction vs. Stride Value Prediction

We compare the proposed AVD predictor to stride value prediction [97]. When an L2-miss is encountered during runahead mode, the stride value predictor (SVP) is accessed for a prediction. If the SVP generates a confident prediction, the value of the L2-miss load is predicted. Otherwise, the L2-miss load marks its destination register as INV. Figure 6.17 shows the normalized execution times obtained with an AVD predictor, a stride value pre-

dictor, and a hybrid AVD-stride value predictor.[9] Stride value prediction is more effective when the predictor is larger, but it provides only 4.5% (4.7% w/o `health`) improvement in average execution time even with a 4K-entry predictor versus the 12.6% (5.5% w/o `health`) improvement provided by the 4K-entry AVD predictor. With a small, 16-entry predictor, stride value prediction improves the average execution time by 2.6% (2.7% w/o `health`), whereas AVD prediction results in 12.1% (5.1% w/o `health`) performance improvement. The filtering mechanism (i.e., the `MaxAVD` threshold) used in the AVD predictor to identify and predict only address loads enables the predictor to be small and still provide significant performance improvements.

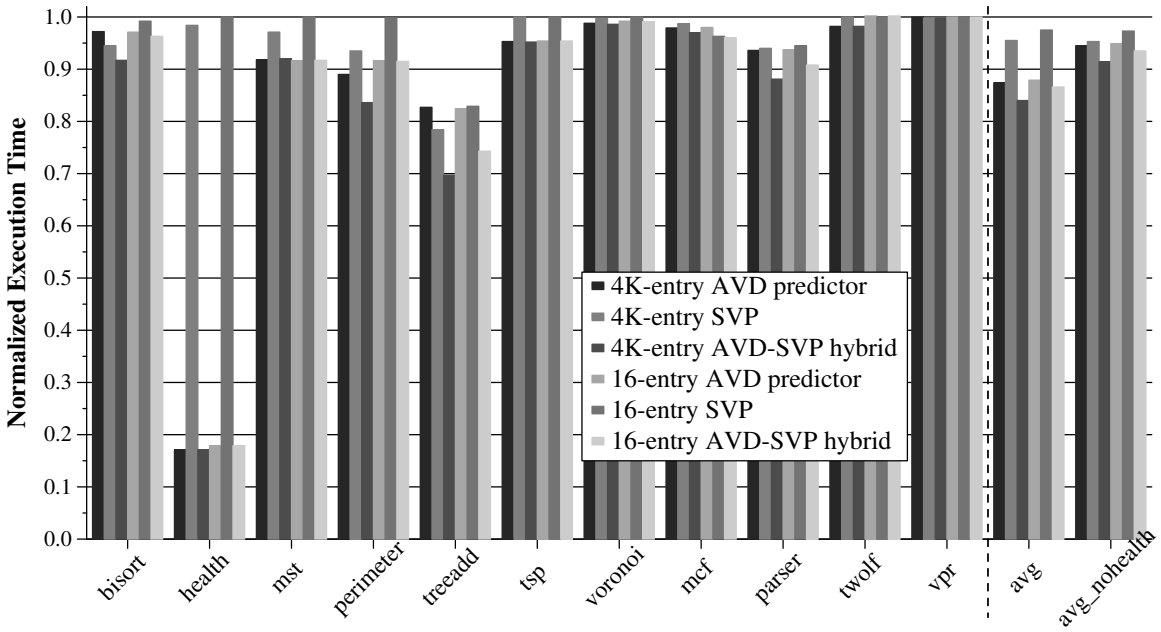


Figure 6.17: AVD prediction vs. stride value prediction.

[9] In our experiments, the hybrid AVD-SVP predictor does not require extra storage for the selection mechanism. Instead, the prediction made by the SVP is given higher priority than the prediction made by the AVD predictor. If the SVP generates a confident prediction for an L2-miss load, its prediction is used. Otherwise, the prediction made by the AVD predictor is used, if confident.

The benefits of stride and AVD predictors overlap for traversal address loads. Both predictors can capture the values of traversal address loads if the memory allocation pattern is regular. Many L2 misses in `treeadd` are due to traversal address loads, which is why both SVP and AVD predictors perform very well and similarly for this benchmark.

Most leaf address loads cannot be captured by SVP, whereas an AVD predictor can capture those with constant AVD patterns. The benchmark `health` has many AVD-predictable leaf address loads, an example of which was described in detail in Section 6.3.2. The traversal address loads in `health` are irregular and therefore cannot be captured by either SVP or AVD. Hence, AVD prediction provides significant performance improvement in `health` whereas SVP does not. We found that benchmarks `mst`, `perimeter`, and `tsp` also have many leaf address loads that can be captured with an AVD predictor but not with SVP.

In contrast to an AVD predictor, an SVP is able to capture data loads with constant strides. For this reason, SVP significantly improves the performance of `parser`. In this benchmark, correctly value-predicted L2-miss data loads lead to the execution and correct resolution of dependent branches that were mispredicted by the branch predictor. SVP improves the performance of `parser` by keeping the processor on the correct path during runahead mode rather than by allowing the parallelization of dependent cache misses.

Figure 6.17 also shows that combining stride value prediction and AVD prediction results in a larger performance improvement than that provided by either of the prediction mechanisms alone. For example, a 16-entry hybrid AVD-SVP predictor results in 13.4% (6.5% w/o health) improvement in average execution time. As shown in code examples in Section 6.3, address-value delta predictability is different in nature from stride value predictability. A load instruction can have a predictable AVD but not a predictable stride, and vice versa. Therefore, an AVD predictor and a stride value predictor sometimes generate predictions for loads with different behavior, resulting in increased performance improve-

ment when they are combined. This effect is especially salient in `parser`, where we found that the AVD predictor is good at capturing leaf address loads and the SVP is good at capturing zero-stride data loads.

6.6.7 Simple Prefetching with AVD Prediction

So far, we have employed AVD prediction for value prediction purposes, i.e., for predicting the data value of an L2-miss address load and thus enabling the pre-execution of dependent instructions that may generate long-latency cache misses. AVD prediction can also be used for simple prefetching without value prediction. This section evaluates the use of AVD prediction for simple prefetching on the runahead processor and shows that the major performance benefit of AVD prediction comes from enabling the pre-execution of dependent instructions.

In the simple prefetching mechanism we evaluate, the value of an L2-miss address load is predicted using AVD prediction during runahead mode. Instead of writing this value into the register file and enabling the execution of dependent instructions, the processor generates a memory request for the predicted value by treating the value as a memory address. A prefetch request for the next and previous sequential cache lines are also generated, since the data structure at the predicted memory address can span multiple cache lines. The destination register of the L2-miss address load is marked as INV in the register file, just like in baseline runahead execution. This mechanism enables the prefetching of only the address loaded by an L2-miss address load that has a stable AVD. However, in contrast to using an AVD predictor for value prediction, it does not enable the prefetches that can be generated further down the dependence chain of an L2-miss load through the execution of dependent instructions.

Figure 6.18 shows the normalized execution times when AVD prediction is used for simple prefetching and when AVD prediction is used for value prediction as evaluated

in previous sections. AVD prediction consistently provides higher performance improvements when used for value prediction than when used for simple prefetching. A 16-entry AVD predictor results in 12.1% performance improvement when it is used for value prediction versus 2.5% performance improvement when it is used for simple prefetching. Hence, the major benefit of AVD prediction comes from the prefetches generated by the execution of the instructions on the dependence chain of L2-miss address loads rather than the prefetching of only the addresses loaded by L2-miss address loads.

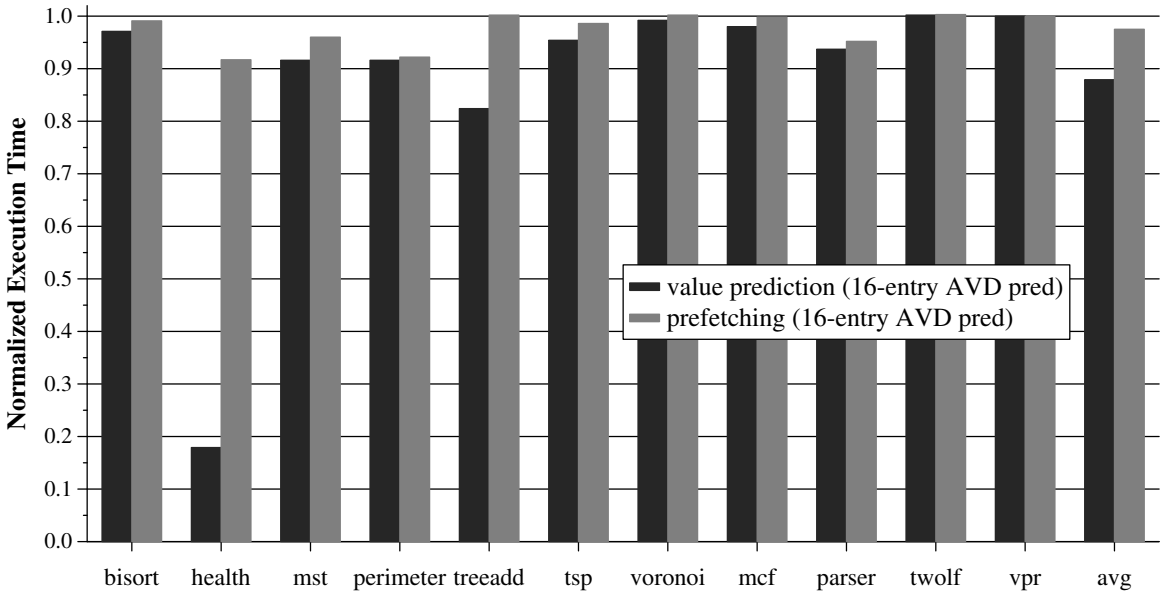


Figure 6.18: AVD prediction performance with simple prefetching.

6.6.8 AVD Prediction on Conventional Processors

We have shown the performance impact of using AVD prediction on runahead execution processors. However, AVD prediction is applicable not only to runahead execution processors. Less aggressive conventional out-of-order execution processors that do not implement runahead execution can also utilize AVD prediction to overcome the serialization

of dependent load instructions.

Figure 6.19 shows the normalized execution times when AVD prediction is used for simple prefetching (as described in Section 6.6.7) and value prediction on a conventional out-of-order processor.[10] Note that execution time is normalized to the execution time on the conventional out-of-order processor. Using a 16-entry AVD predictor for value prediction improves the average execution time on the conventional out-of-order processor by 4%. Using the same AVD predictor for simple prefetching improves the average execution time by 3.2%. The comparison of these results with the impact of AVD prediction on the runahead execution processor shows that AVD prediction, when used for value prediction, is more effective on the runahead execution processor with the same instruction window size. Since runahead execution enables the processor to execute many more instructions than a conventional out-of-order processor while an L2 miss is in progress, it exposes more dependent load instructions than an out-of-order processor with the same instruction window size. The correct prediction of the values of these load instructions results in higher performance improvements on a runahead processor.

6.7 Hardware and Software Optimizations for AVD Prediction

The results presented in the previous section were based on the baseline AVD predictor implementation described in Section 6.4. This section describes one hardware optimization and one software optimization that increases the benefits of AVD prediction by taking advantage of the data structure traversal and memory allocation characteristics in application programs.

[10]The parameters for the conventional out-of-order processor are the same as described in Section 6.5, except the processor does not employ runahead execution. The simple prefetching and value prediction mechanisms evaluated on out-of-order processors are employed for L2-miss loads. We examined using these two mechanisms for all loads or L1-miss loads, but did not see significant performance differences.

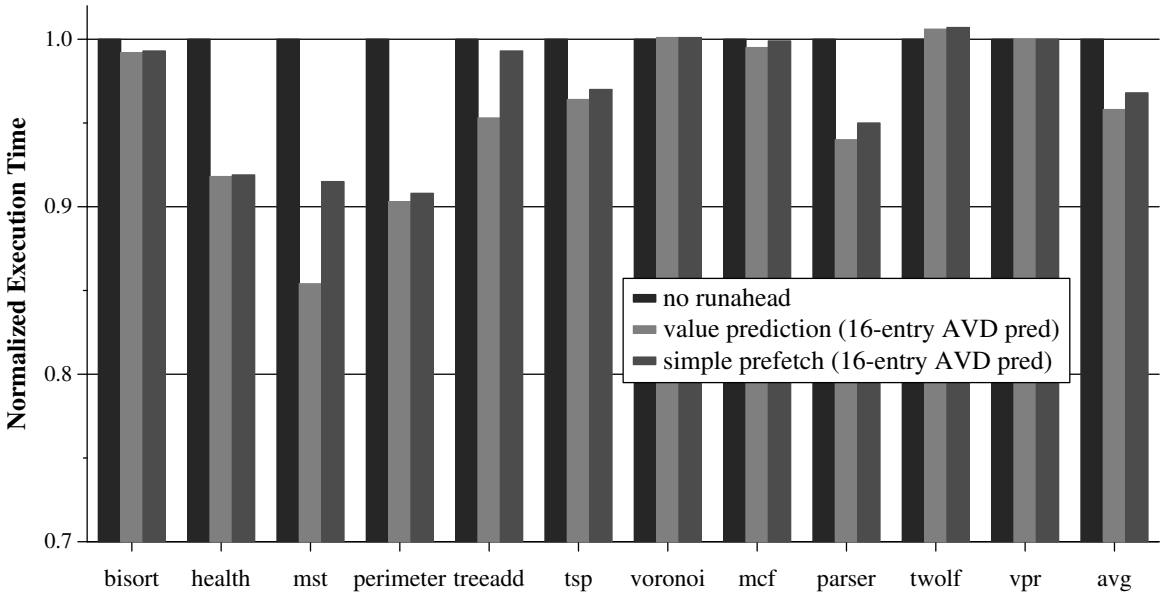


Figure 6.19: AVD prediction performance on a non-runahead processor.

6.7.1 NULL-Value Optimization

In the AVD predictor we have evaluated, the confidence counter of an entry is reset if the computed AVD of the retired address load associated with the entry is not valid (i.e., not within bounds [−MaxAVD, MaxAVD]). The AVD of a load instruction with a data value of zero is almost always invalid because the effective addresses computed by load instructions tend to be very large in magnitude. As a result, the confidence counter of an entry is reset if the associated load is retired with a data value of 0 (zero). For address loads, a zero data value has a special meaning: a NULL pointer is being loaded. This indicates the end of a linked data structure traversal. If a NULL pointer is encountered for an address load, it may be better *not to reset* the confidence counter for the corresponding AVD because the AVD of the load may otherwise be stable except for the intermittent instabilities caused by NULL pointer loads. This section examines the performance impact of not updating the AVD predictor if the value loaded by a retired address load is zero. We

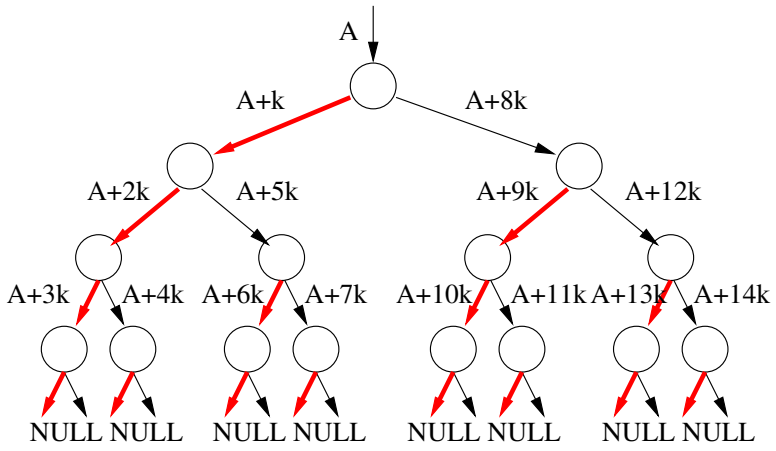
call this optimization the *NULL-value optimization*.

If the AVD of a load is stable except when a NULL pointer is loaded, resetting the confidence counter upon encountering a NULL pointer may result in a reduction in the prediction coverage of an AVD predictor. We show why this can happen with an example. Figure 6.20 shows an example binary tree that is traversed by the `treeadd` program. The tree is traversed with the source code shown in Figure 6.3. The execution history of the load that accesses the left child of each node (Load 1 in Figure 6.3) is shown in Table 6.4. This table also shows the predictions for Load 1 that would be made by two different AVD predictors: one that resets the confidence counters on a NULL value and one that does not change the confidence counters on a NULL value. Both predictors have a confidence threshold of 2. To simplify the explanation of the example, we assume that the predictor is updated before the next dynamic instance of Load 1 is executed.[11]

The execution history of Load 1 shows that not updating the AVD predictor on a NULL value is a valuable optimization. If the confidence counter for Load 1 in the AVD predictor is reset on a NULL data value, the AVD predictor generates a prediction for only 3 instances of Load 1 out of a total of 15 dynamic instances (i.e., coverage = 20%). Only one of these predictions is correct (i.e., accuracy = 33%). In contrast, if the AVD predictor is not updated on a NULL data value, it would generate a prediction for 13 dynamic instances (coverage = 87%), 5 of which are correct (accuracy = 38%).[12] Hence, not updating the AVD predictor on NULL data values significantly increases the coverage without degrading the accuracy of the predictor since the AVD for Load 1 is stable except when it loads a NULL pointer.

[11]Note that this may not be the case in an out-of-order processor. Our simulations faithfully model the update of the predictor based only on information available to the hardware.

[12]Note that, even though many of the predicted AVDs are incorrect in the latter case, the predicted values are later used as addresses by the same load instruction. Thus, AVD prediction can provide prefetching benefits even if the predicted AVDs are not correct.



k: size of each node (k < MaxAVD)
A: virtual address of the root of the tree (A > MaxAVD)

Figure 6.20: An example binary tree traversed by the `treeadd` program. Links traversed by Load 1 in Figure 6.3 are shown in bold.

Dynamic instance	Effective Address	Data Value	Correct AVD	AVD valid?	Predicted AVD and (value) **reset on NULL**	Predicted AVD and (value) **no reset on NULL**
1	A	A+k	-k	valid	no prediction	no prediction
2	A+k	A+2k	-k	valid	no prediction	no prediction
3	A+2k	A+3k	-k	valid	-k (A+3k)	-k (A+3k)
4	A+3k	0 (NULL)	A+3k	not valid	-k (A+4k)	-k (A+4k)
5	A+4k	0 (NULL)	A+4k	not valid	no prediction	-k (A+5k)
6	A+5k	A+6k	-k	valid	no prediction	-k (A+6k)
7	A+6k	0 (NULL)	A+6k	not valid	no prediction	-k (A+7k)
8	A+7k	0 (NULL)	A+7k	not valid	no prediction	-k (A+8k)
9	A+8k	A+9k	-k	valid	no prediction	-k (A+9k)
10	A+9k	A+10k	-k	valid	no prediction	-k (A+10k)
11	A+10k	0 (NULL)	A+10k	not valid	-k (A+11k)	-k (A+11k)
12	A+11k	0 (NULL)	A+11k	not valid	no prediction	-k (A+12k)
13	A+12k	A+13k	-k	valid	no prediction	-k (A+13k)
14	A+13k	0 (NULL)	A+13k	not valid	no prediction	-k (A+14k)
15	A+14k	0 (NULL)	A+14k	not valid	no prediction	-k (A+15k)

Table 6.4: Execution history of Load 1 in the `treeadd` program (see Figure 6.3) for the binary tree shown in Figure 6.20.

For benchmarks similar to `treeadd` where short regular traversals frequently terminated by NULL pointer loads are common, not updating the AVD predictor on a NULL data value would be useful. NULL-value optimization requires that a NULL data value be detected by the predictor. Thus the update logic of the AVD predictor needs to be augmented with a simple comparator to zero (zero checker).

Figure 6.21 shows the impact of using NULL-value optimization on the execution time of the evaluated benchmarks. NULL-value optimization significantly improves the execution time of `treeadd` (by 41.8% versus the 17.6% improvement when confidence is reset on NULL values) and does not significantly impact the performance of other benchmarks. On average, it increases the execution time improvement of a 16-entry AVD predictor from 12.1% to 14.3%, mainly due to the improvement in `treeadd`.

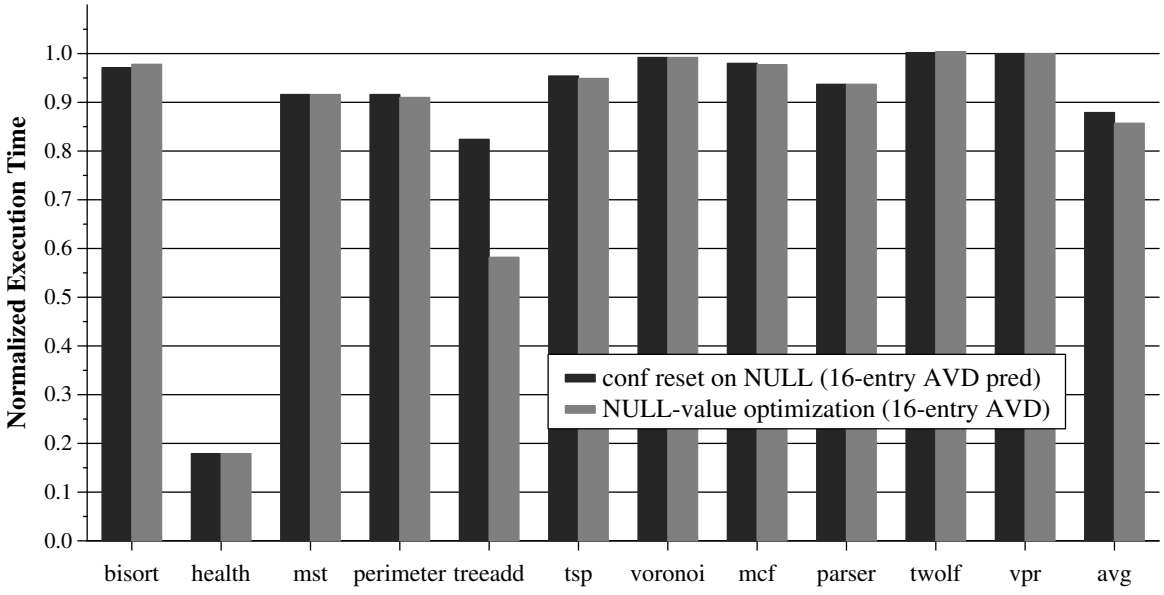


Figure 6.21: AVD prediction performance with and without NULL-value optimization.

To provide insight into the performance improvement in `treeadd`, Figures 6.22 and 6.23 show the coverage and accuracy of AVD predictions for L2-miss address loads.

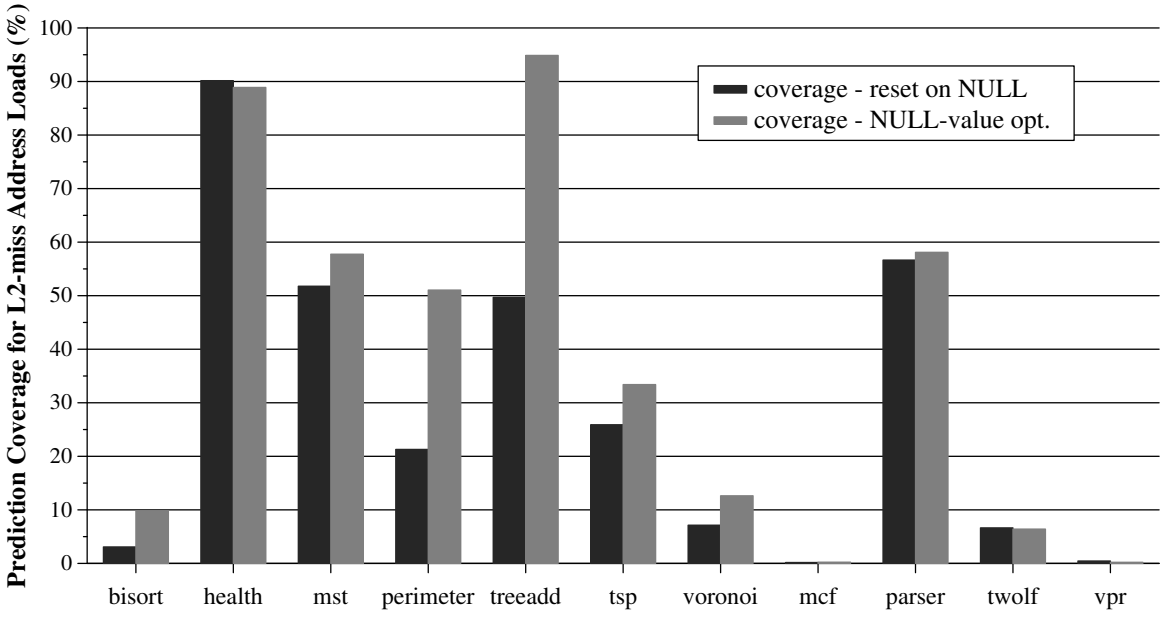


Figure 6.22: Effect of NULL-value optimization on AVD prediction coverage.

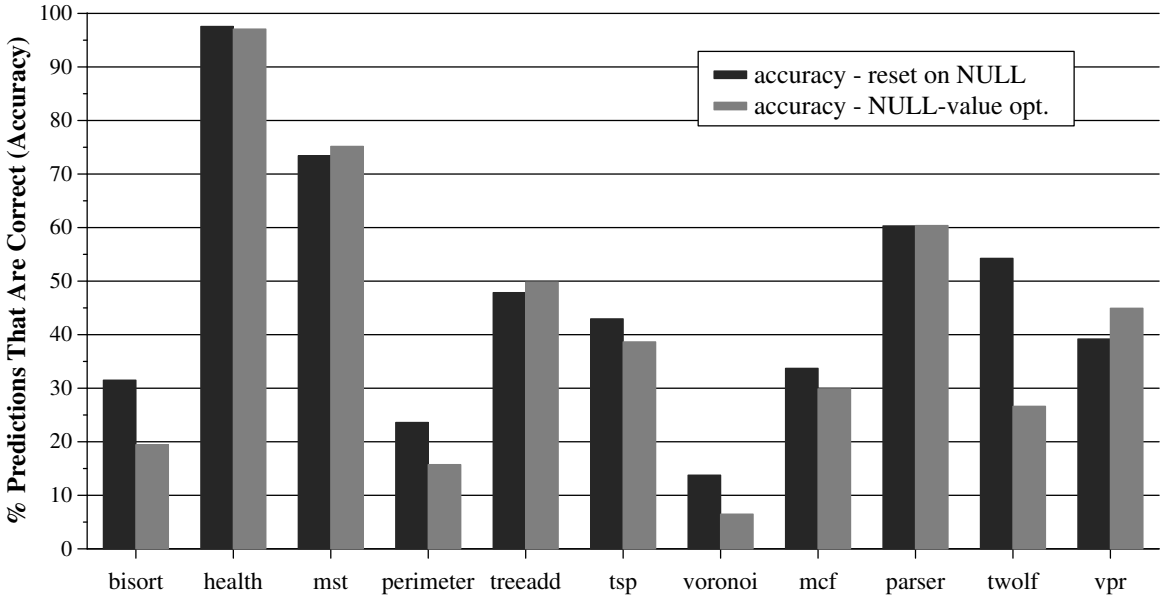


Figure 6.23: Effect of NULL-value optimization on AVD prediction accuracy.

Not updating the AVD predictor on NULL values increases the coverage of the predictor from 50% to 95% in `treeadd` while also slightly increasing its accuracy. For most other benchmarks, the AVD prediction coverage also increases with the NULL-value optimization, however the AVD prediction accuracy decreases. Therefore, the proposed NULL-value optimization does not provide significant performance benefit in most benchmarks.[13]

6.7.2 Optimizing the Source Code to Take Advantage of AVD Prediction

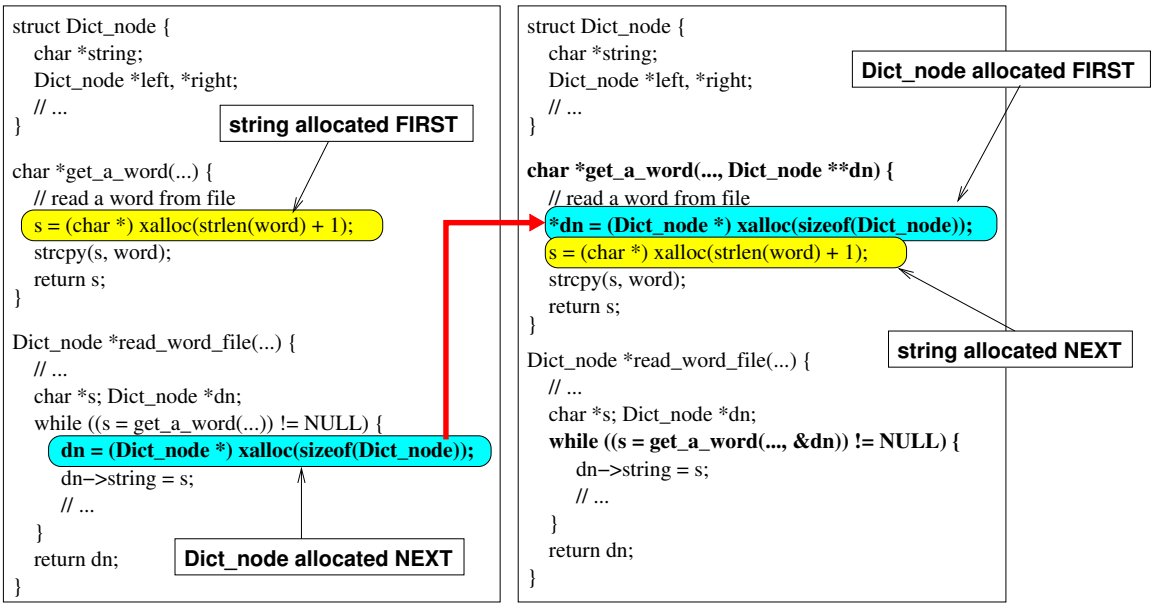
As evident from the code examples shown in Section 6.3, the existence of stable AVDs depends highly on the existence of regular memory allocation patterns arising from the way programs are written. We demonstrate how increasing the regularity in the allocation patterns of linked data structures -by modifying the application source code- increases the effectiveness of AVD prediction on a runahead processor. To do so, we use the source code example from the `parser` benchmark that was explained in Section 6.3.2 and Figure 6.4.[14]

In the `parser` benchmark, stable AVDs for Load 1 in Figure 6.4 occur because the distance in memory between a `string` and its associated `Dict_node` is constant for many nodes in the dictionary. As explained in Section 6.3.2, the distance in memory between a `string` and its associated `Dict_node` depends on the size of the `string` because the `parser` benchmark allocates memory space for `string` first and `Dict_node`

[13] In some benchmarks, encountering a NULL pointer actually coincides with the end of a stable AVD pattern. Not updating the AVD predictor on NULL values in such cases increases coverage but reduces accuracy.

[14] Note that the purpose of this section is to provide insights into how simple code optimizations can help increase the effectiveness of AVD prediction. This section is not meant to be an exhaustive treatment of all possible code optimizations for AVD prediction. We believe program, compiler, and memory allocator optimizations that can increase the occurrence of stable AVDs in applications is a large and exciting area for future research.

next. If the allocation order for these two structures is reversed (i.e., if space for `Dict_node` is allocated first and `string` next), the distance in memory between `string` and `Dict_node` would no longer be dependent on the size of the `string`, but it would be dependent on the size of `Dict_node`. Since the size of the data structure `Dict_node` is constant, the distance between `string` and `Dict_node` would always be constant. Such an optimization in the allocation order would therefore increase the stability of the AVDs of Load 1. Figure 6.24b shows the modified source code that allocates memory space for `Dict_node` first and `string` next. Note that this optimization requires only three lines to be modified in the original source code of the `parser` benchmark.



(a) Base source code that allocates the nodes of the dictionary (binary tree) and the strings

(b) Modified source code (modified lines are in bold)

Figure 6.24: Source code optimization performed in the `parser` benchmark to increase the effectiveness of AVD prediction.

Figure 6.25 shows the execution time of the baseline `parser` binary and the modified `parser` binary on a runahead processor with an without AVD prediction support. The performance of the baseline and modified binaries are the same on the runahead processor that does not implement AVD prediction, indicating that the code modifications shown in Figure 6.24 does not significantly change the performance of `parser` on the baseline runahead processor. However, when run on a runahead processor with AVD prediction, the modified binary outperforms the base binary by 4.4%. Hence, this very simple source code optimization significantly increases the effectiveness of AVD prediction by taking advantage of the way AVD prediction works.

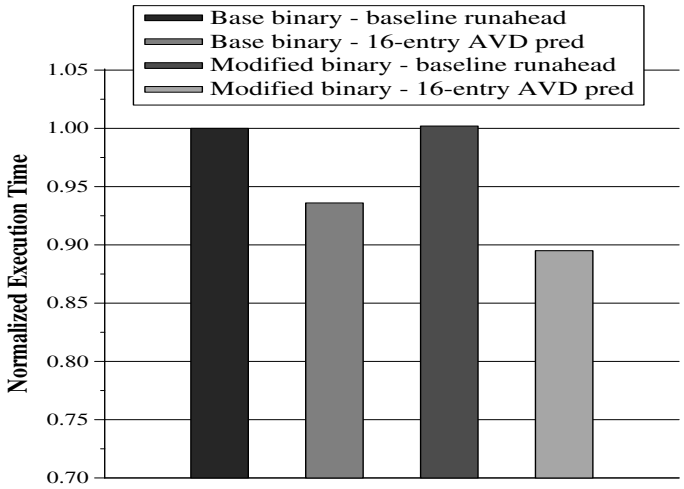


Figure 6.25: Effect of source code optimization on AVD prediction performance in the `parser` benchmark.

Figure 6.26 shows the AVD prediction coverage and accuracy for L2-miss address loads on the baseline binary and the modified binary. The described source code optimization increases the accuracy of AVD prediction from 58% to 83%. Since the modified binary has more regularity in its memory allocation patterns, the resulting AVDs for Load 1 are more stable than in the baseline binary. Hence the increase in AVD prediction accuracy and performance.

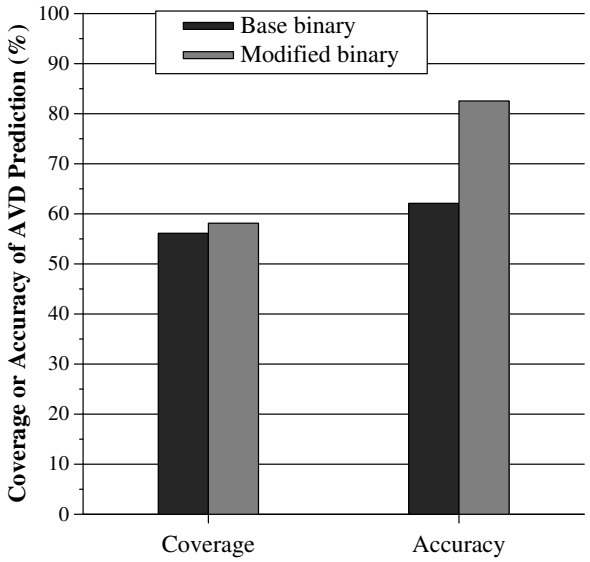


Figure 6.26: Effect of source code optimization on AVD prediction coverage and accuracy in the `parser` benchmark.

6.8 Interaction of AVD Prediction with Other Techniques

A runahead processor will likely incorporate other techniques that interact with AVD prediction, such as techniques for efficient runahead processing and stream-based hardware data prefetching. Some of the benefits provided by these mechanisms can be orthogonal to the benefits provided by AVD prediction, some not. This section analyzes the interaction of techniques for efficient runahead processing and stream-based hardware data prefetching with AVD prediction.

6.8.1 Interaction of AVD Prediction with Efficiency Techniques for Runahead Execution

Section 5.2 proposed several techniques to increase the efficiency of a runahead processor. The proposed efficiency techniques improve runahead efficiency by eliminating *short*, *overlapping*, and otherwise *useless* runahead periods without significantly reducing

the performance improvement provided by runahead execution. In essence, these techniques predict whether or not a runahead period is going to be useful (i.e., will generate a useful L2 cache miss). If the runahead period is predicted to be useless, entry into runahead mode is disabled.

In contrast, AVD prediction improves the efficiency of a runahead processor by increasing the usefulness of runahead periods (either by turning a useless runahead period into a useful one or by increasing the usefulness of an already useful runahead period). Since AVD prediction and runahead efficiency techniques improve runahead efficiency in different ways, we would like to combine these two approaches and achieve even further improvements in runahead efficiency.

This section evaluates the runahead efficiency techniques proposed in Section 5.2 alone and in conjunction with AVD prediction. Table 6.5 lists the evaluated efficiency techniques and the threshold values used in the implementation.

Short period elimination	Processor does not enter runahead on an L2 miss that has been in flight for more than T=400 cycles.
Overlapping period elimination	Not implemented. Overlapping periods were useful for performance in the benchmark set examined.
Useless period elimination	1. 64-entry, 4-way RCST
	2. Exit runahead mode if 75% of the executed loads are INV after 50 cycles in runahead mode
	3. Sampling: If the last N=100 runahead periods caused less than T=5 L2 cache misses, do not enter runahead mode for the next M=1000 L2 cache misses

Table 6.5: Runahead efficiency techniques evaluated with AVD prediction.

Figures 6.27 and 6.28 show respectively the normalized execution time and the normalized number of executed instructions when AVD prediction and efficiency techniques are utilized individually and together. We assume that NULL-value optimization is employed in the AVD predictor. In general, efficiency techniques are very effective at reducing the number of executed instructions. However, they also result in a slight performance loss. On average, using the efficiency techniques results in a 30% reduction in executed instructions accompanied with a 2.5% increase in execution time on the baseline runahead processor.

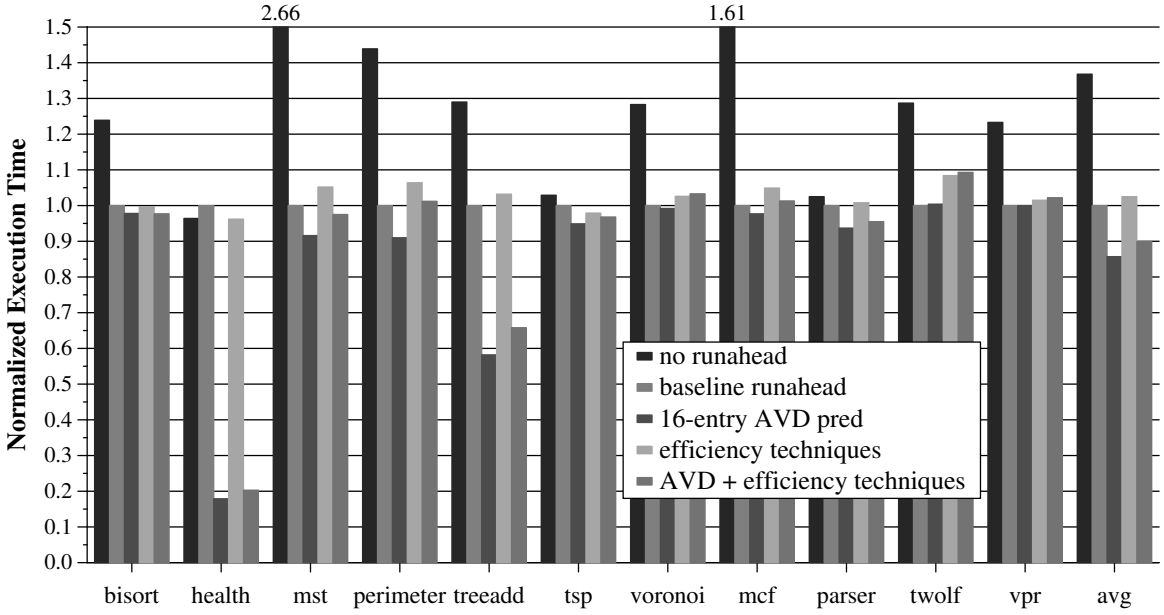


Figure 6.27: Normalized execution time when AVD prediction and runahead efficiency techniques are used individually and together.

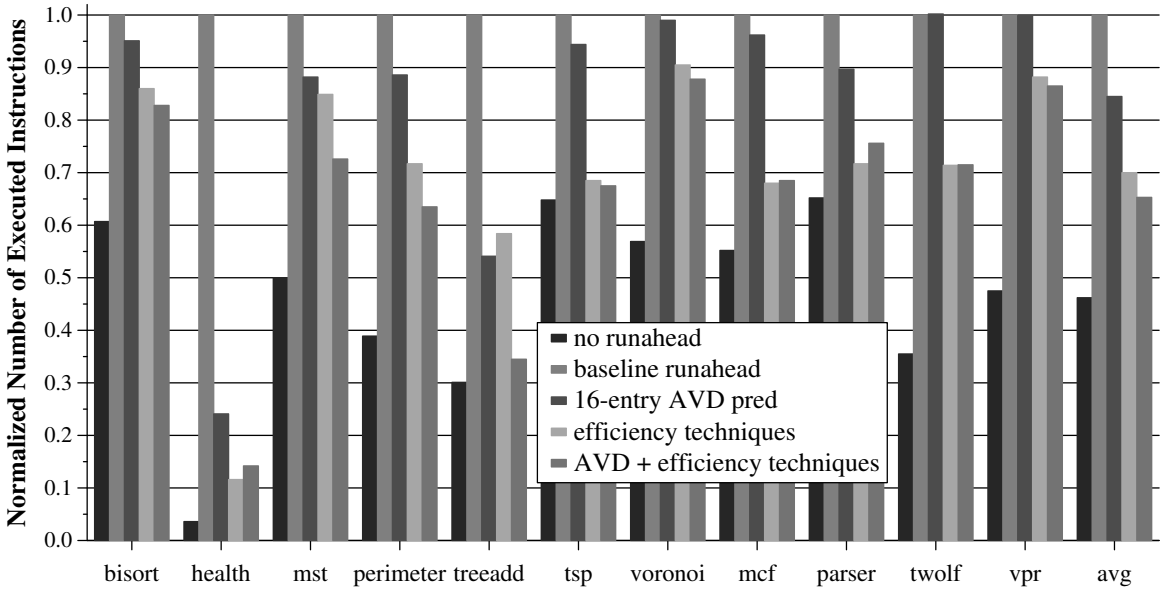


Figure 6.28: Normalized number of executed instructions when AVD prediction and runahead efficiency techniques are used individually and together.

Compared to the efficiency techniques, AVD prediction is less effective in reducing the number of executed instructions. However, AVD prediction *increases* the baseline runahead performance while also reducing the executed instructions. On average, using a 16-entry AVD predictor results in a 15.5% reduction in executed instructions accompanied with a 14.3% reduction in execution time.

Using AVD prediction in conjunction with the previously-proposed efficiency techniques further improves efficiency by *both* reducing the number of instructions *and* at the same time increasing performance. When AVD prediction and efficiency techniques are used together in the baseline runahead processor, a 35.3% reduction in executed instructions is achieved accompanied with a 10.1% decrease in execution time. Hence, AVD prediction and the previously-proposed efficiency techniques are complementary to each other and they interact positively.

Figure 6.29 shows the normalized number of runahead periods using AVD prediction and efficiency techniques. Efficiency techniques are more effective in eliminating useless runahead periods than AVD prediction. Efficiency techniques alone reduce the number of runahead periods by 53% on average. Combining AVD prediction and efficiency techniques eliminates 57% of all runahead periods and the usefulness of already-useful runahead periods also increases.

We conclude that using both AVD prediction and efficiency techniques together provides a better efficiency-performance trade-off than using either of the mechanisms alone. Therefore, an efficient runahead processor should incorporate both AVD prediction and runahead efficiency techniques.

6.8.2 Interaction of AVD Prediction with Stream-based Prefetching

Stream-based prefetching [56] is a technique that identifies regular streaming patterns in the memory requests generated by a program. Once a streaming pattern is identi-

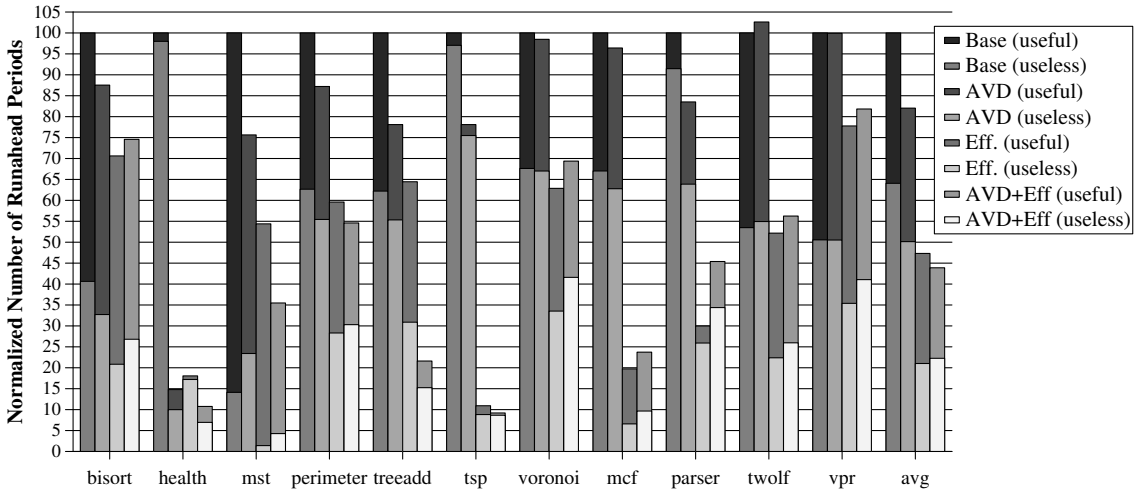


Figure 6.29: Normalized number of useful and useless runahead periods when AVD prediction and runahead efficiency techniques are used individually and together.

fied, the stream prefetcher generates speculative memory requests for later addresses in the identified stream. We compare the performance benefits and bandwidth requirements of an AVD predictor and an aggressive state-of-the-art stream-based prefetcher (described in Section 4.2.3) along with a combination of both techniques. The experiments in this section assume that the AVD predictor implements the NULL-value optimization described in Section 6.7.1.

Figure 6.30 shows the execution time improvement when AVD prediction and stream prefetching are employed individually and together on the baseline runahead processor. Figures 6.31 and 6.32 respectively show the increase in the number of L2 accesses and main memory accesses when AVD prediction and stream prefetching are employed individually and together. On average, the stream prefetcher with a prefetch distance of 32 improves the average execution time of the evaluated benchmarks by 16.5% (18.1% when `health` is excluded) while increasing the number of L2 accesses by 33.1% and main memory accesses by 14.9%. A prefetch distance of 8 provides an average performance

improvement of 13.4% (14.8% excluding `health`) and results in a 25% increase in L2 accesses and a 12.2% increase in memory accesses. In contrast, a 16-entry AVD predictor improves the average execution time of the evaluated benchmarks by 14.3% (7.5% excluding `health`) while increasing the number of L2 accesses by only 5.1% and main memory accesses by only 3.2%. Hence, AVD prediction is much less bandwidth-intensive than stream prefetching, but it does not provide as much performance improvement.

Using AVD prediction and stream prefetching together on a runahead processor improves the execution time by more than either of the two techniques does alone. This shows that the two techniques are in part complementary. Using a 16-entry AVD predictor and a stream prefetcher with a prefetch distance of 32 together improves the average execution time by 24.9% (19.5% excluding `health`) while increasing the L2 accesses by 35.3% and main memory accesses by 19.5%.

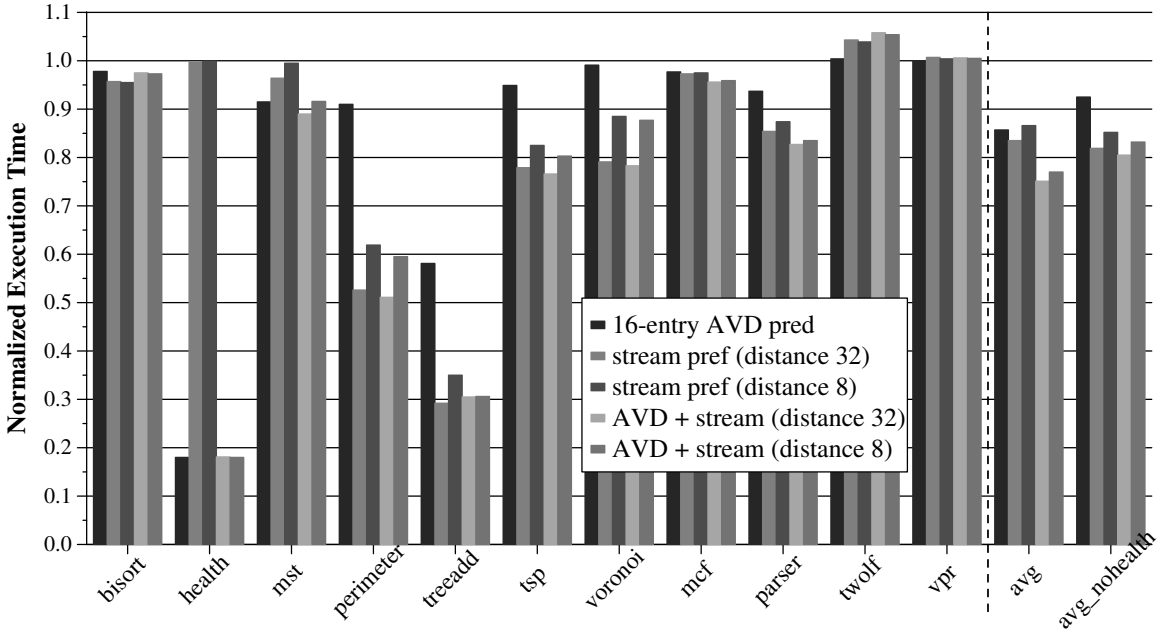


Figure 6.30: Performance comparison of AVD prediction, stream prefetching and AVD prediction combined with stream prefetching.

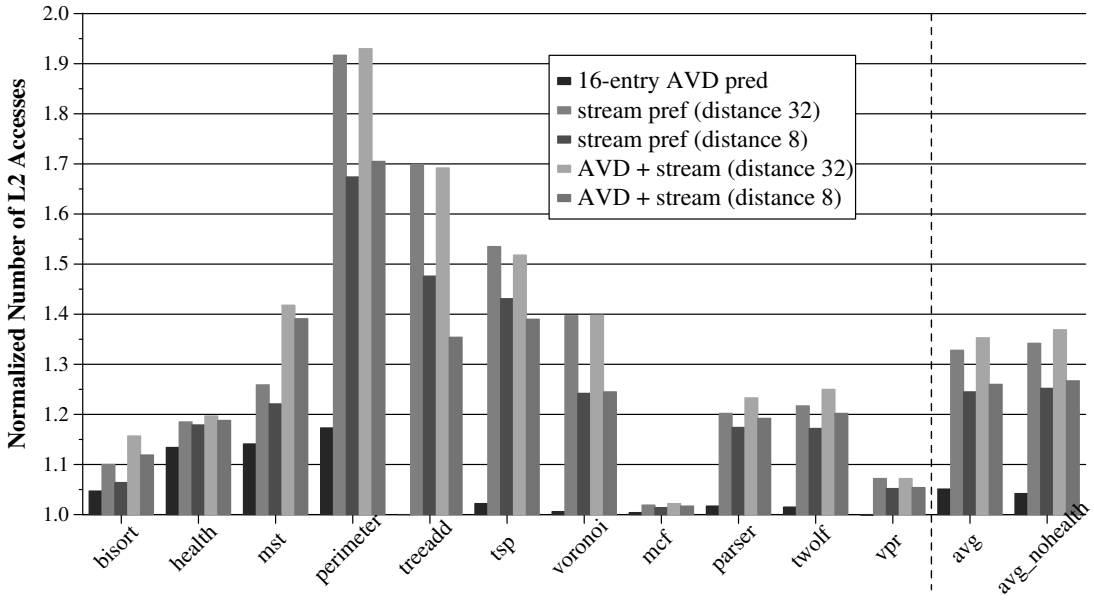


Figure 6.31: Increase in L2 accesses due to AVD prediction, stream prefetching and AVD prediction combined with stream prefetching.

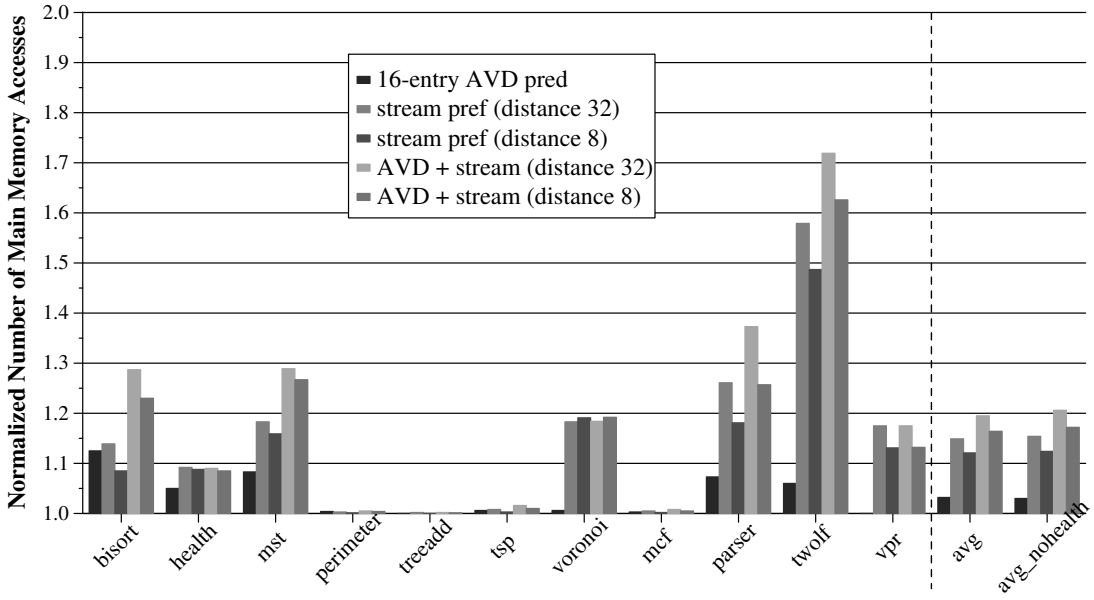


Figure 6.32: Increase in memory accesses due to AVD prediction, stream prefetching and AVD prediction combined with stream prefetching.

In general, AVD prediction is limited to prefetching the addresses of dependent load instructions whereas a stream prefetcher can prefetch addresses generated by both dependent and independent load instructions. Therefore, a stream prefetcher can capture a broader range of address patterns that are of a streaming nature. A traversal address load with a stable AVD (in this case also a regular stride) results in a streaming memory access pattern. Hence, similarly to an AVD predictor, a stream prefetcher can prefetch the addresses generated by a traversal address load with a constant AVD.

In contrast to an AVD predictor, a stream prefetcher can capture the addresses generated by a leaf address load with a stable AVD and its dependent instructions *only if* those addresses form a streaming access pattern or are part of a streaming access pattern. An AVD predictor is therefore more effective in predicting the addresses dependent on leaf address loads with stable AVDs. For this very reason, AVD prediction significantly improves the performance of two benchmarks, `health` and `mst`, for which the stream prefetcher is ineffective.

6.9 Summary and Conclusions

Even though runahead execution is effective at parallelizing independent L2 cache misses, it is unable to parallelize dependent cache misses. To overcome this limitation of runahead execution, this chapter developed a new, area-efficient technique, *address-value delta (AVD) prediction*, that is used to predict the values loaded by address load instructions by predicting the AVD (the arithmetic difference between the effective address and the data value) of the instruction. Our analysis showed that stable AVDs exist due to the regular memory allocation patterns in programs that heavily utilize linked data structures.

The proposed AVD prediction mechanism requires neither significant hardware cost or complexity nor hardware support for state recovery. Our experimental results showed

that a simple AVD predictor can significantly improve both the performance and efficiency of a runahead execution processor. AVD prediction also interacts positively with two previously-proposed mechanisms, efficiency techniques for runahead execution and stream-based data prefetching.

The performance benefit of AVD prediction is not limited to that quantified in this chapter. As we have shown, optimizing the source code by increasing the regularity in the memory allocation patterns increases the occurrence of stable AVDs and the performance of AVD prediction. Therefore, we expect the benefits of AVD prediction to increase as programmers write software and compilers optimize the source code by taking into account how AVD prediction works in the underlying microarchitecture.

Chapter 7

Conclusions and Future Research Directions

7.1 Conclusions

Long memory latencies are an important problem limiting the performance of processors. As processor designers push for higher clock frequencies, the main memory latencies will continue to increase in terms of processor clock cycles. An out-of-order execution processor already requires an unreasonably large instruction window to tolerate these latencies, as Chapter 1 showed. Unfortunately, building large, complex, slow, and power-hungry structures associated with a large instruction window is a difficult problem, which is still not solved despite extensive ongoing research.

This dissertation proposed and evaluated the runahead execution paradigm as an alternative to building large instruction windows to tolerate long memory latencies. Runahead execution provides the memory-level parallelism benefits achieved with a large instruction window, without requiring the implementation of structures that are needed to support a large number of in-flight instructions. In fact, the implementation of runahead execution adds very little hardware cost and complexity to an existing out-of-order execution processor, as we showed in Chapter 2.

Chapter 4 presented an evaluation of runahead execution and compared the performance of runahead execution to that of large instruction windows. For a 500-cycle memory latency, implementing runahead execution on a processor with a 128-entry instruction window achieves the same performance as a conventional out-of-order processor with a 384-entry instruction window. Hence, runahead execution can provide the benefits

of a processor with three times the instruction window size of a current processor for a 500-cycle memory latency. If the memory latency is 1000 cycles, implementing runahead execution on a processor with a 128-entry instruction window achieves the same performance as a conventional processor with a 1024-entry instruction window. Thus, runahead execution provides the benefits of a processor with eight times the instruction window size of a current processor when the memory latency is doubled.

As runahead execution's performance benefit significantly increases with increased memory latencies, runahead execution will become more effective on future processors which will face longer memory latencies. Moreover, runahead execution's performance benefit increases with an improved instruction fetch unit. As computer architects continue to improve the performance of instruction fetch units, the performance improvement provided by runahead execution will increase.

The results presented in this dissertation were obtained on an aggressive processor model with a very effective stream-based hardware data prefetcher. Chapter 4 showed that runahead execution is a more effective prefetching mechanism than a state-of-the-art hardware data prefetcher. Runahead execution and the stream prefetcher interact positively and they are complementary to each other. Therefore, future processors should employ both mechanisms to tolerate long memory latencies.

Chapter 4 also evaluated the effectiveness of runahead execution on in-order execution processors that do not support dynamic instruction scheduling. Runahead execution significantly improves the latency tolerance of an in-order processor. An in-order processor with runahead execution can actually reach and surpass the performance of a conventional out-of-order processor with a 128-entry instruction window, when the memory latency is longer than 1500 cycles. However, the best performance is obtained when runahead execution is used on an out-of-order processor. An out-of-order processor with runahead execution always provides significantly better performance than both an out-of-order pro-

cessor without runahead execution and an in-order processor with runahead execution, for a large variety of memory latencies ranging from 100 to 1900 cycles. Therefore, future processors should employ runahead execution in conjunction with out-of-order execution to provide the highest performance.

This dissertation proposed solutions to a major disadvantage of the runahead execution paradigm. Inefficiency of runahead execution was shown to be a problem that would result in increased dynamic energy consumption in a processor that implements runahead execution because runahead execution significantly increases the number of speculatively executed instructions. Chapter 5 analyzed the causes of inefficiency in runahead execution and identified three major causes of inefficiency: *short*, *overlapping*, and *useless* runahead periods. Simple and implementable techniques were proposed to eliminate these causes. The evaluations showed that using the proposed techniques for improving efficiency reduces the extra instructions due to runahead execution from 26.5% to 6.2% while only slightly reducing runahead execution's IPC improvement from 22.6% to 22.1%. Thus, the performance benefit of runahead execution can be achieved without significantly increasing the number of executed instructions.

Reuse of the results of the instructions that are executed in runahead mode was also evaluated as a technique that could potentially improve the performance and efficiency of runahead execution. However, evaluations and analyses presented in Chapter 5 found that augmenting runahead execution with an aggressive reuse mechanism does not significantly improve performance while it likely adds significant hardware cost and complexity. Therefore, runahead execution should be employed as solely a prefetching mechanism without result reuse.

Finally, this dissertation proposed a solution to a major limitation of the runahead execution paradigm. Runahead execution is unable to parallelize long-latency cache misses that are due to dependent load instructions. To overcome this limitation, Chapter 6 pro-

posed and evaluated *address-value delta prediction*, a new technique that is used to predict the value loaded by an address load instruction by predicting the AVD (the arithmetic difference between the effective address and the data value) of the instruction. Our evaluations showed that a simple, 16-entry AVD predictor improves the execution time of a set of pointer-intensive applications by 14.3% on a runahead execution processor. AVD prediction also reduces the number of instructions executed in a runahead processor by 15.5%. Chapter 6 also described hardware and software optimizations that significantly improve both the performance and the efficiency benefits of AVD prediction. Hence, with the assistance provided by a simple AVD predictor, runahead execution can parallelize dependent cache misses and therefore it can improve the performance of pointer-intensive programs that heavily utilize large linked data structures.

Based on the evaluations presented in this dissertation, we believe that the runahead execution paradigm has three major advantages:

- Runahead execution provides the benefits obtained with a much larger instruction window, but it does not require the implementation of large, complex, slow, and power-hungry structures in the processor core to support a large number of in-flight instructions. Instead, it adds little hardware cost and complexity to an existing processor because it utilizes the already-existing processor structures to improve memory latency tolerance.

- With the simple efficiency techniques described in Chapter 5, runahead execution requires the execution of only a small number of extra instructions to provide significant performance improvements.

- With the simple AVD prediction mechanism described in Chapter 6, runahead execution provides the ability to parallelize long-latency cache misses due to dependent, as well as independent load instructions. This makes runahead execution a general

memory latency tolerance paradigm for parallelizing cache misses in a wide variety of applications.

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

7.2 Future Research Directions

Future research in runahead execution can proceed in several different directions. This dissertation provides a complete initial proposal for a general efficient runahead execution paradigm. The mechanisms to improve the efficiency and to overcome the limitations of runahead execution in ways orthogonal to those proposed in this dissertation can provide significant improvements in the performance and the efficiency of runahead execution.

7.2.1 Research Directions in Improving Runahead Efficiency

Orthogonal approaches can be developed to solve the inefficiency problem in runahead execution. We believe this is an important research area for runahead execution in particular and for precomputation-based memory latency tolerance techniques in general. Especially 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 (or other more effective techniques for handling branch instructions) 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, as we showed in Section 6, runahead execution is inefficient, and sometimes ineffective, for pointer-chasing workloads where dependent load instructions are common. AVD prediction provides a solution to this problem. However, there are some dependent cache misses that cannot be captured by AVD prediction because they occur in data structures with irregular memory allocation patterns. We believe orthogonal techniques that enable the parallelization of such dependent cache misses is another promising area of future research in runahead execution.

Future research can also focus on refining the methods for increasing the usefulness of runahead execution periods to improve both the performance and the efficiency of a runahead execution processor. 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. For example, the compiler can statically predict which instructions are likely to lead to the generation of long-latency cache misses and mark the dependence chain leading to those instructions with hint bits. The microarchitecture, at run time, can execute only those instructions in runahead mode. Furthermore, the compiler may be able to increase the usefulness of runahead periods by trying to arrange code and data structures such that independent L2 cache misses are clustered close together during program execution. Such compilation techniques that improve the efficiency and effectiveness of runahead execution perhaps with support from both the programmer and the microarchitecture are also promising research topics.

7.2.2 Research Directions in Improving AVD Prediction

As we showed in Chapter 6, the effectiveness of AVD prediction is highly dependent on the memory allocation patterns in programs. Optimizing the memory allocator, the program structures, and the algorithms used in programs for AVD prediction can increase the occurrence of stable AVDs. Hence, software (i.e., programmer, compiler, memory allocator, and garbage collector) support can improve the effectiveness of a mechanism that exploits address-value deltas.

Since AVD prediction exploits high-level programming constructs, we believe cooperative mechanisms that combine the efforts of the programmer, the compiler, and the microarchitecture through efficient architectural abstractions can provide significant benefits and are worthy of future research. Chapter 6 provided a simple example of how the performance improvement provided by AVD prediction can be increased by the programmer, if the programmer allocates memory for the data structures in the parser benchmark by taking into account the properties of AVD prediction. Future research can explore software engineering techniques, programming language abstractions, and architectural abstractions for helping programmers to write programs where the memory allocation patterns are regular and convey their knowledge about the data structure layouts to the underlying microarchitecture. The design of compilers and memory allocators that optimize the memory layout of linked data structures to increase the occurrence of stable AVDs can increase the performance of a runahead processor that employs AVD prediction. Dynamic techniques that can cost-efficiently re-organize the layout of data structures at run-time may also be useful in increasing the occurrence of stable AVDs as long as they have small performance overhead.

Bibliography

[1] A.-R. Adl-Tabatabai, R. L. Hudson, M. J. Serrano, and S. Subramoney. Prefetch injection based on hardware monitoring and object metadata. In *Proceedings of the ACM SIGPLAN'04 Conference on Programming Language Design and Implementation (PLDI)*, pages 267–276, 2004.

[2] H. Akkary and M. A. Driscoll. A dynamic multithreading processor. In *Proceedings of the 31st International Symposium on Microarchitecture (MICRO-31)*, pages 226–236, 1998.

[3] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *Proceedings of the 36th International Symposium on Microarchitecture (MICRO-36)*, pages 423–434, 2003.

[4] G. M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*, pages 483–485, 1967.

[5] J. Baer and T. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of Supercomputing '91*, pages 178–186, 1991.

[6] R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi. Dynamically allocating processor resources between nearby and distant ILP. In *Proceedings of the 28th International Symposium on Computer Architecture (ISCA-28)*, pages 26–37, 2001.

[7] R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi. Reducing the complexity of the register file in dynamic superscalar processors. In *Proceedings of the 34th International Symposium on Microarchitecture (MICRO-34)*, pages 237–248, 2001.

[8] BAPCo Benchmarks. *BAPCo*. http://www.bapco.com/.

[9] R. D. Barnes, E. M. Nystrom, J. W. Sias, S. J. Patel, N. Navarro, and W. W. Hwu. Beating in-order stalls with flea-flicker two-pass pipelining. In *Proceedings of the 36th International Symposium on Microarchitecture (MICRO-36)*, pages 387–398, 2003.

[10] R. D. Barnes, S. Ryoo, and W. W. Hwu. Flea-flicker multipass pipelining: An alternative to the high-power out-of-order offense. In *Proceedings of the 38th International Symposium on Microarchitecture (MICRO-38)*, pages 319–330, 2005.

[11] M. Bekerman, S. Jourdan, R. Ronen, G. Kirshenboim, L. Rappoport, A. Yoaz, and U. Weiser. Correlated load-address predictors. In *Proceedings of the 26th International Symposium on Computer Architecture (ISCA-26)*, pages 54–63, 1999.

[12] D. Boggs, A. Baktha, J. Hawkins, D. T. Marr, J. A. Miller, P. Roussel, R. Singhal, B. Toll, and K. Venkatraman. The microarchitecture of the Intel Pentium 4 processor on 90nm technology. *Intel Technology Journal*, 8(1), Feb. 2004.

[13] E. Borch, E. Tune, S. Manne, and J. S. Emer. Loose loops sink chips. In *Proceedings of the 8th International Symposium on High Performance Computer Architecture (HPCA-8)*, pages 299–310, 2002.

[14] J. A. Butts and G. S. Sohi. Use-based register caching with decoupled indexing. In *Proceedings of the 31st International Symposium on Computer Architecture (ISCA-31)*, pages 302–313, 2004.

[15] H. W. Cain and M. H. Lipasti. Memory ordering: A value-based approach. In *Proceedings of the 31st International Symposium on Computer Architecture (ISCA-31)*, pages 90–101, 2004.

[16] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 40–52, 1991.

[17] L. Ceze, K. Strauss, J. Tuck, J. Renau, and J. Torrellas. CAVA: Hiding L2 misses with checkpoint-assisted value prediction. *Computer Architecture Letters*, 3, Dec. 2004.

[18] P.-Y. Chang, E. Hao, and Y. N. Patt. Predicting indirect jumps using a target cache. In *Proceedings of the 24th International Symposium on Computer Architecture (ISCA-24)*, pages 274–283, 1997.

[19] R. S. Chappell. *Simultaneous Subordinate Microthreading*. PhD thesis, University of Michigan, Ann Arbor, 2004.

[20] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt. Simultaneous subordinate microthreading (SSMT). In *Proceedings of the 26th International Symposium on Computer Architecture (ISCA-26)*, pages 186–195, 1999.

[21] M. Charney. *Correlation-based Hardware Prefetching*. PhD thesis, Cornell University, Aug. 1995.

[22] T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of the ACM SIGPLAN'02 Conference on Programming Language Design and Implementation (PLDI)*, pages 199–209, 2002.

[23] Y. Chou, B. Fahs, and S. Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. In *Proceedings of the 31st International Symposium on Computer Architecture (ISCA-31)*, pages 76–87, 2004.

[24] Y. Chou, L. Spracklen, and S. G. Abraham. Store memory-level parallelism optimizations for commercial applications. In *Proceedings of the 38th International Symposium on Microarchitecture (MICRO-38)*, pages 183–194, 2005.

[25] G. Z. Chrysos and J. S. Emer. Memory dependence prediction using store sets. In *Proceedings of the 25th International Symposium on Computer Architecture (ISCA-25)*, pages 142–153, 1998.

[26] J. D. Collins, S. Sair, B. Calder, and D. M. Tullsen. Pointer cache assisted prefetching. In *Proceedings of the 35th International Symposium on Microarchitecture (MICRO-35)*, pages 62–73, 2002.

[27] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen. Dynamic speculative precomputation. In *Proceedings of the 34th International Symposium on Microarchitecture (MICRO-34)*, pages 306–317, 2001.

[28] R. Colwell. Maintaining a leading position. *IEEE Computer*, 31(1):45–48, Jan. 1998.

[29] R. Cooksey, S. Jourdan, and D. Grunwald. A stateless, content-directed data prefetching mechanism. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, pages 279–290, 2002.

[30] A. Cristal, J. F. Martinez, J. Llosa, and M. Valero. A case for resource-conscious out-of-order processors. *Computer Architecture Letters*, Oct. 2003.

[31] A. Cristal, D. Ortega, J. Llosa, and M. Valero. Out-of-order commit processors. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture (HPCA-10)*, pages 48–59, 2004.

[32] A. Cristal, O. J. Santana, F. Cazorla, M. Galluzzi, T. Ramirez, M. Pericas, and M. Valero. Kilo-instruction processors: Overcoming the memory wall. *IEEE Micro*, 25(3):48–57, May 2005.

[33] A. Cristal, M. Valero, J.-L. Llosa, and A. Gonzalez. Large virtual robs by processor checkpointing. Technical Report UPC-DAC-2002-39, Dept. de Computadors, Universitat Politecnica de Catalunya, July 2002.

[34] J.-L. Cruz, A. Gonzalez, M. Valero, and N. N. Topham. Multiple-banked register file architectures. In *Proceedings of the 27th International Symposium on Computer Architecture (ISCA-27)*, pages 316–325, 2000.

[35] P. Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, SE-6(1):64–84, Jan. 1980.

[36] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the 1997 International Conference on Supercomputing (ICS)*, pages 68–75, 1997.

[37] J. D. Dundas. *Improving Processor Performance by Dynamically Pre-Processing the Instruction Stream*. PhD thesis, University of Michigan, 1998.

[38] R. J. Eickemeyer and S. Vassiliadis. A load-instruction unit for pipelined processors. *IBM Journal of Research and Development*, 37:547–564, 1993.

[39] M. Franklin and G. S. Sohi. The expandable split window paradigm for exploiting fine-grain parallelism. In *Proceedings of the 19th International Symposium on Computer Architecture (ISCA-19)*, pages 58–67, 1992.

[40] I. Ganusov and M. Burtscher. Future execution: A hardware prefetching technique for chip multiprocessors. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT-14)*, pages 350–360, 2005.

[41] J. D. Gindele. Buffer block prefetching method. *IBM Technical Disclosure Bulletin*, 20(2):696–697, July 1977.

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

[43] 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.

[44] A. Gonzalez, J. Gonzalez, and M. Valero. Virtual-physical registers. In *Proceedings of the 4th International Symposium on High Performance Computer Architecture (HPCA-4)*, pages 175–184, 1998.

[45] M. K. Gowan, L. L. Biro, and D. B. Jackson. Power considerations in the design of the Alpha 21264 microprocessor. In *Proceedings of the 35th Design Automation Conference (DAC-35)*, pages 726–731, 1998.

[46] G. Grohoski. Reining in complexity. *IEEE Computer*, 31(1):41–42, Jan. 1998.

[47] L. Gwennap. Intel's P6 uses decoupled superscalar design. *Microprocessor Report*, Feb. 1995.

[48] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, 5(1), Feb. 2001.

[49] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa. An elementary processor architecture with simultaneous instruction issuing from multiple threads. In *Proceedings of the 19th International Symposium on Computer Architecture (ISCA-19)*, pages 136–145, 1992.

[50] W. W. Hwu and Y. N. Patt. Checkpoint repair for out-of-order execution machines. In *Proceedings of the 14th International Symposium on Computer Architecture (ISCA-14)*, pages 18–26, 1987.

[51] S. Iacobovici, L. Spracklen, S. Kadambi, Y. Chou, and S. G. Abraham. Effective stream-based and execution-based data prefetching. In *Proceedings of the 18th International Conference on Supercomputing (ICS-18)*, pages 1–11, 2004.

[52] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture*, 2001.

[53] Intel Corporation. *Intel Pentium 4 Processor Optimization Reference Manual*, 2001.

[54] D. A. Jiménez and C. Lin. Dynamic branch prediction with perceptrons. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture (HPCA-7)*, pages 197–206, 2001.

[55] D. Joseph and D. Grunwald. Prefetching using Markov predictors. In *Proceedings of the 24th International Symposium on Computer Architecture (ISCA-24)*, pages 252–263, 1997.

[56] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th International Symposium on Computer Architecture (ISCA-17)*, pages 364–373, 1990.

[57] S. Jourdan, T.-H. Hsing, J. Stark, and Y. N. Patt. The effects of mispredicted-path execution on branch prediction structures. In *Proceedings of the 1996 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 58–67, 1996.

[58] R. E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, 1999.

[59] N. Kırman, M. Kırman, M. Chaudhuri, and J. F. Martínez. Checkpointed early load retirement. In *Proceedings of the 11th International Symposium on High Performance Computer Architecture (HPCA-11)*, pages 16–27, 2005.

[60] A. C. Klaiber and H. M. Levy. An architecture for software-controlled data prefetching. In *Proceedings of the 18th International Symposium on Computer Architecture (ISCA-18)*, pages 1–11, 1991.

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

[62] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th International Symposium on Computer Architecture (ISCA-8)*, pages 81–87, 1981.

[63] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A large, fast instruction window for tolerating cache misses. In *Proceedings of the 29th International Symposium on Computer Architecture (ISCA-29)*, pages 59–70, 2002.

[64] S.-J. Lee and P.-C. Yew. On some implementation issues for value prediction on wide-issue ILP processors. In *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 145–156, 2000.

[65] M. H. Lipasti, W. J. Schmidt, S. R. Kunkel, and R. R. Roediger. SPAID: Software prefetching in pointer- and call-intensive environments. In *Proceedings of the 28th International Symposium on Microarchitecture (MICRO-28)*, pages 232–236, 1995.

[66] M. H. Lipasti, C. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 226–237, 1996.

[67] C.-K. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proceedings of the 28th International Symposium on Computer Architecture (ISCA-28)*, pages 40–51, 2001.

[68] C.-K. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 222–233, 1996.

[69] J. F. Martinez, J. Renau, M. C. Huang, M. Prvulovic, and J. Torrellas. Cherry: checkpointed early resource recycling in out-of-order microprocessors. In *Proceedings of the 35th International Symposium on Microarchitecture (MICRO-35)*, pages 3–14, 2002.

[70] A. M. G. Maynard, C. M. Donnelly, and B. R. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 145–156, 1994.

[71] S. McFarling. Combining branch predictors. Technical Report TN-36, Digital Western Research Laboratory, June 1993.

[72] A. Moshovos and G. S. Sohi. Streamlining inter-operation memory communication via data dependence prediction. In *Proceedings of the 30th International Symposium on Microarchitecture (MICRO-30)*, pages 235–245, 1997.

[73] M. Moudgill, K. Pingali, and S. Vassiliadis. Register renaming and dynamic speculation: an alternative approach. In *Proceedings of the 26th International Symposium on Microarchitecture (MICRO-26)*, pages 202–213, 1993.

[74] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 62–73, 1992.

[75] O. Mutlu, H. Kim, D. N. Armstrong, and Y. N. Patt. Cache filtering techniques to reduce the negative impact of useless speculative memory references on processor performance. In *Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing*, pages 2–9, 2004.

[76] O. Mutlu, H. Kim, D. N. Armstrong, and Y. N. Patt. Understanding the effects of wrong-path memory references on processor performance. In *Proceedings of the 3rd Workshop on Memory Performance Issues*, pages 56–64, 2004.

[77] O. Mutlu, H. Kim, D. N. Armstrong, and Y. N. Patt. An analysis of the performance impact of wrong-path memory references on out-of-order and runahead execution processors. *IEEE Transactions on Computers*, 54(12):1556–1571, Dec. 2005.

[78] O. Mutlu, H. Kim, D. N. Armstrong, and Y. N. Patt. Using the first-level caches as filters to reduce the pollution caused by speculative memory references. *International Journal of Parallel Programming*, 33(5):529–559, October 2005.

[79] O. Mutlu, H. Kim, and Y. N. Patt. Address-value delta (AVD) prediction: Increasing the effectiveness of runahead execution by exploiting regular memory allocation patterns. In *Proceedings of the 38th International Symposium on Microarchitecture (MICRO-38)*, pages 233–244, 2005.

[80] O. Mutlu, H. Kim, and Y. N. Patt. Techniques for efficient processing in runahead execution engines. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA-32)*, pages 370–381, 2005.

[81] O. Mutlu, H. Kim, and Y. N. Patt. Efficient runahead execution: Power-efficient memory latency tolerance. *IEEE Micro Special Issue: Top Picks from Microarchitecture Conferences*, 26(1), 2006.

[82] 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.

[83] 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 *Proceedings of the 9th International Symposium on High Performance Computer Architecture (HPCA-9)*, pages 129–140, 2003.

[84] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An effective alternative to large instruction windows. *IEEE Micro Special Issue: Top Picks from Microarchitecture Conferences*, 23(6):20–25, 2003.

[85] M. D. Nemirovsky, F. Brewer, and R. C. Wood. DISC: Dynamic instruction stream computer. In *Proceedings of the 18th International Symposium on Computer Architecture (ISCA-18)*, pages 163–171, 1991.

[86] V. S. Pai and S. Adve. Code transformations to improve memory parallelism. In *Proceedings of the 32nd International Symposium on Microarchitecture (MICRO-32)*, pages 147–155, 1999.

[87] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th International Symposium on Computer Architecture (ISCA-24)*, pages 206–218, 1997.

[88] S. Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of the 21st International Symposium on Computer Architecture (ISCA-21)*, pages 24–33, 1994.

[89] G. M. Papadopoulos and K. R. Traub. Multithreading: A revisionist view of dataflow architectures. In *Proceedings of the 18th International Symposium on Computer Architecture (ISCA-18)*, pages 342–351, 1991.

[90] Y. Patt, W. Hwu, and M. Shebanow. HPS, a new microarchitecture: Rationale and introduction. In *Proceedings of the 18th International Symposium on Microarchitecture (MICRO-18)*, pages 103–107, 1985.

[91] Y. N. Patt, S. W. Melvin, W. Hwu, and M. C. Shebanow. Critical issues regarding HPS, a high performance microarchitecture. In *Proceedings of the 18th International Symposium on Microarchitecture (MICRO-18)*, pages 109–116, 1985.

[92] P. Racunas. *Reducing Load Latency Through Memory Instruction Characterization*. PhD thesis, University of Michigan, 2003.

[93] A. Rogers, M. C. Carlisle, J. Reppy, and L. Hendren. Supporting dynamic data structures on distributed memory machines. *ACM Transactions on Programming Languages and Systems*, 17(2):233–263, Mar. 1995.

[94] A. Roth, A. Moshovos, and G. S. Sohi. Dependence based prefetching for linked data structures. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pages 115–126, 1998.

[95] A. Roth and G. S. Sohi. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the 26th International Symposium on Computer Architecture (ISCA-26)*, pages 111–121, 1999.

[96] A. Roth and G. S. Sohi. Speculative data-driven multithreading. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture (HPCA-7)*, pages 37–48, 2001.

[97] Y. Sazeides and J. E. Smith. The predictability of data values. In *Proceedings of the 30th International Symposium on Microarchitecture (MICRO-30)*, pages 248–257, 1997.

[98] S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore, and S. W. Keckler. Scalable hardware memory disambiguation for high ILP processors. In *Proceedings of the 36th International Symposium on Microarchitecture (MICRO-36)*, pages 399–410, 2003.

[99] T. Sha, M. M. K. Martin, and A. Roth. Scalable store-load forwarding via store queue index prediction. In *Proceedings of the 38th International Symposium on Microarchitecture (MICRO-38)*, pages 159–170, 2005.

[100] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, pages 45–57, 2002.

[101] R. L. Sites. *Alpha Architecture Reference Manual*. Digital Press, Burlington, MA, 1992.

[102] B. J. Smith. A pipelined shared resource MIMD computer. In *Proceedings of the 1978 International Conference on Parallel Processing (ICPP)*, pages 6–8, 1978.

[103] J. E. Smith and A. R. Pleszkun. Implementation of precise interrupts in pipelined processors. In *Proceedings of the 12th International Symposium on Computer Architecture (ISCA-12)*, pages 36–44, 1985.

[104] A. Sodani and G. S. Sohi. Dynamic instruction reuse. In *Proceedings of the 24th International Symposium on Computer Architecture (ISCA-24)*, pages 194–205, 1997.

[105] Y. Solihin, J. Lee, and J. Torrellas. Using a user-level memory thread for correlation prefetching. In *Proceedings of the 29th International Symposium on Computer Architecture (ISCA-29)*, pages 171–182, 2002.

[106] SPARC International, Inc. *The SPARC Architecture Manual, Version 9*, 1994.

[107] E. Sprangle, Aug. 2002. Personal communication.

[108] E. Sprangle and D. Carmean. Increasing processor performance by implementing deeper pipelines. In *Proceedings of the 29th International Symposium on Computer Architecture (ISCA-29)*, pages 25–34, 2002.

[109] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual flow pipelines. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XI)*, pages 107–119, 2004.

[110] The Standard Performance Evaluation Corporation. *Welcome to SPEC*. http://www.specbench.org/.

[111] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream processors: Improving both performance and fault tolerance. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 257–268, 2000.

[112] J. A. Swensen and Y. N. Patt. Hierarchical registers for scientific computers. In *Proceedings of the 1988 International Conference on Supercomputing (ICS)*, pages 346–354, 1988.

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

[114] J. E. Thornton. *Design of a Computer: The Control Data 6600*. Scott, Foresman and Company Press, 1970.

[115] TimesTen, Inc. *TimesTen*. http://www.timesten.com/.

[116] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11:25–33, Jan. 1967.

[117] Transaction Processing Performance Council. *TPC Benchmarks*. http://www.tpc.org/information/benchmarks.asp.

[118] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd International Symposium on Computer Architecture (ISCA-22)*, pages 392–403, 1995.

[119] Volano Software. *The Volano Report and Benchmark Tests*. http://www.volano.com/benchmarks.html.

[120] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. In *Proceedings of the 30th International Symposium on Microarchitecture (MICRO-30)*, pages 281–290, 1997.

[121] M. V. Wilkes. Slave memories and dynamic storage allocation. *IEEE Transactions on Electronic Computers*, 14(2):270–271, 1965.

[122] M. V. Wilkes. The memory gap and the future of high performance memories. *ACM Computer Architecture News*, 29(1):2–7, Mar. 2001.

[123] Y. Wu. Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching. In *Proceedings of the ACM SIGPLAN'02 Conference on Programming Language Design and Implementation (PLDI)*, pages 210–221, 2002.

[124] W. Wulf and S. McKee. Hitting the memory wall: Implications of the obvious. *ACM Computer Architecture News*, 23(1):20–24, Mar. 1995.

[125] C.-L. Yang and A. R. Lebeck. Push vs. pull: Data movement for linked data structures. In *Proceedings of the 2000 International Conference on Supercomputing (ICS)*, pages 176–186, 2000.

[126] K. C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–41, Apr. 1996.

[127] T.-Y. Yeh and Y. N. Patt. Alternative implementations of two-level adaptive branch prediction. In *Proceedings of the 19th International Symposium on Computer Architecture (ISCA-19)*, pages 124–134, 1992.

[128] R. Yung and N. Wilhelm. Caching processor general registers. In *Proceedings of the 1995 International Conference on Computer Design (ICCD)*, pages 307–312, 1995.

[129] H. 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 (PACT-14)*, pages 231–242, 2005.

[130] H. Zhou and T. M. Conte. Enhancing memory level parallelism via recovery-free value prediction. In *Proceedings of the 17th International Conference on Supercomputing (ICS-17)*, pages 326–335, 2003.

[131] Ziff Davis Media benchmarks from eTesting Labs. *Ziff Davis*. http://www.etestinglabs.com/benchmarks/default.asp.

[132] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *Proceedings of the 28th International Symposium on Computer Architecture (ISCA-28)*, pages 2–13, 2001.

Vita

Onur Mutlu was born in Kayseri, Turkey on 6 November 1978, to father Nevzat Mutlu and mother Hikmet Mutlu. After early schooling in various cities and countries of the world, he finished Besiktas Ataturk Anatolian High School, Istanbul, Turkey in May 1996. Onur completed his undergraduate studies at the University of Michigan, Ann Arbor, where he received a B.S. degree in Psychology and a B.S.E. degree in Computer Engineering in August 2000. Onur has been a graduate student at the University of Texas since August 2000. He received an M.S.E. degree in Electrical and Computer Engineering from the University of Texas at Austin in 2002.

Onur received the University of Texas Continuing Fellowship (2003), the Intel Foundation PhD fellowship (2004), and the University Co-op/George H. Mitchell Award for Excellence in Graduate Research (2005) during his graduate studies. He served as a teaching assistant for *Introduction to Computing* and *Computer Architecture* courses. His research has been published in major computer architecture conferences (MICRO, ISCA, HPCA, CGO, DSN). Three of the papers he co-authored have been selected in a collection of the most industry-relevant computer architecture research papers by the *IEEE Micro* technical magazine in the years 2003 and 2005. Onur worked at Intel Corporation during summers 2001-2003 and at Advanced Micro Devices during summers 2004-2005. When he is not doing research, Onur enjoys reading, thinking, hiking, and traveling.

Permanent address: Koru Mahallesi, 39. Cadde, Yesiltepe Evleri, No. 49
Cayyolu, Ankara, Turkey

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.