Out-of-Order Fetch, Decode, and Issue

by

Jared Warner Stark IV

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2000

Doctoral Committee:

Professor Yale N. Patt, Chair
Professor Richard B. Brown
Professor Edward S. Davidson
Assistant Professor Steven Reinhardt
Michael Shebanow, VP and Chief Technical Officer, HAL Computer Systems

ABSTRACT

Out-of-Order Fetch, Decode, and Issue

by

Jared Warner Stark IV

Chair: Yale N. Patt

To exploit larger amounts of parallelism, processors are being built with ever wider issue widths. Unfortunately, as issue widths grow, instruction cache misses increasingly bottleneck performance.

Out-of-order fetch, decode, and issue reduces the bottleneck due to these misses. The term *issue* denotes the act of writing instructions into reservation stations, *fetch block* denotes the instructions brought into the processor by an instruction cache fetch, and *branch predictor* denotes the entire next fetch address generation logic. Upon encountering an instruction cache miss, a conventional processor waits until the miss has been serviced before continuing to fetch, decode, and issue any new fetch blocks. A processor with out-of-order fetch, decode, and issue temporarily ignores the block associated with the miss, and attempts to fetch, decode, and issue the blocks that follow. The addresses of these blocks are generated by the branch predictor. Because the predictor does not depend on the instructions in the current block to generate the address of the next fetch block, it can continue making predictions even if the current block misses in the cache. Thus, the processor can skip the block that missed and continue fetching, decoding, and issuing the following blocks using the addresses generated by the predictor. After servicing the miss, the processor can fetch, decode, and issue the skipped block.

This dissertation evaluates the *potential* performances of processors with various issue widths and window sizes, shows that enough potential performance exists to justify building a 16-wide processor, examines bottlenecks that would prevent this processor from achieving its potential performance, and demonstrates that the bottleneck due to instruction cache misses would be severe. It introduces a remedy for this bottleneck—out-of-order fetch, decode, and issue—describes its implementation, and demonstrates its usefulness. For a 16-wide processor with a 16k byte direct-mapped instruction cache, instruction cache misses reduce its performance on a set of sixteen integer benchmarks by an average of 24%. When it is equipped with out-of-order fetch, decode, and issue, its performance is only reduced by an average of 5%.

To my family.

ACKNOWLEDGEMENTS

This work would not have been possible without the help and support of many people. In particular, I would like to thank the following:

- Professor Yale Patt, my advisor, for sparking my interest in computer architecture many years ago when I was an undergraduate, and for his advice and guidance over the years since then.

- My committee and Joel Emer for feedback on my dissertation and for suggestions on how to make it better. Their suggestions greatly improved its quality.

- Eric Hao for the initial brainstorming on out-of-order fetch, decode, and issue.

- Paul Racunas for further brainstorming on out-of-order fetch, decode, and issue; and for providing feedback and proofreading the initial versions of this dissertation.

- Marius Evers, who also provided feedback and proofread the initial versions of this dissertation.

- The other members of the HPS research group, both past and present, for the stimulating environment they provide.

- Our industrial partners—in particular Intel Corporation, HAL Computer Systems, and Advanced Micro Devices—for their financial support.

TABLE OF CONTENTS

LIST OF TABLES

Table

LIST OF FIGURES

Figure

CHAPTER 1

Introduction

Significant parallelism exists within a single instruction stream [14, 59]. To achieve high performance, today's processors are built to take advantage of some of this instruction level parallelism. Techniques such as speculative execution, dynamic scheduling, register renaming, and branch prediction are used to exploit the instruction level parallelism by removing or reducing performance bottlenecks that result from unpredictable latencies, false dependencies, and changes of flow in the instruction stream.

To exploit even larger amounts of instruction level parallelism, tomorrow's processors will be built with wider issue widths. In the past fifteen years, instruction issue widths have grown from one (Intel i486, Motorola 68020, Sun MicroSparc, MIPS R2000), to two (Intel Pentium, Alpha 21064, HP PA-7200), to four (PowerPC 604, Sun UltraSparc, Alpha 21164, MIPS R10000, HP PA-8000). It has been projected that by the year 2005, it will be possible to place a billion transistors on a chip. With a billion transistors on chip, a processor that can issue sixteen or more instructions per cycle is not infeasible. However, for such a processor, the occurrence of instruction cache misses will impose a severe performance penalty.

1.1 The Problem: The Instruction Cache Bottleneck

Instruction cache misses interrupt the supply of instructions to the processor. When an instruction cache miss occurs, the front end of the processor (i. e., the instruction fetch and decode pipelines) stalls until the instructions have been obtained from the next level of the memory hierarchy. During the time of this stall, no new instructions are supplied to the processor core.

A simple back-of-the-envelope calculation can be used to roughly assess the severity of this bottleneck. The calculation uses the instruction fetch width (i. e., the peak number of instructions that can be fetched from the instruction cache in a single cycle), the instruction cache miss rate, and the instruction cache miss penalty to calculate the performance degradation that results from instruction cache misses. It assumes that the processor does not stall unless there is an instruction cache miss. The calculation is described best by using the following example. Assume that the instruction fetch width is 16, the instruction cache miss rate is one miss per 100 instructions, and the instruction cache miss penalty is 10 cycles. On average, 99 instructions are fetched from the instruction cache before a miss occurs. These 99 instructions are fetched in about 6 cycles: 99 instructions, fetched at a rate of 16 per cycle, are fetched in 99/16 cycles. When the 100th instruction is fetched, the miss occurs. 10 cycles (the instruction cache miss penalty) ensue in which no instructions are fetched. Thus, for every 16 cycles, 6 cycles are spent fetching, and 10 are spent not fetching. The final result: over 60% ($10/16 \cdot 100\%$) of the processor's potential performance is lost due to instruction cache misses.

To obtain a more accurate assessment of the severity of the instruction cache bottleneck, software models of different processor microarchitectures can be used to simulate the execution of a program. Figure 1.1 was created using such models. It plots the performance, in Instructions Per Cycle (IPC), of the gcc benchmark for two different processors as the fetch width of these processors varies between 1 and 64. Both processors have perfect branch prediction, large instruction windows, large pools of functional units [1], and perfect data caches. To illustrate the performance degradation that results from instruction cache

[1] The difference in performance between a processor with an unbounded number of functional units and a processor where the number of functional units is twice the fetch width is negligible. At any given fetch width, the figure plots the performance of two processors that have a number of functional units equal to twice the fetch width. There is no perceptible difference between this figure and the analogous figure that plots the performance of processors that have an unbounded number of functional units.

misses, the first processor was given a perfect (100 percent hit rate) instruction cache and the second processor was given a real instruction cache. The real instruction cache is a 16k byte direct mapped cache with a 64 byte line size and a 10 cycle miss penalty. The difference between the performance of the processor with the perfect instruction cache and the performance of the processor with the real instruction cache is the performance that is lost due to instruction cache misses. Note that as the fetch width increases, the fraction of the performance lost due to instruction cache misses increases. At a width of 1, only 22% of the performance is lost. At a width of 16, 81% of the performance is lost. And at a width of 64, 90% of the performance is lost. Thus, as issue widths grow, the instruction cache bottleneck becomes more severe.

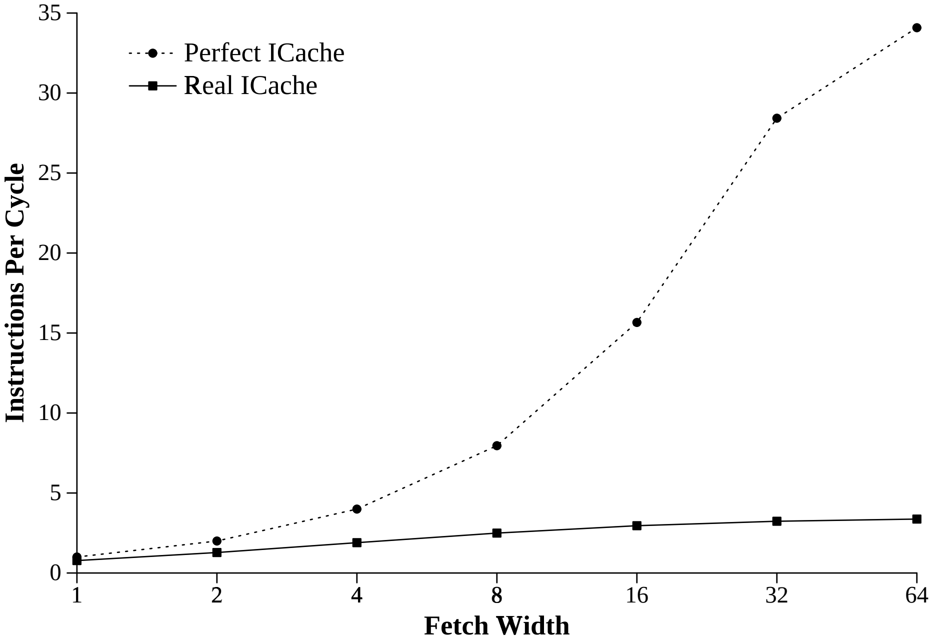


Figure 1.1: Demonstration of the problem (the instruction cache bottleneck) investigated by this dissertation

1.2 The Solution: Out-of-Order Fetch, Decode, and Issue

The occurrence of instruction cache misses imposes a severe performance penalty on wide issue processors. Instruction cache misses prevent the processor from fetching, decoding, and issuing new useful instructions until the instruction cache miss has been serviced. Out-of-order fetch, decode, and issue reduce the performance penalty due to instruction cache misses by enabling the processor to continue fetching, decoding, and issuing new useful instructions even in the event of an instruction cache miss.

Note: there are no standard definitions for the terms *issue* and *dispatch*. Throughout this dissertation, I will use the term *issue* to indicate the act of writing the instructions into the reservation stations, and the term *dispatch* to indicate the act of sending the instructions from the reservation stations to the functional units (see Figure 1.2). (There is a caveat about these definitions: many people interchange these definitions for the terms issue and dispatch; that is, they use the term issue to indicate the act of sending the instructions from the reservation stations to the functional units, and the term dispatch to indicate the act of writing the instructions into the reservation stations.) Many of today's high performance processors employ out-of-order execution, in which instructions are *dispatched* to the functional units out of the normal program order. However, for all these processors, instructions are still *issued* to the reservation stations in program order. This dissertation addresses out-of-order issue, that is, the process by which instructions are written into the reservation stations out of normal program order.

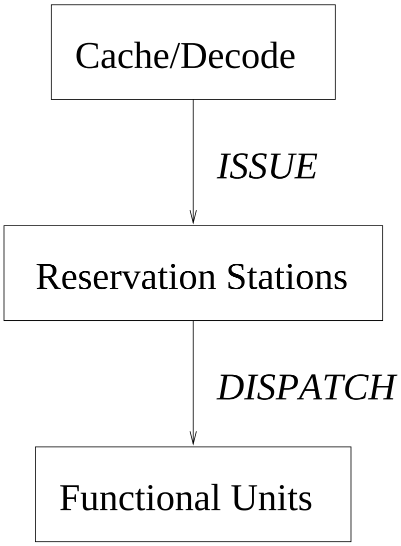


Figure 1.2: Terminology

Out-of-order fetch, decode, and issue reduce the performance penalty caused by instruction cache misses. I will use the term *fetch block* to refer to the group of instructions that are brought into the processor by an instruction cache fetch. A fetch block is a sequence of consecutive instructions that contains a single branch. The sequence starts with an instruction that is the target (either taken or not-taken [fall-through]) of a branch and ends with the branch. Upon encountering an instruction cache miss, a conventional processor will wait until the instruction cache miss is serviced before continuing to fetch, decode, and issue any new fetch blocks. A processor with out-of-order fetch, decode, and issue temporarily ignores the fetch block associated with the instruction cache miss, and attempts to fetch, decode, and issue the fetch blocks that follow the block associated with the miss. The addresses of these blocks can be generated by the branch predictor. (I will use the term *branch predictor* to refer to all the next fetch address generation logic.) Because the branch predictor does not depend on the instructions in the current block to generate the address of the next block to be fetched, it can continue to make predictions even when the current block misses in the instruction cache. Thus, the processor can skip the block that missed in the instruction cache and continue fetching, decoding, and issuing the following block using the address generated by the branch predictor. After the instruction cache miss is serviced, the processor can fetch, decode, and issue the skipped block.

Consider the example in Figure 1.3. It shows a graph consisting of five fetch blocks A–E. In the first cycle, the processor fetches block A and the predictor generates address B for the next block. Suppose in the second cycle, the processor's fetch of block B results in an instruction cache miss. The predictor can still generate the address for the next block, block C. The processor will fetch nothing in the second cycle, but can attempt to fetch block C in the third cycle, followed by block D in the fourth cycle (regardless of whether C hit in the cache or not). Once the instruction cache miss is serviced, the processor can return to the point of the miss and fetch, decode, and issue the skipped block(s), starting with block B. Thus, out-of-order fetch, decode, and issue allows the processor to fetch, decode, and issue useful work in the presence of instruction cache misses. In the worst case, where all the following fetches result in instruction cache misses, out-of-order fetch, decode, and issue still provides the performance benefit of prefetching.

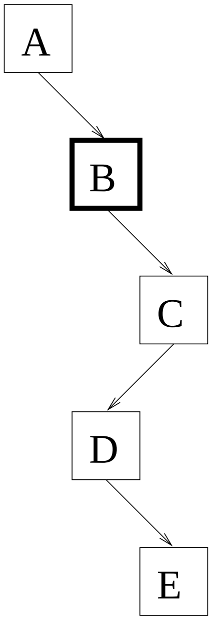


Figure 1.3: Out-of-Order Fetch, Decode, and Issue Example

1.3 Thesis Statement

The performance penalty that results from instruction cache misses is nearly eliminated if the processor is allowed to fetch, decode, and issue instructions out of program order. The penalty is defined as the difference in performance, in Instructions Per Cycle, between a processor with a real instruction cache and a comparable processor with a perfect (100 percent hit rate) instruction cache. For the processors with sixteen wide issue and 16k byte direct mapped instruction caches studied in this dissertation, for sixteen integer benchmarks, on average about three-quarters of this penalty is eliminated.

1.4 Contributions

This dissertation makes four major contributions:

1. This dissertation evaluates the benefit of out-of-order fetch, and describes the implementation of out-of-order fetch. A processor with out-of-order fetch initiates fetch requests in program order, but allows these requests to complete out-of-order. As in conventional processors, the instructions are decoded and issued in program order. Out-of-order fetch is almost as effective as out-of-order fetch, decode, and issue in eliminating the performance penalty that results from instruction cache misses. However, the implementation of out-of-order fetch is much simpler than the implementation of out-of-order fetch, decode, and issue.

2. This dissertation evaluates the benefit and describes the implementation of out-of-order fetch, decode, and issue. It demonstrates that the performance penalty that results from instruction cache misses is nearly eliminated if the processor is allowed to fetch, decode, and issue instructions out of program order. *Note: hereafter, out-of-order fetch, decode, and issue will be called* out-of-order fetch/decode/issue. *Out-of-order fetch with in-order decode and issue will simply be called* out-of-order fetch. *I will use the phrase* out-of-order fetch, decode, and issue *to refer to both out-of-order fetch and out-of-order fetch/decode/issue.*

3. This dissertation proposes a new technique for creating the index into the Branch Target Buffer (BTB). The BTB is a cache that the branch predictor uses to determine the type of a branch (unconditional branch, conditional branch, branch to subroutine, jump [computed branch], jump to subroutine, or subroutine return) and some of its possible target addresses. If a BTB access results in a miss, the branch predictor is typically led astray, which results in one or more cycles of nonproductive fetching. Thus, a BTB with a low miss rate is desirable. Conventional indexing schemes create "hot spots" in the BTB. (A "hot spot" is a set of BTB entries that are read and/or written a disproportionate number of times.) The proposed technique eliminates most of these "hot spots".

4. This dissertation evaluates the potential performances of processors with various issue widths, instruction window sizes, and memory dependency handling techniques. It then examines the bottlenecks that prevent these processors from achieving their potential performance. The architects of future processors can use this information to make design decisions. For example, this information can be used to answer the questions: Does it make sense to build a processor that can issue sixteen instructions per cycle? How many reservation stations should a sixteen wide issue processor have? And, will instruction cache misses significantly bottleneck a sixteen wide issue processor?

1.5 Dissertation Organization

This dissertation is organized into nine chapters. Chapter 2 presents the related work. Chapter 3 describes the machine model, the simulators, and the benchmarks used throughout the dissertation. Chapter 4 uses an abstract machine model to calculate the amount of parallelism available in a single instruction stream. Additionally, the *potential* performances of machines with various issue widths, instruction window sizes, and memory dependency handling techniques are calculated. The data in this chapter shows that there is enough potential performance to justify building a machine that can issue sixteen instructions per cycle. Chapter 5 examines some bottlenecks that would prevent such a machine from achieving its potential performance. Among these bottlenecks are the bottleneck that results from having a real instruction cache, the bottleneck that results from having a real branch predictor, the bottleneck that results from having a real execution core, and the bottleneck that results from having a real data cache. Chapter 6, using an abstract machine model, demonstrates that the instruction cache bottleneck for a sixteen wide issue machine can be nearly eliminated by allowing the machine to fetch, decode, and issue instructions out of program order. Chapter 7 describes the implementation of out-of-order fetch, decode, and issue. Chapter 8 uses a realistic machine model to evaluate the actual performance benefit of out-of-order fetch, decode, and issue. It also examines various implementation tradeoffs that could not be evaluated with the abstract machine model. Chapter 9 provides some concluding remarks.

CHAPTER 2

Related Work

Researchers have proposed many solutions for dealing with instruction cache misses. Many of these solutions can be used in combination with each other to provide better overall performance than if only a single solution were used alone. For example, the compiler can eliminate some of the instruction cache misses by re-ordering the instructions in a program. To tolerate the remaining misses, the machine can fetch, decode, and issue instructions out-of-order.

Research on the handling of instruction cache misses can be divided into two areas. The first area contains all research that looks at eliminating (some of the) instruction cache misses. Most research has been done in this first area. The second area contains all the research that looks at tolerating instruction cache misses. Out-of-order fetch, decode, and issue falls into the second area.

2.1 Eliminating Instruction Cache Misses

Many techniques exist for reducing the number of cache misses. Many of these techniques are applicable to instruction caches. Examples include improved hashing functions [2, 109, 123], better cache management policies [56, 70, 79, 107], cache conscious virtual page mapping [9, 66], and victim caches [58]. In this section, I will describe some of the most promising techniques aimed at reducing the number of instruction cache misses. These techniques can be placed in one of the following three categories: increasing the cache associativity, re-ordering the code, and prefetching.

2.1.1 Increasing Cache Associativity

One way to reduce the number of instruction cache misses is to build a set-associative cache instead of a direct-mapped cache. However, set-associative caches have longer access times than direct-mapped caches. In a direct-mapped cache, cache line data can be (speculatively) used by the next stage of processing before the tag comparison for the line has completed. If the tag comparison fails, the next stage of processing simply discards the data. On a cache access for a set-associative cache, all the lines in a set are read out, and their tags are compared to the address of the requested data to determine which line contains the sought after data. Once the tag comparisons have completed, the matching line (if there is one) is identified, and its data is passed on to the next stage of processing. Thus, for a set-associative cache, the tag comparisons introduce an extra delay between the time when the cache data becomes available and the time when the data can be (speculatively) used by the next stage of processing.

What is needed is an instruction cache with the organization and access time of a direct-mapped cache, *and* the hit rate of a set-associative cache. There are two approaches to achieving this goal. The first approach is to build a direct-mapped cache that uses several different hash functions. The second approach is to build a set-associative cache with a *way predictor*.

The hash-rehash cache [3] uses the first approach. It uses two different hash functions to access the cache. Initially, the first hash function is applied to the address, and the result is used to index into the cache. If the resulting cache line contains the sought after data, a *first-time* hit occurs. If it misses, the second hash function provides an index into the cache. If a *second-time* hit occurs, the data is retrieved. The data in the two cache lines is then swapped so that the next access will likely result in a first-time hit. However, if the second access also misses, then the data is retrieved from main memory, placed in the cache line indexed by the second hash function, and then swapped with the data indexed by the first hash function.

The column-associative cache [1] improves the performance of the hash-rehash cache by adding a *rehash bit* to each line in the cache. The rehash bit is reset if the data stored in that cache line is associated with the first hash function, and set if the data stored in that cache line is associated with the second hash function. When a first-time miss occurs, if the rehash bit is reset, the processor tries to find the data by accessing the cache via the second hash function. If the rehash bit is set, the cache is not accessed again, and the data is retrieved from main memory. When a cache line must be replaced, a line whose rehash bit is set is preferred as the victim.

The second approach is to build a set-associative cache with a way predictor. The RAM in these caches is organized just like the RAM in direct-mapped caches: each index into the RAM specifies the location of one, and only one, cache line. The way predictor guesses which cache line within a set contains the sought after data. The way prediction and the set index are combined to form an index into the cache's RAM. If the indexed line contains the data, the data is retrieved. Otherwise, the remaining lines in the set are probed. If the data is found in one of the other lines, it is retrieved. If it is not found in any of the lines, it must be retrieved from main memory.

For the MRU cache [18, 67, 118], the way predictor records the line that was used most recently for each cache set. When the cache is accessed, the most recently used line is predicted to contain the sought after data. Note that for caches that implement a least recently used (LRU) replacement policy, the LRU information for each set indicates which line was used most recently. Thus, there is little additional overhead in implementing an MRU cache if the underlying cache uses an LRU replacement policy.

For the Direct-mapped Access Set-associative Check (DASC) cache [110], the address used to access the cache provides the way prediction. When the cache is accessed, the lower order bits of the tag portion of the address are used to select the line within the set. (That is, the data array is accessed as if the cache was direct-mapped.) If the prediction is incorrect, and a hit in detected in one of the other lines in the set, the two lines (the predicted line and the line registering a hit) are swapped. In the case of a miss, the line is written according to the replacement algorithm, and then swapped with the predicted line.

Johnson [55] proposed a technique that eliminates the need for a Branch Target Buffer (BTB) by embedding a set and way predictor in the instruction cache. (The way predictor is not needed if the cache is direct-mapped.) This technique was used for the UltraSPARC's two-way set associative instruction cache [137]. In the UltraSPARC, each instruction cache line contains eight instructions, two set predictions (one for the first four instructions and one for the second four instructions), and two way predictions (again, one for the first four instructions and one for the second four instructions). The UltraSPARC fetches at most four instructions from the cache each cycle. When the instructions are fetched, the associated set prediction and way prediction are also fetched. These predictions are then used in the following cycle to fetch the next cache line.

Calder and Grunwald [17] integrated the functions performed by a Branch Target Buffer (BTB) and a way predictor into a common structure, which they call the next cache line and set (NLS) predictor.[1] After a taken branch is fetched, the processor consults the NLS predictor to determine which cache line to fetch next. Unlike a BTB, which specifies the target address of the branch, the NLS predictor specifies the set and way of the cache line that contains the target.

Finally, Juan, Lang, and Navarro introduced the difference-bit cache [63], which is a two-way set associative cache that has an access time close or equal to that of a direct-mapped cache. In conventional two-way set associative caches, both tags in a set are compared to the address. After the comparisons are complete, the matching tag is then used to select the line within the set. Juan, Lang, and Navarro noticed that the two tags within a cache set must differ by at least one bit. When an address is presented to the difference-bit cache, it is also presented to a small memory. For the cache set that corresponds to that address, the memory identifies the position of the least significant bit in which the two tags differ. This bit is then extracted from the address, and used to select the line within the set. Thus the line can be selected without performing any tag comparisons, which, hopefully, reduces the critical path for the cache access. Of course, the selected line will not contain the requested data if that data is not in the cache. A tag comparison is still required to determine if the selected line actually contains the requested data.

2.1.2 Code Re-Ordering

The compiler can reduce the number of instruction cache misses by re-ordering the instructions in a program.

McFarling [78] described a program re-ordering algorithm for direct-mapped instruction caches. The program is profiled to determine basic block execution counts. The algorithm constructs a Directed Acyclic Graph (DAG) of the program consisting of loop,

[1] The term *set* usually refers to a group of cache lines associated with a common index. Calder and Grunwald use the term *set* to refer to a particular cache line within one of these groups. This nonstandard nomenclature was undoubtably borrowed from Digital Equipment Corporation, with whom Calder and Grunwald collaborated.

procedure, and basic block nodes. Each loop node is labeled with the average number of times the loop was executed during the profile run. Edges are labeled with a fraction that indicates how often the child was executed when the parent was executed. The algorithm then partitions the graph, paying special attention to the loop nodes, with the goal of fitting each subgraph into the cache without any conflicts. The partitioning is done based on the labels assigned to the loop nodes and the edges. McFarling also showed that higher hit rates are possible if the algorithm can force the cache to exclude certain instructions.

Hwu and Chang [50] described a five step re-ordering algorithm. In step one, the program is profiled to determine the number of times each basic block is executed, and the number of times each path between a pair of basic blocks is traversed. In step two, frequently executed procedures are integrated (or inlined) into their callers. Although inlining may increase code size, and hence the number of compulsory instruction cache misses, it also improves spatial locality, since almost all control transfers are within procedures rather than between procedures. Additionally, inlining reduces potential cache mapping conflicts between procedures, thus reducing the number of potential conflict misses. In step three, basic blocks that tend to execute in sequence are grouped into traces. The basic blocks in a trace are placed in contiguous memory to improve sequential and spatial locality. In step four, the traces that comprise each procedure are placed. To further improve sequential and spatial locality, traces that tend to execute in sequence are placed in contiguous memory. To minimize a procedure's cache footprint, the traces are also loosely arranged in the order of most frequently executed to least frequently executed. Finally, in step five, global analysis arranges the program's procedures to reduce inter-function cache conflicts.

Pettis and Hansen [103] described the following techniques for improving code layout: basic block reordering, procedure reordering, and procedure splitting. These techniques are not mutually exclusive. All techniques rely on profile information. Their basic block reordering technique is similar to Hwu and Chang's: it groups together the basic blocks that tend to execute in a sequence, and then places the basic blocks within a group in contiguous memory. This improves sequential and spatial locality. Their procedure reordering technique tries to place a caller and its most frequent callees close to one another in the final code image. Doing this decreases the chances that the caller and its callees will contend for the same cache lines. Their procedure splitting technique partitions the basic blocks of each procedure into two groups: *primary* blocks and *fluff* blocks. Primary blocks are the blocks that were executed during the profile run. Fluff blocks are the blocks that were not executed. When the linker lays out the final code image, it places all the primary blocks (for all the procedures) at the beginning of the image, and all the fluffs blocks at the end. This minimizes the size of the program's cache footprint.

Gupta and Chi [41] looked at repositioning code so that either all instructions belonging to an instruction cache line were executed or none of them were executed. In some cases, repositioning code reduces the number of cache misses. For example, sometimes the number of cache lines occupied by a loop is reduced by one if the loop is repositioned to start at a cache line boundary. Reducing the size of the loop's cache footprint will reduce the number of misses the loop experiences. Unlike the techniques that were described above, this code repositioning technique does not require profiling.

If not done with care, procedure inlining can dramatically increase the size of a program's executable image. As a result, the number of instruction cache misses may soar, and machine performance may actually decrease. McFarling [80] proposed a procedure inlining algorithm that accounts for instruction cache behavior when deciding whether or not to inline a procedure. For each procedure, the benefit and cost of inlining the procedure is estimated. The benefit is the number of instructions that can be eliminated by inlining the procedure, multiplied by the number of times the procedure is executed. (The program is profiled to determine how often each procedure is executed.) The cost is the (estimated) number of extra cache misses that will occur if the procedure is inlined, multiplied by the cache miss penalty (or cost of each miss). If the benefit outweighs the cost, the procedure is inlined.

Mendlson, Pinter, and Shtokhamer [86] described a program re-ordering algorithm similar to McFarling's [78]. Unlike McFarling's, this algorithm does not require profile information and it can be applied to set-associative caches. The algorithm constructs a Nested Flow Graph (NFG), which is a control flow graph augmented with information about the nesting structures of loops and procedures. The algorithm then partitions the graph, based on the information about the nesting structures, with the goal of fitting each subgraph into the cache without any conflicts. In some cases, the algorithm must replicate code in order to avoid conflicts.

Torrellas, Xia, and Daigle [124, 131] proposed an algorithm for repositioning operating system code. Their algorithm was similar to Hwu and Chang's: it groups together the basic blocks that tend to execute in a sequence, and then places the basic blocks within a group in contiguous memory. The algorithm then arranges the groups in the order of most frequently executed to least frequently executed. However, unlike Hwu and Chang's algorithm, their algorithm can group together basic blocks that cross procedure boundaries. In addition, their algorithm is conscious of the instruction cache. Frequently executed groups of basic blocks are assigned to a special "conflict-free" area of the cache. The remaining groups are assigned such that they use this special area as little as possible.

Hashemi, Kaeli, and Calder [48, 49] introduced a procedure reordering algorithm called *color mapping*. The algorithm uses a call graph. Each node corresponds to a procedure. An edge identifies each caller/callee pair. The edge is weighted by the number of times the caller calls the callee. Edge weights can be determined by either profiling [48] or compile-time heuristics [49]. The algorithm processes edges in the order of heaviest weighted to lightest weighted. When processing an edge, the procedures associated with the edge are mapped into the address space, and each procedure is assigned the cache lines (colors) that it will use. As the other procedures are mapped into the address space, cache conflicts are avoided by using these colors to guide procedure placement.

The color mapping algorithm for procedure reordering only eliminates *first generation* cache conflicts; that is, conflicts between a procedure and any of its immediate callers or callees. Kalamatianos and Kaeli [64] introduced an enhanced color mapping algorithm that also eliminates higher order generation conflicts. This algorithm uses a Conflict Miss Graph (CMG) instead of a call graph. Each node in the CMG corresponds to a procedure. If two procedures can conflict with each other, an edge is placed between their corresponding nodes. The edge is weighted by an approximation of the worst-case number of misses the two competing procedures can inflict on each other. The algorithm uses profile information to determine where edges should be placed, and what their weights should be. The remainder of the algorithm is identical to the original color mapping algorithm, except that the CMG is processed instead of the call graph.

2.1.3 Prefetching

With prefetching, the hardware and/or compiler anticipates that data is about to be accessed that is not in the cache. The hardware attempts to fetch this data before the access occurs.

Smith [115] examined three hardware prefetch algorithms: *always prefetch*, *prefetch on misses*, and *tagged prefetch*. For all three algorithms, when a line is fetched, the processor prefetches (or may prefetch) the line located sequentially after the fetched line. For *always prefetch*, the processor prefetches after every fetch. The major disadvantage of this algorithm is that it generates a large amount of prefetch traffic. For *prefetch on misses*, the processor prefetches only if the fetched line misses. This can cut the number of misses for a purely sequential reference stream in half. For *tagged prefetch*, each line in the cache has a single bit called a *tag*. When a new line is inserted in the cache, its tag is set to zero. When a line in the cache is accessed, its tag is set to one. If a tag changes from a zero to a one, the processor prefetches the next sequential line. For a purely sequential reference stream, this algorithm can reduce the number of misses to zero.

Smith and Hsu [117] described several hardware prefetch algorithms for instruction caches. For *fall-through prefetch* (also referred to as "one block lookahead" [115]), when a line is fetched, the processor prefetches the next sequential line if the requested instruction falls within some specified distance (called the *fetchahead distance*) from the end of the cache line. If the fetchahead distance is short, the processor is more likely to use the prefetched line, but less likely to receive the instructions soon enough to avoid any miss penalty. For *target-line prefetching*, a *target prediction table* is accessed whenever an instruction is fetched from the cache. The table entry indicates the address of the line that most recently followed the current instruction line. Smith and Hsu also describe a target prediction table that allows each cache line to specify multiple successor lines—one successor line for each branch instruction in the current line. The address identified by the table entry is prefetched the next cycle. The key advantage that target-line prefetching has over fall-through prefetch is that is more accurately predicts which cache lines will be needed, and thus it generates better prefetches. A third prefetch algorithm combines fall-through prefetch and target-line prefetching in a hybrid. Each cycle the processor generates two prefetches: one prefetch for the next sequential line and one prefetch for a (non-sequential) line specified by the target prediction table. For conditional branches, the result of this algorithm is that both the not-taken and taken paths of the branch are prefetched. Smith and Hsu found that this hybrid algorithm performed significantly better than either component algorithm alone.

Wrong-path instruction prefetching [104], proposed by Pierce and Mudge, is similar to Smith and Hsu's hybrid algorithm, except it spares the expense of the target prediction table. This algorithm combines fall-through prefetch with the prefetching of all conditional branch targets regardless of the predicted directions of the branches. When an instruction cache line is fetched, the processor also prefetches the next sequential line. Additionally, whenever a conditional branch is decoded, the line containing its taken target is also prefetched. Thus, both targets of a conditional branch are always prefetched: the not-taken target with fall-through prefetching, and the taken target with the target prefetching. Note that if a conditional branch is taken, prefetching the line that contains its taken target *after* it has been decoded is not very useful, since the processor generates a normal fetch request for that line anyway. The prefetch of that line can only be useful if the branch is not-taken. Hopefully, the next time the branch is fetched and predicted to be taken, the line containing the taken target will reside in the cache because of the prefetch.

The prefetch algorithms described above were designed for machines that fetch a small number of instructions from the cache each cycle. For these machines, a single cache line is accessed many times before the machine moves on to the next line. All prefetches are initiated the first time the cache line is accessed. By the time the next cache line is accessed, the prefetches have (hopefully) completed. For machines that fetch a large number of instructions from the cache each cycle, a single cache line is accessed only once before the machine moves on to the next line. Any prefetches initiated by accessing that line will not complete before the next line is accessed. To improve the performance of these algorithms for wide fetch machines, each prefetch is modified to request N sequential lines rather than just a single line.[2] A large value of N tends to increase the prefetching distance, but also increases the likelihood of polluting the cache with useless prefetches.

To reduce cache pollution, sequentially prefetched lines can be inserted in a stream buffer [58] instead of the instruction cache. A stream buffer is a FIFO prefetch buffer used to eliminate some capacity and compulsory misses. Each entry in the FIFO contains a cache line. When a cache miss occurs, the entry at the head of the FIFO is probed for the data. If the entry contains the data, the cache line associated with the entry is copied to the cache. The entry is then removed from the stream buffer by advancing the FIFO. If the entry at the head of the FIFO does not contain the requested data, all entries in the FIFO are invalidated, and the data is fetched from the next level of the memory and written into the cache. In addition, the stream buffer begins prefetching sequential cache lines, starting

[2] The original *next N-line prefetch* algorithm is described by Smith [114, 115].

with the cache line just after the cache line that caused the miss. The prefetched lines are inserted in the FIFO instead of the cache. Enhancements to the stream buffer have been proposed by Palacharla and Kessler [95], and by Farkas et al. [32, 33]

Xia and Torrellas [131] proposed a prefetch algorithm that uses the compiler to guide a hardware prefetch engine. The compiler first uses an algorithm similar to Hwu and Chang's [50] to arrange a program's basic blocks to reduce the number of instruction cache misses. Next, the compiler marks the beginning and end of each segment of straight-line code. Xia and Torrellas classify instruction cache misses as either being *sequential* misses or *transition* misses. A sequential miss is a miss that occurs when the machine is fetching a segment of straight-line code. A transition miss is a miss that occurs when the machine fetches the target of a (taken) branch. To eliminate sequential misses, when the prefetch engine encounters a segment of straight-line code (which is delineated by the markers that were inserted by the compiler), it prefetches all cache lines in that segment. To eliminate transition misses, the compiler inserts, as early as possible in the code, a software prefetch instruction that identifies the target of the next branch the compiler predicts will be taken. When the prefetch engine encounters a prefetch instruction, it prefetches all cache lines in the segment of straight-line code identified by the target of the prefetch instruction.

Joseph and Grunwald [57] introduced *Markov prefetching.* A Markov prefetcher builds a Markov model of the *miss address stream* by examining the sequence of addresses that miss in the cache, and then uses this model to predict the sequence of misses. Each time a miss occurs, the miss address is used to interrogate the Markov model. The Markov model spits out the addresses of misses that follow the current miss. Each of these addresses is assigned a prefetch priority according its probability of following the current miss. For example, assume that $miss_A$ follows the current miss 75% of the time, and that $miss_B$ follows the current miss 25% of the time. When the Markov model is interrogated, it will spit out the addresses of $miss_A$ and $miss_B$. The address of $miss_A$ will be assigned a higher prefetch priority than the address of $miss_B$. The prefetch engine then prefetches these addresses, starting with the highest priority prefetch. The Joseph and Grunwald study focused on data cache misses. However, this type of prefetching could also be used for instruction caches.

Chen, Lee, and Mudge [21] describe a prefetch engine that uses a branch predictor to run ahead of the instruction fetch unit and to prefetch potentially useful instructions.

Their prefetch engine races through the program's predicted dynamic instruction stream at a rate close to one fetch block per cycle. Their instruction fetch unit and execution core, on the other hand, handle at most four instructions per cycle. Since typical fetch blocks are larger than four instructions, the prefetch engine runs ahead of the processor. Unfortunately, future processors will fetch and execute one or more fetch blocks per cycle. For these processors, the prefetch engine will not run ahead of the instruction fetch unit, and, as a result, it will not be able to generate useful prefetch requests. An additional drawback of this approach is that it requires an additional branch predictor. (The authors claim that the processor shares its table of 2-bit counters with the prefetch engine. However, adding an additional port to this large table is not without expense. An additional claim is that the prefetch engine does not need a branch target buffer: it can fetch a cache line, identify the next branch, and then use a dedicated adder to calculate the target of the branch—all in one cycle. This claim is clearly absurd.)

Finally, Luk and Mowry [72] introduced a *prefetch filtering* mechanism that enables more aggressive sequential prefetching without polluting the cache with useless prefetches. A two-bit saturating counter is stored in each second level cache tag. The counter records the number of *consecutive* times the line was prefetched into the first level cache but was not used before it was replaced. When a prefetch request from the first level cache arrives at the second level cache, it is dropped if the counter associated with the requested line exceeds some threshold. In addition, Luk and Mowry inserted software prefetch instructions—described by Xia and Torrellas [131]—to eliminate the (transition) instruction cache misses that occur when the machine fetches the target of a (taken) branch. For the software prefetch instructions to be effective, the compiler schedules the prefetch instructions so that they execute X instructions before the branches execute, where X is the *prefetch-scheduling distance*. For example, if the cache miss latency is 12 cycles, and the expected performance is 1.6 Instructions Per Cycle (IPC), the prefetch-scheduling distance is about 20 (12×1.6). The size of the code increases as the prefetch-scheduling distance increases, since more prefetches must be inserted to cover the larger number of unique paths. Luk and Mowry found that for a prefetch-scheduling distance of 12, code size increased by an average of 8%, and that for a prefetch-scheduling distance of 28, code size increased by an average of 11%. Future machines will require larger prefetch-scheduling distances. For larger prefetch-scheduling distances, the increase in code size may offset any gains due to prefetching.

2.2 Tolerating Instruction Cache Misses

Increasing the cache associativity, re-ordering the code, and prefetching can all be used to eliminate instruction cache misses. The remaining related work covers microarchitectures or microarchitectural techniques that tolerate instruction cache misses once they occur. This work includes multiprocessors and multiscalar processors, multithreading, instruction stockpiling, and out-of-order fetch/decode/issue.

2.2.1 Multiprocessors and Multiscalar Processors

Programs for multiprocessors and Multiscalar processors [36] are broken up into a series of tasks. The tasks are distributed to processing elements. Each processing element has its own instruction cache. When one of the processing elements suffers an instruction cache miss, execution of the task on that processing element stalls. However, the execution of the other tasks in the program continues on the other processing elements.[3]

2.2.2 Multithreading

Multithreaded processors [125, 129, 132] tolerate instruction cache misses by exploiting thread level parallelism. Each cycle, the processor selects which thread it will fetch instructions from. If an instruction cache miss occurs while fetching instructions from the selected thread, the thread won't be selected again until its instruction cache miss has been serviced.

Multithreading improves *overall* performance by simultaneously processing multiple threads. However, multithreading does nothing to improve the performance of each individual thread. When the workload does not provide multiple concurrent threads, this becomes a significant limitation.

To overcome this limitation, we proposed Simultaneous Subordinate Microthreading (SSMT) [20]. An SSMT machine spawns subordinate threads that perform optimizations on behalf of the single primary thread. These threads, written in microcode, are issued

[3] To further enhance a multiprocessor's or Multiscalar processor's ability to tolerate instruction cache misses, its processing elements could be equipped to fetch, decode, and issue instructions out-of-order.

and executed concurrently with the primary thread. They directly manipulate the microarchitecture to improve the primary thread's branch prediction accuracy, cache hit rate, and prefetch effectiveness. All contribute to the performance of the primary thread.

The SSMT microcode routines for the subordinate threads are stored in a micro-RAM. Since they aren't stored in the instruction cache, they never suffer cache misses. This means that microthread instructions are available for issue even when primary thread instructions aren't, due to a cache miss. When the primary thread experiences an instruction cache miss, instructions from the subordinate threads are issued during the servicing of the miss.

2.2.3 Instruction Stockpiling

Another way to tolerate instruction cache misses is to add a FIFO buffer after the instruction fetch pipeline stage so that a "stockpile" of instructions can be built up. This buffer is typically inserted between the *fetch* and *decode* pipeline stages [40, 116]. Instructions are fetched from the cache and then written into the buffer in program order (or perhaps, in the predicted program order). During a cache miss, fetch stalls. As a result, no new instructions are written into the buffer while the miss is serviced. However, the execution of the stockpiled instructions may cover all or part of the miss penalty.

Drach and Seznec [30] described a pipeline organization in which two buffers are inserted between the *decode* and *execute* pipeline stages of a scalar processor. The organization was designed to reduce the penalties that result from both branches and instruction cache misses. The processor's instruction fetch bandwidth is twice its execution bandwidth. Each cycle, a pair of instructions stored at consecutive addresses is fetched, decoded, and then inserted into one of the two buffers. Each cycle, one of these buffers is selected to feed the processor's sole execution unit. When a branch is encountered, the processor's copious fetch bandwidth allows it to fill one buffer with instructions from the not-taken path, and the other with instructions from the taken path. When the branch is executed, its outcome (not-taken or taken) selects the buffer containing the instruction that will be executed in the next cycle. Thus, with this organization, a branch does not stall execution. When an instruction cache miss is encountered, the buffers can continue supplying the execution unit with new instructions. Since the fetch bandwidth is greater than the execution bandwidth, the buffers tend to fill up. Instruction cache misses give the execution unit a chance to catch up with instruction fetch and decode. In a conventional pipeline organization, the execution unit would stall during the instruction cache miss. With this organization, the execution unit does not stall.

The machines modeled in this dissertation use two buffers: one buffer (called the *fetch buffer*) between the fetch and decode pipeline stages, and one "buffer" between the decode and execute stages. The "buffer" between the decode and execute stages is simply the machine's set of reservation stations. All machines have a lockup-free [69] instruction cache; that is, an instruction cache that continues to service requests during a miss.[4]

For all machines modeled, instructions are removed (or retired) from the reservation stations in program order. When modeling a conventional machine (i. e., a machine that fetches, decodes, and issues instructions in program order), instructions are inserted into and removed from each of the buffers in program order. When modeling a machine that supports out-of-order fetch, instructions may be inserted into the fetch buffer out-of-order. However, instructions are still removed from the fetch buffer in order, and inserted into the reservation stations in order. When modeling a machine that supports out-of-order fetch/decode/issue, instructions may be inserted into and removed from the fetch buffer out-of-order, and inserted into the reservation stations out-of-order.

2.2.4 Out-of-Order Fetch/Decode/Issue

Vajapeyam and Mitra [127] described a microarchitecture that uses out-of-order fetch/decode/issue to eliminate the performance penalty that results from trace cache [98, 108] misses. To hasten instruction decode, the microarchitecture caches decoded trace cache lines. The trace cache contains the raw trace cache lines. Each line in the *rename cache* contains the decode information for a trace cache line. This information indicates which architectural registers are sourced by the trace cache line, and which architectural registers are written by the trace cache line. The lines contained in the rename cache are a superset of the lines contained in the trace cache.

During fetch, the trace cache and the rename cache are accessed in parallel. If

[4] All machines modeled in this dissertation speculatively fetch instructions from the wrong path whenever a branch is mispredicted. Instruction cache misses can occur while fetching from this wrong path. If the mispredicted branch resolves before these cache misses have been serviced, the machines immediately begin fetching instructions from the correct path—even though there are outstanding cache misses. The fetch requests that generated the wrong-path cache misses are converted into (wrong-path) prefetch requests. Prefetch requests that significantly contend for resources (e. g., the system bus) needed by fetch requests are dropped.

there are hits in both caches, everything proceeds as normal. If there are misses in both caches, fetch stalls while the requested instructions are obtained from the next level of the memory hierarchy. If there is a miss in the trace cache, but a hit in the rename cache, the information from the rename cache is used to rename the registers for the missing trace cache line. The processor saves the identifiers of the physical registers sourced by the trace cache line, and the identifiers of the physical registers written by the trace cache line. Fetch then continues on while the requested instructions are obtained from the next level of the memory hierarchy. Once the requested instructions have been obtained, they are decoded (using the physical register identifiers that were saved by the processor) and issued into the trace window.

Concurrently, we developed the concept of out-of-order fetch/decode/issue [120]. Our initial study proposed the concept and described three techniques for handling the register dependency problems that result from decoding instructions out of program order. One of these techniques is conceptually the same as the rename cache technique proposed by Vajapeyam and Mitra: it uses a separate cache (which we call a *mask cache*) to record the dependency information needed to rename registers. The other two techniques avoid the expense of caching dependency information, and, as a result, provide better performance for a given cache budget. Both our study and Vajapeyam and Mitra's found that processor performance improved significantly when the processor allowed out-of-order fetch/decode/issue.

CHAPTER 3

Simulation Methodology

3.1 Machine Model

The machine model used in this dissertation is the High Performance Substrate (HPS) [100, 101]. HPS allows multiple instructions to be completed per cycle by exploiting instruction level parallelism and by performing aggressive speculative execution. Hallmarks of HPS are aggressive branch prediction, wide instruction issue, multiple functional units, deep non-blocking pipelines, dynamic register renaming, out-of-order execution, and dynamic scheduling. Many elements of HPS are embodied in today's high end microprocessors, for example, the Intel Pentium II [27, 44], the AMD K7 [88], and the Compaq Alpha 21264 [39, 43, 65].

The soul of HPS is the Restricted Data Flow (RDF) model of execution [14, 100, 101]. With a classical data flow machine, the entire data flow graph for a program is resident in the machine, and the machine processes this data flow graph all at once [6, 25, 42]. An RDF machine, on the other hand, processes only a subset of a program's data flow graph at any one time. The term *active window* is used to refer to the set of instructions whose corresponding data flow nodes are resident in the RDF machine. As the active window slides through the dynamic instruction stream, the RDF machine executes the entire program.

Figure 3.1 illustrates the RDF model of execution. Instructions are removed from the program's dynamic instruction stream and converted into data flow nodes. Each instruction is converted into zero or more nodes. For the Alpha AXP instruction set architecture [112], most instructions are converted into a single node. The instructions are then inserted, or *issued*, into the active window. The *issue rate* specifies the maximum number of instructions that can be removed from the dynamic instruction stream, converted into data flow nodes, and then inserted into the active window in a single cycle. The *window size* specifies the maximum number of instructions that can be in the active window at any instant in time. Instructions are issued as long as the number of instructions in the active window is less than the window size. The data flow nodes are scheduled for execution when their flow dependencies have been resolved. An instruction exits the active window, or *retires*, after all its nodes have executed. Instructions are issued and retired in the order they appear in the dynamic instruction stream.

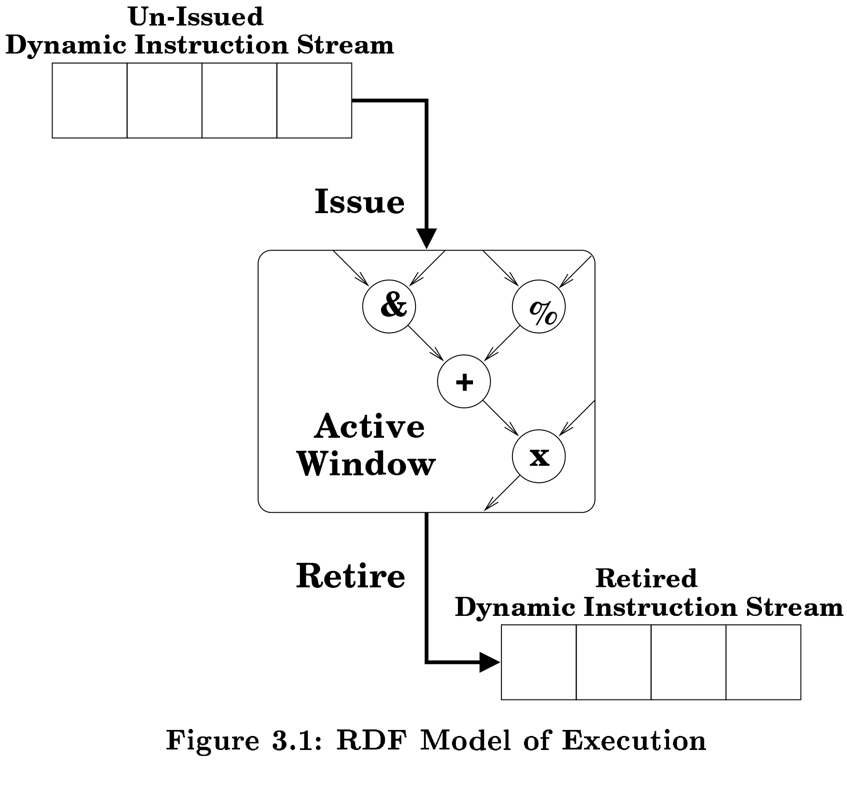


Figure 3.1: RDF Model of Execution

HPS is a realizable implementation of the RDF model of execution; i. e., it is an instance of the abstract RDF paradigm [14]. Figure 3.2 shows the basic microarchitecture. The branch predictor predicts the sequence of fetch blocks that comprise the dynamic instruction stream. The branch predictor does not predict the entire sequence all at once: it predicts only a piece of the sequence each cycle. The sequence is predicted in program order. State variables are used to keep track of the current point in the sequence. Each time a piece of the sequence is predicted, these state variables are updated. In certain situations, the branch predictor is forced to restart at an earlier point in the sequence. An example of such a situation is when a branch is mispredicted [60, 62, 113]. To restart the branch predictor, the state variables are simply set to the values they had at that earlier point.

The branch predictor predicts the sequence of fetch blocks that comprise the dynamic instruction stream without having the data for the instructions in those blocks. That is, the branch predictor can do its job without having *any* instruction data. Because of this, the branch predictor can operate independently of the instruction cache: it can sequence through the fetch blocks of the dynamic instruction stream even if those fetch blocks do not reside in the instruction cache.

Each cycle, the branch predictor predicts the next few fetch blocks in the dynamic instruction stream. Subsequently, the machine attempts to fetch each of these blocks from the instruction cache. On a cache hit, the instructions in the block are written into the *fetch buffer*. On a cache miss, the block is fetched from the next level of the memory hierarchy. During this fetch, the branch predictor and instruction cache stall. Additionally, all blocks logically following the block being fetched are flushed from the machine. After the fetch completes, the instructions in the block are written into the instruction cache and fetch buffer, the branch predictor is forced to predict the blocks that follow the block that missed, and the branch predictor and instruction cache are restarted.

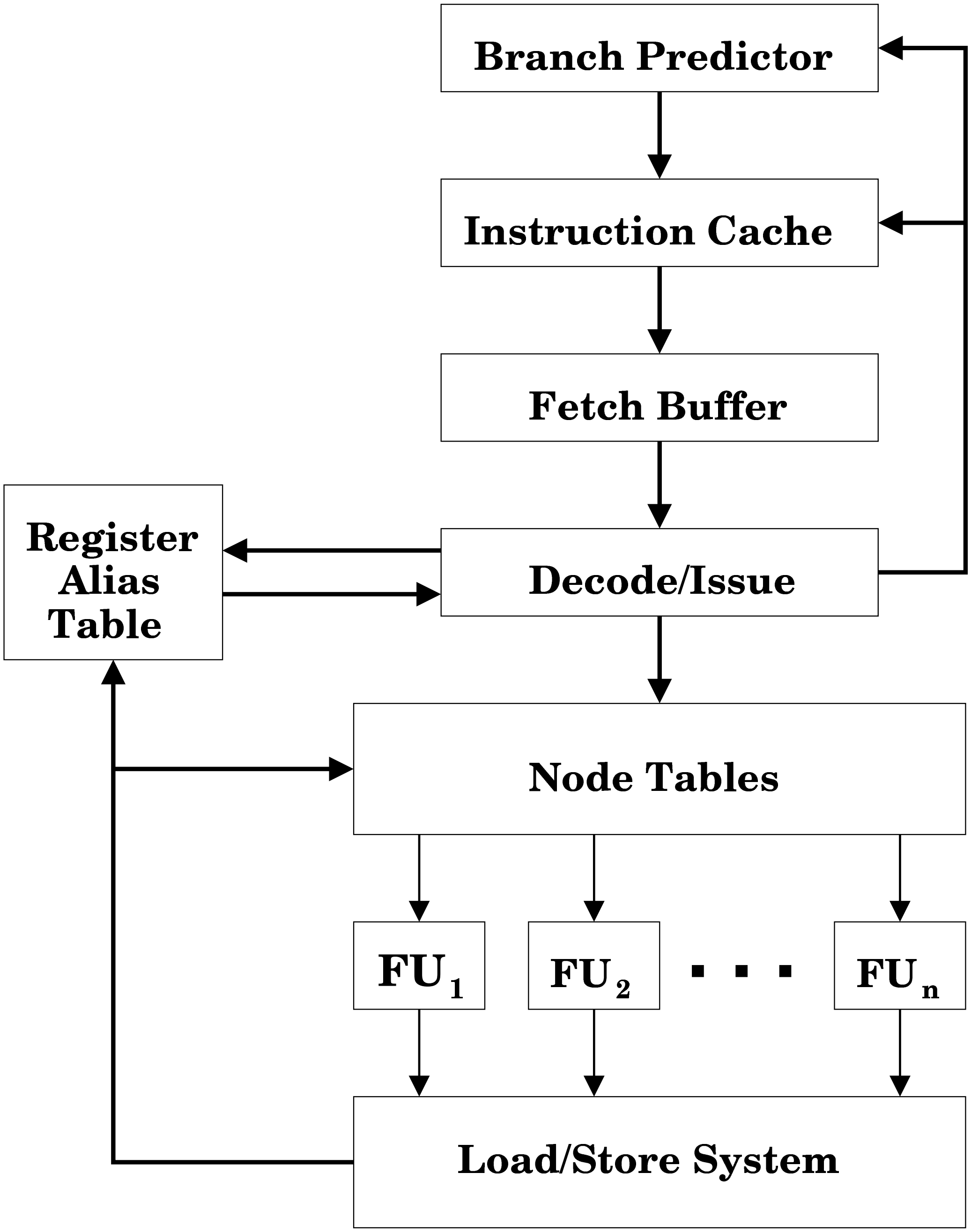


Figure 3.2: Basic HPS Microarchitecture

The fetch buffer supplies the dynamic instruction stream required for RDF execution. Each cycle, instructions are removed from the fetch buffer and decoded into data flow nodes. Using the information in the Register Alias Table, a generalized version of Tomasulo's algorithm [122] renames the registers. The instructions are then issued into the Node Tables; that is, their associated nodes are installed in the Node Tables. The Node Tables, which are more commonly known as reservation stations, house the active window. Associated with each node in the Node Tables are the source operands for that node (or identifiers for obtaining the operands), and destination information. When all of a node's source operands become available, the node becomes eligible for firing. Each cycle, a subset of the firable nodes are scheduled for execution on one of the functional units. After execution, the node results are distributed to the Register Alias Table and to the other nodes in the Node Tables awaiting those results.

Loads and stores require special consideration. The physical addresses of load and store memory accesses are not known when the loads and stores are issued into the Node Tables. Thus, unlike register dependencies, which are dealt with before the instructions are issued into the Node Tables via register renaming, the dependencies between loads and stores must be dealt with after the instructions have been issued.

In Figure 3.2, the functional units compute the logical addresses of load and store memory accesses and then deliver these addresses to the load/store system. The addresses are not necessarily computed and delivered to the load/store system in program order. The delivery of store data to the load/store system is (or can be) decoupled from the delivery of the addresses. That is, a store is (or can be) split into two data flow nodes. The first node computes the logical address of the memory access. The second node delivers the store data.

The load/store system is responsible for obtaining the correct data for loads and for updating the architectural state whenever stores retire. It contains a translation lookaside buffer (TLB), a memory disambiguator, and a data cache. The TLB converts the logical addresses of the load and store memory accesses into physical addresses. The disambiguator uses these physical addresses to determine which loads are dependent on stores. Dependent loads have their data forwarded from the proper store. The other loads receive their data from the data cache. Note that the disambiguator may not receive the physical addresses of load and store memory accesses in program order. This complicates dependency analysis in that, at any given point in time, the disambiguator may not have all the addresses required to correctly identify the dependencies between loads and stores. That is, for a particular load, the disambiguator may not know the addresses of all potentially aliasing stores. The data cache is never updated speculatively. It is only updated when stores retire. The store data delivered to the load/store system is buffered until the stores retire, at which point the data is written into the cache.

3.2 Simulators

I built three different simulators for studying computer architecture. Each simulates the execution of programs written for the Alpha AXP instruction set architecture (ISA) [112]. All of them are stand-alone execution driven simulators. That is, each simulator reads in the simulated program's executable image and any associated input data, initializes the processor model (if there is one), and then, instruction by instruction, calculates all of the program's results.

To hasten development time, I first built a basic foundation that can be used to build many different simulators. The three simulators were then built using this foundation. This foundation includes the simulation of process images, the simulation of Alpha ISA instructions, and the emulation of DEC OSF/1 system calls. The DEC OSF/1 system calls are emulated via proxy system calls on the host machine. (The host machine is the machine the simulation runs on.) The simulators do not actually simulate the code of the system call. Special function calls, called hooks, are used to interface the simulators to this basic foundation.

The three different simulators are the functional simulator, the RDF simulator, and the full simulator. The functional simulator does not model a processor. The RDF and full simulators do. The full simulator models the processor more accurately than the RDF simulator. However, it does so at the expense of simulation cost (run time as well as development time). The RDF and full simulators determine the execution time of a program by simulating its execution on the processor model. They can model many different processor configurations. The processor configuration is specified via command line options.

3.2.1 The Functional Simulator

The functional simulator is the simplest and fastest simulator. This simulator simply executes the instructions. No timing information is recorded. The functional simulator is used for creating instruction traces, for performing trace sampling, and for recovering the state associated with a killed simulation. (In the event that a naive or malicious user kills off a simulation running using the full simulator on the Computer Aided Engineering Network, the functional simulator is used to quickly recover the state of the simulation to the point just before the simulation was killed. Upon reaching this point, the full simulator is activated and the simulation continues on as if it had never been killed.)

3.2.2 The RDF Simulator

Originally, the RDF simulator simulated instructions using the abstract RDF model of execution discussed in Section 3.1. Three parameters characterize this model: window size, issue rate, and the latencies required for executing each type of data flow node. Later, the simulator was modified so that it could model a more realistic processor. The simulator was modified so that a dispatch rate and retire rate could be specified. (The dispatch rate is the maximum number of data flow nodes that can be scheduled for execution on functional units in a single cycle. The retire rate is the maximum number of instructions that can be retired in a single cycle.) Additionally, the simulator was modified to include a real data cache, a real instruction cache, and a real branch predictor.

The RDF simulator behaves like a trace driven simulator. It processes each instruction in the dynamic instruction stream one by one. It does not begin processing a new instruction until it is done with the old one. It only sees the instructions executed along the correctly predicted path. Because of this, it cannot correctly model the effects of fetching, issuing, and executing instructions along a mispredicted path. It does, however, correctly model the penalty due to a mispredicted branch. As branches are encountered in the dynamic instruction stream, a prediction is made. This prediction is compared to the real outcome of the branch. If a prediction fails, issue is stalled until the branch is resolved and instructions from the correct path are available for issuing. Note that during the time of this stall, a real machine would actually issue instructions from the mispredicted path. The RDF simulator mimics the effect of not issuing useful work by performing this stall.

Compared to the full simulator, the RDF simulator is simpler, easier to modify, and runs quicker. However, it does not model as much as the full simulator does, so it is not as accurate. This simulator is typically used as a proving ground for new microarchitectural ideas. Ideas that pass muster are then implemented in and tested on the full simulator. Unfortunately, some ideas cannot be implemented in the RDF simulator and must be implemented in the full simulator.

3.2.3 The Full Simulator

The full simulator performs a cycle by cycle simulation of the executable using the HPS microarchitecture presented in Section 3.1. This simulator is much more complex, harder to modify, and slower than the RDF simulator. It also simulates the processor with a much greater level of detail than is possible with the RDF simulator, so it is more accurate. This simulator models a split transaction bus, two levels of instruction cache, two levels of data cache, banking of the first level data cache, a memory disambiguator, a branch predictor, a fetch buffer, a Register Alias Table, reservation stations, functional units, and precise exceptions via checkpointing [53]. Most of the processor pipeline lengths are programmable. For example, the length of the instruction decode pipe can be specified with a command line option.

Unlike the RDF simulator, when the full simulator mispredicts a branch, it will proceed with fetching, issuing, and executing instructions along the mispredicted path. When the processor discovers that the prediction is incorrect (as indicated by the execution of the corresponding branch instruction), the simulator flushes the speculative instructions and then proceeds along the correct path.

3.3 Benchmarks

The results presented in this dissertation are for the eight SPECint95 benchmarks [119] and for eight other common UNIX programs, which I will refer to as the Non-SPEC benchmarks. Table 3.1 lists the SPEC benchmarks along with their training and test data sets. Table 3.2 does likewise for the Non-SPEC benchmarks. All benchmarks are written in C, except for groff, which is written in C++. The training sets are used to generate the benchmark profiles for the experiments that require profiling. The test sets are used to generate all of the performance numbers presented in this dissertation. For the SPEC benchmarks, the training and test sets are either the data sets provided by SPEC or modified versions of those data sets. A modified data set is used whenever the running time of the unmodified version is too long.

Benchmark	Description	Training Set	Test Set
cmp	data compression program	35KB.in*	30KB.in*
gcc	GNU C compiler	dbxout.i	jump.i
go	Go-playing program	olaf.in*	2stone9.in*
ijpeg	image compression program	vigo.ppm*	specmun.ppm*
li	XLISP interpreter	queens.lsp*	train.lsp
m88k	88100 microprocessor simulator	dhry.test.lit	dcrand.train.lit
perl	Perl programming language interpreter	primes.pl*	scrabbl.pl†
vortex	object-oriented database	35M.lit*	230M.lit*

*The data set is a modified version of one of the SPECint95 data sets.

†This data set is from the SPECint95 training data set for perl.

Table 3.1: The SPECint95 benchmarks and their training and test data sets

Benchmark	Description	Training Set	Test Set
chess	GNU Chess	train.in	sim.in
groff	GNU groff document formatting system	troff.1	gcc.1
gs	Aladdin Ghostscript interpreter	graph.ps	sigmetrics94.ps
pgp	Pretty Good Privacy encryption system	tasuki1.jpg	IJPP97.ps
plot	gnuplot plotting program	singulr.dem	surface2.dem
python	Python programming language interpreter	morse.py	yarn.tests.py
ss	SimpleScalar superscalar processor simulator	random	fmath-little
tex	TEX document formatting system	slide-root.tex	PACT96.tex

Table 3.2: The Non-SPEC benchmarks and their training and test data sets

I downloaded the source code for the Non-SPEC benchmarks over the Internet and then modified the source code so that the programs could be used as benchmarks. Here is one example of such a modification. The pgp program uses the current system time to seed its random number generator. The encryption algorithm in pgp uses numbers generated by this random number generator to encrypt a file. If file X is encrypted at times A and B, the output of the encryption at time A will be different from the output of the encryption at time B. Each time pgp is run, its behavior is guaranteed to be different from the last time it was run. This variability makes it impossible to perform fair comparisons between the results generated from two different simulations that use pgp as a benchmark. To make fair comparisons possible, I eliminated the variability. That is, I modified the source code so that the random number generator was seeded with a fixed time instead of the current system time.

For each of the sixteen benchmarks, I created two versions of the benchmark executable. The first version was created without the aid of profile information. The second version was created with the aid of profile information.

The first version was created by simply compiling the benchmark. All the benchmarks were compiled for the Alpha AXP instruction set architecture [112]. The groff benchmark was compiled with version 2.7.2 of the GNU C++ compiler using the following optimization flag: `-O3`. The remaining fifteen benchmarks were compiled with version 3.11 of the DEC OSF/1 AXP Compiler using the following optimization flags: `-O2 -Olimit 3000`. All benchmarks were statically linked.

The second version was created by rearranging the procedures in the first version of the executable. The procedures were rearranged so that the program would use the instruction cache more efficiently. To create the second version of each executable, the following steps were taken:

1. Profiling code was added to the first version of the executable using `pixie` [28]. `pixie` reads an executable, partitions it into basic blocks, and then writes an equivalent executable containing additional code that counts the execution of each basic block.

2. The executable created by `pixie` was run using the training data set. This produced a file containing the number of times each basic block in the program was executed.

3. The file produced by step 2 was analyzed using `prof` [28]. `prof` estimates the amount of time spent executing each procedure in the program. (For each basic block, `prof` estimates the time required to execute that block once and multiplies that time by the number of times the block was executed. This yields, for each basic block, the [estimated] total amount of time spent executing that block. The time per procedure is obtained by summing the total amount of time per block over all the basic blocks in the procedure.)

4. Using `cord` [28] and the time estimates produced by step 3, the second version of the executable was created by rearranging the procedures in the first version. `cord` reads an executable, and, using the time estimates, writes an equivalent executable with the procedures arranged in order of procedure density. The procedure density is the total amount of time spent in a procedure divided by the number of instructions in that procedure.

There are a total of 32 benchmark executables: sixteen that are first versions, and sixteen that are second versions. Throughout this dissertation, I will refer to the sixteen executables that are first versions as the (benchmark) executables without profiling. I will refer to the sixteen executables that are second versions as the (benchmark) executables with profiling.

For all experiments, the benchmarks were run to completion. For each benchmark, Table 3.3 shows the number of instructions required to run the benchmark to completion for both the training and test data sets. These numbers do not depend on the version (i. e., with or without profiling) of the benchmark executable.

Benchmark	Training Set	Test Set
cmp	173 M	150 M
gcc	194 M	182 M
go	142 M	137 M
ijpeg	220 M	334 M
li	276 M	248 M
m88k	543 M	145 M
perl	217 M	46 M
vortex	41 M	252 M
chess	180 M	250 M
groff	152 M	238 M
gs	98 M	228 M
pgp	100 M	187 M
plot	225 M	270 M
python	93 M	276 M
ss	183 M	128 M
tex	138 M	242 M

Table 3.3: The number of instructions executed per benchmark

CHAPTER 4

Available Instruction Level Parallelism

In this chapter, I measure the amount of parallelism available in a single instruction stream, and the upper bound on the amount of parallelism that can be exploited by practical processor implementations. The information provided in this chapter can be used by the architects of future processors to make design decisions. For example, this information can be used to answer the questions: Is there enough available instruction level parallelism to justify building a processor that can issue sixteen instructions per cycle? And, how many reservation stations should a sixteen wide issue processor have? The information will show that building a processor that can issue sixteen instructions per cycle and that has 1024 reservation stations is a worthwhile endeavor. This processor will be used exclusively in the following chapters of the dissertation.

To calculate the upper bound on the amount of parallelism that can be exploited by practical processor implementations, I will use the Restricted Data Flow (RDF) model of execution [14, 100, 101]. In this execution model, the exploitation of instruction level parallelism is only limited by the flow dependencies in a program, the amount of buffering (e. g., the number of reservation stations or the number of reorder buffer entries) the machine supports, and the machine's instruction fetch (and instruction decode) bandwidth. I will also use a variant of the RDF model, the Unrestricted Data Flow (UDF) model [14], to measure the total amount of parallelism available in a single instruction stream. The UDF model is simply an RDF model that supports an infinite amount of buffering and that has infinite instruction fetch bandwidth. I will determine the performance, in Instructions Per Cycle (IPC), of the RDF model for many different machine configurations, where each configuration specifies the amount of instruction buffering the machine supports and the machine's instruction fetch bandwidth.

This chapter is organized into five sections. Section 4.1 formally describes the RDF model of execution. Section 4.2 describes the problem caused by dependencies communicated via memory, and some possible solutions. Section 4.3 describes the assumptions and configuration parameters that are common to all RDF machines modeled in this dissertation. Section 4.4 evaluates the performance of the RDF model for many different machine configurations. Section 4.5 summarizes the chapter.

4.1 The RDF Model of Execution

To measure the available parallelism, I will use an abstract model of execution that does not unnecessarily restrict the exploitation of instruction level parallelism. This model is called the Restricted Data Flow (RDF) model of execution [14, 100, 101]. The RDF model of execution is characterized by three parameters: window size, issue rate, and instruction class latencies.

Processing consists of issuing instructions from a program's dynamic instruction stream, converting those instructions into a dynamic data flow graph, scheduling instructions for execution when their flow dependencies have been resolved, and retiring those instructions after execution has completed. The dynamic instruction stream originates from a perfect (100 percent hit rate) instruction cache and is created by an omniscient branch predictor that always knows the direction a branch will take. Instructions are issued and retired in the order they appear in the dynamic instruction stream.

The term *active window* will be used to refer to the set of instructions whose corresponding data flow nodes are resident in the RDF machine. The window size specifies the maximum number of instructions that can be in the active window at any instant in time. Instructions can be issued as long as the number of instructions in the active window is less than the window size. The issue rate is the maximum number of instructions that can be removed from the dynamic instruction stream and entered into the active window in a single cycle. The instruction class latencies specify the set of operations and the latency associated with each operation. In the RDF model, there is never contention among instructions for functional units, each functional unit can perform every desired operation, and the latency associated with each operation is specified.

Although this execution model is abstract, it accounts for many practical constraints on CPU design. The restriction on window size accounts for the constraint on the amount of buffering. To be more concrete, in current processors, the window size accounts for the maximum number of instructions that the reservation stations (or the equivalent microarchitectural analogue) can hold. The restriction on issue rate accounts for the constraint on the amount of instruction fetch (and instruction decode) bandwidth.

To measure the total amount of parallelism available in a single instruction stream, I will use a variant of the RDF model called the Unrestricted Data Flow (UDF) model. The UDF model is an RDF model with an infinite window size and an infinite issue rate.

4.2 Memory Dependency Handling

The RDF model of execution relies on dynamically reordering code in order to harvest the available instruction level parallelism (ILP). Its goal is to execute as many instructions in parallel as possible, while, at the same time, preserving the sequential semantics of the instruction stream. To accomplish this goal, all anti and output dependencies must be eliminated, and true (flow) dependencies must be enforced. Additionally, for flow dependencies, the passing of values from producers to consumers must be timely. Dependencies are communicated between instructions either through registers or through memory.

The anti and output dependencies communicated between instructions via registers are eliminated using register renaming. The registers are renamed before they are issued into the active window. Once an instruction is inside the active window, the only remaining dependencies (communicated via registers) are true (flow) dependencies. For these flow dependencies, Tomasulo's algorithm [122] is used to guarantee the timely passing of values from producers to consumers.

Ideally, the RDF machine would handle the dependencies communicated via memory as effectively as it handles the dependencies communicated via registers. Unfortunately, the unknown address problem [101]—also known as the memory disambiguation problem [94]—prevents this. The physical addresses of load and store memory accesses are unknown when the loads and stores are issued into the active window. These addresses are needed to determine the dependencies between loads and stores. Thus, unlike register dependencies, which are dealt with before the instructions are issued into the active window via register renaming, the dependencies between loads and stores must be dealt with after the instructions have been issued.

4.2.1 The Unknown Address Problem

Consider the example in Figure 4.1. The instructions inside the box represent the instructions inside the active window that are either waiting for their dependencies to be resolved, or are waiting to be sent to functional units. A letter, for example the ‘a’ associated with instruction I3, represents a known address; that is, an address that the RDF machine has already computed a value for. The instruction at the top of the box, I1, is the oldest instruction in the active window. The oldest instruction is the instruction that would have executed first on a processor that executes instructions in program order. The instruction at the bottom of the box, I3, represents the youngest instruction.

```
I1:   idiv    r6,r1,r2
I2:   store   r1,(r6)
I3:   load    r2,a
```

Figure 4.1: The Unknown Address Problem Example

An RDF machine tries to execute as many instructions in parallel as possible. Suppose all three instructions are issued into the RDF machine during the same cycle. The integer divide (I1)—divide r1 by r2 and put the result into r6—has all its operands available and it begins executing immediately. The divide will not finish executing until many cycles later. The store (I2)—store r1 into the memory address specified by r6—uses the result of the divide to perform an indirect store. Thus, the address to which the store is performed is unknown until the divide finishes. The load (I3)—load the value stored in memory address ‘a’ and put the result into r2—also has all its register operands available when it is issued into the RDF machine. It could begin executing immediately. Should the RDF machine allow this load to begin executing immediately?

The RDF machine does not know whether this load is dependent on the store because the address of the store is unknown. If the RDF machine allows the load to begin executing immediately, the sequential semantics of the instruction stream are only preserved if the address of the store is an address other than 'a'. If the address of the store happens to be the address 'a', a flow dependency is violated. The RDF machine could force the load to wait until the address of the store is known. Once the address of the store has been computed, the RDF machine can know for certain whether or not the load is dependent on the store, and can make the proper decision as to when to execute the load. Unfortunately, if the address of the store is not 'a', the load will have been unnecessarily delayed.

4.2.2 Memory Disambiguation Paradigms

Memory disambiguation is used to solve the unknown address problem. There are many different ways of solving the unknown address problem. As a result, there are many different memory disambiguation techniques. Memory disambiguation techniques follow one of two basic paradigms: either the non-speculative paradigm or the speculative paradigm. Techniques in the non-speculative paradigm do not speculate on the dependencies between loads and stores. The techniques in the speculative paradigm do.

Non-Speculative Paradigm

Techniques in the non-speculative paradigm never distribute the result of a load until they know for certain whether or not the load is dependent on a store in the active window. Loads are forced to wait until the addresses of older stores have been computed. Once the dependencies are known for certain, the load is allowed to distribute its result. The advantage of using techniques in this paradigm is that, when a load distributes its result, that result is guaranteed to contain the correct data. The disadvantage: some loads are forced to wait for stores they are not dependent on, which needlessly delays the distribution of their results.

Consider the example from Figure 4.1 again. A copy of this figure is also provided in Figure 4.2. Assume that the machine implements a memory disambiguation technique that follows the non-speculative paradigm. The load can begin executing at the same time as the divide. Its address can be calculated and its data can be (speculatively) fetched from the data cache. The result of the load, however, cannot be distributed until it is known for certain whether or not the load is dependent on the store. As a result, the load will not be able to distribute its data until after the divide finishes. Once the divide finishes, the machine will know for certain whether or not the load is dependent on the store. If the load is not dependent, it will distribute the data that was fetched from the data cache. If it is dependent, it will distribute data forwarded from the store.

```
I1:   idiv    r6,r1,r2
I2:   store   r1,(r6)
I3:   load    r2,a
```

Figure 4.2: Non-Speculative Memory Disambiguation Paradigm Example

Several memory disambiguation techniques follow this paradigm. The simplest technique forces all loads and stores to be executed in strict program order. That is, a load or store is not allowed to be sent to a functional unit until all older loads and stores have been sent to functional units. Instructions that are not loads or stores are still allowed to execute out-of-order. A slightly more aggressive technique lets some of the loads execute out-of-order. Stores are forced to execute in strict program order. Loads are executed in-order with respect to stores, but out-of-order with respect to other loads. The most aggressive technique that fits this paradigm is described algorithmically by Patt et al. [101].

Patt et al. [101] also describe another technique that fits this paradigm, which I will call *dependency matrix*.[1] I will use this memory disambiguation technique for some of the experiments in this dissertation. With this technique, the order in which loads and stores *begin* execution is completely unconstrained. Regardless of whether or not the addresses of all potentially aliasing stores are known, when the operands needed to calculate the address of a load are available, the load is sent to a functional unit, its address is calculated, and its data is obtained from the data cache. The only constraint is that a load may not distribute its result (i. e., *finish* execution) until the addresses of all previous stores have

[1] The technique is called dependency matrix because a dependency matrix is used to keep track of which stores have unknown addresses, and which loads are allowed to distribute their results.

been computed.

Speculative Paradigm

When it is uncertain whether or not a load is dependent on a store in the active window, techniques in the speculative paradigm will predict whether or not the load is dependent. The prediction might only provide a simple yes or no answer to the question: Is the load dependent on any stores in the active window? If a more complex predictor is used, the prediction might also provide the answer to the question: *Which* store in the active window is the load dependent on?

If a load is predicted to be independent of any stores, it may distribute its result before its dependencies have been fully resolved. If a load is predicted to be dependent, the aliasing store must be identified. For the simple prediction, the load's address and the addresses of older stores—all of which must first be computed by functional units—are used to identify the aliasing store. For the complex prediction, the prediction itself identifies the aliasing store. Once the aliasing store has been identified, the store data (when available) is forwarded to the load. The load then distributes its result. Note that for the complex prediction, the aliasing store is identified without the aid of the store's address or the load's address. As a result, the store data can be forwarded and the load result distributed even if one or both of these addresses are unknown.

The prediction is verified when all of the load's dependencies have been resolved. If the prediction was correct, no further action is required. If the prediction was incorrect, there are two cases to consider. First, the load has not yet distributed its result. Since all the load's dependencies are known at this point, the machine can determine exactly where the load's data is (or will be) stored. The machine simply locates this data and distributes it. The only real harm done in this case is that the load's result distribution was (possibly) delayed longer than it needed to be. For the second case, the load has already distributed its result. Since the prediction was incorrect, the load probably distributed incorrect data. Instructions that are dependent on the load may have executed using this incorrect data, producing and distributing still more incorrect data. Recovering from this mispredict may be trickier. The machine must re-execute the load and any instructions that executed using incorrect data.

The advantage of using techniques in this paradigm is that, assuming the predictions are mostly correct, loads are rarely forced to wait for stores they are not dependent on. As a result, the distribution of load results is almost never needlessly delayed. The disadvantage, of course, is that the predictions are sometimes wrong. And when a prediction is wrong, a load may distribute incorrect data.

Consider the example from Figure 4.1 again. A copy of this figure is also provided in Figure 4.3. Assume that the machine implements a memory disambiguation technique that follows the speculative paradigm. Also assume that the machine predicts that the load is not dependent on the store. The load begins executing at the same time as the divide. Its address is calculated, its data is fetched from the data cache, and its result is distributed—all before the divide finishes executing. Later, when the divide finishes, its result is compared to the address ‘a’. If the result is not ‘a’, no dependencies were violated, so no further action is required. If the result is ‘a’, then the load is dependent on the store, and a flow dependency was violated. To recover, the load must distribute the correct value for its result. In addition, any instructions either directly dependent or indirectly dependent on the load that executed using incorrect data must be re-executed using the correct data.

```
I1:   idiv    r6,r1,r2
I2:   store   r1,(r6)
I3:   load    r2,a
```

Figure 4.3: Speculative Memory Disambiguation Paradigm Example

Several memory disambiguation techniques follow this paradigm. Techniques that use the simple yes/no prediction include blind, or naive, speculation and Store Sets [22]. Techniques that use the more complex type of prediction (i. e., not only yes/no but also *which* store) have been proposed by Moshovos et al. [91, 92] and by Tyson and Austin [126]. Blind speculation is the simplest of all of these techniques. When it is uncertain whether or not a load is dependent on a store in the active window, this technique *always* predicts that the load is independent. A mechanism for implementing blind speculation, called the Address Resolution Buffer, has been proposed by Franklin and Sohi [37].

For experiments in this chapter and the next two, I will use two memory disambiguation techniques that follow the speculative paradigm: *simple oracle* and *ComplexOracle*. simple oracle uses the simple yes/no predictions and ComplexOracle uses the more complex predictions. Both techniques obtain their predictions from oracles that always provide correct predictions. These oracles can't be built, so neither technique can actually be implemented in hardware. I use these two techniques because they provide upper bounds on performance: simple oracle provides the upper bound for techniques that use the simple predictions, and complex oracle provides the upper bound for techniques that use the more complex predictions.

For both techniques, the order in which loads and stores begin execution is completely unconstrained. Regardless of whether or not the addresses of all potentially aliasing stores are known, when the operands needed to calculate the address of a load are available, the load is sent to a functional unit and its address is calculated. If the oracle predicts that the load is independent of any stores, the load result is obtained from the data cache and immediately distributed to dependent instructions. If the oracle predicts that the load is dependent on a store, the store's data is forwarded to the load. For simple oracle, the forward does not occur until the load's address, the store's address, and, of course, the store's data are known.[2] For ComplexOracle, the load's address and the store's address are not needed to perform the forward. For ComplexOracle, only the store's data needs to be known in order to perform the forward.

[2] For simple oracle, a load distributes its result before all its dependencies have been resolved. Each load keeps track of who it thinks is the aliasing store. As the store addresses are calculated, they are compared to the load's address. Each time the load detects a new aliasing store, the store data is forwarded to the load, and the load distributes this new data. Because of this, a load may distribute its data multiple times. The initial distributions will all contain bogus data. The final distribution will occur just after the actual flow dependency is detected. This distribution will provide the correct data.

4.2.3 Unified versus Split Stores

In the RDF model of execution, instructions are removed from the program's dynamic instruction stream, converted into data flow nodes, and then issued into the active window. Each instruction is converted into zero or more nodes. For the Alpha AXP instruction set architecture (ISA) [112], most instructions are converted into a single node. Store instructions, however, may be converted into either one or two nodes. If the RDF machine converts stores into single nodes, the machine implements *unified stores.* If the machine converts every store into two nodes, the machine implements *split stores.* The machine will implement either unified stores or split stores, but not both. There are advantages and disadvantages for implementing unified stores, and for implementing split stores.

If unified stores are implemented, each store is converted into a single data flow node. This node is then issued into the active window. Note that a store has one set of source operands that are used to compute the address of the store memory access, and another set for specifying the data to be stored at that address. The node waits in the active window until *all* of the store's source operands become available, at which point it is sent to a functional unit. The functional unit computes the address of the store memory access. After computing the address, the functional unit passes the address and the store data to the load/store system. The load/store system uses the address to determine which loads, if any, are dependent on the store. Dependent loads are forwarded the store data.

If split stores are implemented, each store is converted into two data flow nodes. Both nodes are then issued into the active window. The first node, the address calculation node, computes the address of the store memory access. The address calculation node is sent to a functional unit as soon as the set of operands required for computing the address are available. The functional unit computes the address and passes it to the load/store system. The second node, the data delivery node, simply delivers the store data to the load/store system. It waits for the operand(s) that contain the store data to become ready, and then passes these operand(s) to the load/store system. The execution of an address calculation node is completely decoupled from the execution of the corresponding data delivery node: either node can be executed first, or, both nodes can be executed at the same time.

One advantage of unified stores over split stores is that they are easier to implement. In particular, for unified stores, upkeep of the store instruction's state is accomplished by monitoring the actions of the store's single data flow node. (Store instruction state keeps track of such things as: Is the store a candidate for retirement?) For split stores, upkeep of the store instruction's state requires monitoring the actions of both the store's data flow nodes.

Another advantage of unified stores over split stores is that the forwarding of store data to dependent loads is simpler. The load/store system uses the addresses of load and store memory accesses to determine which loads are dependent on stores. If a load is found to be dependent on a store, the store data is forwarded as soon as that data becomes available. For unified stores, the address of the memory access and the store data are always delivered to the load/store system at the same time. Thus, the store data is always available when a dependency is detected, and the forward can always occur at the time the dependency is detected. For split stores, the address may be delivered to the load/store system before the store data. If the address is delivered before the data, the store data will not be available when a dependency is detected, so the forward will need to be delayed. To delay the forward, the RDF machine must have a mechanism to remember this dependency, and to recall it when the store data arrives.

The disadvantage of unified stores is that the address of a store's memory access cannot be computed until the operand(s) that contain the store data are ready. As a result, store address computations are delayed longer than they would be for split stores, and store addresses remain unknown for longer period of time. The performance of loads suffers if store addresses remain unknown for longer periods of time. For memory disambiguation techniques that follow the non-speculative paradigm, loads do not distribute their results until their dependencies are known. The store addresses are needed to determine these dependencies. Delaying the store address calculations will therefore delay the distribution of load results. For techniques in the speculative paradigm, loads can distribute their results before their dependencies are fully known. Delaying the calculation of store addresses will only increase the number of dependence predictions that are required, and delay the verification of those predictions. If, however, the predictions are always (or almost always) correct, delaying the store address calculations will have no (or little) impact on load performance.

For all the experiments in this dissertation, the machine will implement split stores. However, in Section 4.4, I will explore the interaction between memory disambiguation technique and store type (i. e., unified or split). I will look at three different disambiguation techniques: dependency matrix, simple oracle, and complex oracle. Only dependency matrix, which follows the non-speculative paradigm, is affected by store type. The other two techniques follow the speculative paradigm. For these techniques, the dependence predictions are always correct, so their performance is not affected by store type.

4.3 The RDF Model Configuration and Assumptions

Three parameters characterize the RDF model of execution: window size, issue rate, and instruction class latencies. The window size and issue rate will vary depending on the experiment. All experiments will use the instruction class latencies provided in Table 4.1. Loads require one cycle for address calculation, and, if a data cache access is necessary, one or more cycles for the cache access.[3]

Instructions	Execution Latency (cycles)
FP div	16
other FP	4
INT mul	8
load	1 + cache latency
all others	1

Table 4.1: Instruction Class Latencies

The RDF model also assumes the following:

- Register and memory renaming are performed. Renaming eliminates anti and output dependencies.

[3] It's possible to perform both the address calculation and the data cache access in a single cycle. Lynch, Lauterbach, and Chamdani [73] described how to perform a true addition using the decoder of the cache's RAM array. This allows the cache access to occur before the address calculation, resulting in a single cycle load.

- For machines that use the dependency matrix memory disambiguation technique, a load must wait until the addresses of all previous stores have been computed. I optimistically assume one cycle between when all these addresses have been computed and when the load may distribute its result. This cycle accounts for the time the machine requires to fully resolve the load's dependencies.

- A store forwards its data to a dependent load if and only if the store and the load are both resident in the active window. I assume the machine requires one cycle to bypass the store data to the dependent load. That is, for machines that use either the dependency matrix or the simple oracle memory disambiguation technique, there is one cycle between when the load's address, the store's address, and the store's data are known; and when the load may distribute its result. For machines that use the complex oracle technique, there is one cycle between when the store's data is known and when the load may distribute its result.

- When a serializing instruction is encountered, the simulated machine must stop issue, wait for all instructions currently in the active window to complete, and then execute the serializing instruction. Issue resumes once the serializing instruction completes. (A serializing instruction is an instruction that must be executed in order with respect to the other instructions in the program's dynamic instruction stream. A trap is an example of a serializing instruction.)

4.4 Experimental Results

This section evaluates the performance of the RDF model for many different machine configurations, where each configuration specifies the window size, issue rate, and memory disambiguation technique. All configurations modeled in this section have a perfect (100 percent hit rate) instruction cache, a perfect (omniscient) branch predictor, a perfect execution core (i. e., an execution core with an unbounded numbers of functional units, each of which can perform every desired operation), and a perfect single cycle data cache.

Each benchmark was simulated using a variety of machine configurations. The results for all the machines that used the most aggressive memory disambiguation technique, complex oracle, are provided in Figures 4.4–4.6. In each graph, the performance, in Instructions Per Cycle (IPC), is listed on the vertical axis. The window size, in instructions, is listed on the horizontal axis. A window of infinite size is indicated by an infinity (∞). There is a single line for each issue rate. The line corresponding to an infinite issue rate is indicated by an infinity (∞) in the legend. An issue rate of X indicates that the machine can issue X instructions per cycle, regardless of their instruction class, so long as the window is not full and no serializing instructions are encountered.

In some cases, the performance of the machines with infinite issue rates exceeded 64 IPC. For these cases, the performance (rounded to the nearest integer) is listed at the top of the graph. For example, for the go benchmark (see Figure 4.5), the performances of the machines with infinite issue rates exceeded 64 IPC at window sizes of 8192, 16384, and infinity. The performances of the machines at these three window sizes are 64 IPC, 70 IPC, and 90 IPC, respectively.

Only machines with finite window sizes can be built. For a machine with a finite window size and an infinite issue rate, issue stalls when the window is full. This limits the maximum number of instructions that can be issued each cycle to the size of the window. Thus, these machines can be built if it becomes possible to issue a window's worth of instructions in one cycle. I simulated these machines because (a) I believe that one day they can be built, and (b) I wanted to provide upper bounds on the performances of machines that exploit all the ILP within fixed sized windows. The machines with infinite window sizes and finite issue rates were simulated to provide upper bounds on the performances of machines that exploit ILP at fixed issue rates. The machine with an infinite window size and an infinite issue rate is a UDF machine. The performance of this machine indicates the total amount of instruction level parallelism in the benchmark.

Figure 4.4 shows the performance averaged over all the benchmarks, both SPEC and Non-SPEC. At finite window sizes, the line corresponding to the infinite issue rate hugs the line corresponding to the issue rate of 64. Even at an infinite window size, these two lines are fairly close: only 3 IPC separates them. This indicates that an issue rate of 64 is sufficient to exploit all the ILP. Note that if the issue rate is high enough, the machine's performance is strongly dependent on the size of its window. Also note that at a given issue rate, the difference in performance between a machine with a window size of 16384 and a machine with an infinite window size is small. This indicates that machines with small windows (relative to the number of instructions in a program's dynamic instruction stream) can exploit almost all the available ILP.

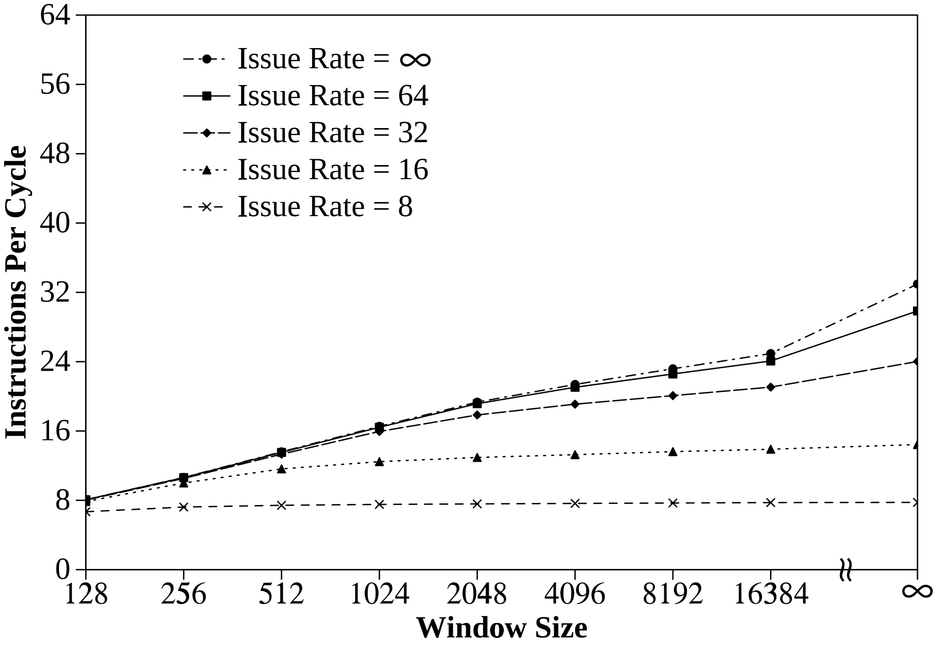


Figure 4.4: Instruction Level Parallelism—Harmonic Average

Overall, these results indicate that there is a significant amount of ILP. This finding is consistent with that of previous studies [7, 14, 128]. At an issue rate of 8, the performance approaches 8 IPC as the window size is increased. For this issue rate, when the window size is greater than or equal to 256 instructions, the IPC is greater than 7. It is likely that machines that can issue 8 instructions per cycle will be built in the near future. At issue rates of 16 and 32, the performance approaches 14 and 24 IPC, respectively. This is probably enough *parallelism* to justify building a machine that can issue 16 or 32 instructions per cycle. Of course, certain architectural and implementation problems must first be solved in order to fully justify the building of such machines. For example, a solution to the instruction cache bottleneck will be needed. At an issue rate of 64, the performance never exceeds 30 IPC. Future programmers and compilers may produce code that exposes more parallelism, raising this upper bound. Additionally, future workloads may contain more applications with large amounts of parallelism. However, unless either or these occurs, there is probably not enough parallelism to justify building a (general purpose) processor that can issue 64 instructions per cycle. Finally, the performance of the UDF model is 33 IPC. This is the average total amount of ILP for the benchmarks tested.

Figure 4.5 shows the results for the SPEC benchmarks and Figure 4.6 shows the results for the Non-SPEC benchmarks. The pgp and plot benchmarks contain very little ILP. The chess, go, ijpeg, m88k, and perl benchmarks, on the other hand, all contain large amounts of ILP. For the go, ijpeg, and perl benchmarks, the machines with small window sizes (i. e., window sizes no bigger than 16384 instructions) can exploit significant amounts of this ILP, perhaps even enough to justify building a machine that can issue 64 instructions per cycle. For the chess and m88k benchmarks, the machines with small window sizes cannot exploit significant amounts of this ILP. These benchmarks have large amounts of medium-grain (procedure level or subprogram level) parallelism that cannot be exploited unless the machine has a large window.

Serializing instructions can significantly limit the amount of ILP. When a serializing instruction is encountered, the machine must stop issue, wait for all instructions currently in the active window to complete, and then execute the serializing instruction. Issue resumes once the serializing instruction completes. As the window size and issue rate increases, the number of cycles that the machine stalls issue due to serializing instructions becomes a larger fraction of the total execution time of the program. For the plot benchmark, at an issue rate of 64 and a window size of 16384, instructions are not issued in 78% of the cycles due to serializing instructions. This severely limits the performance of machines when executing the plot benchmark. The remaining benchmarks are not significantly affected by issue stalls due to serializing instructions. For the gs benchmark, at an issue rate of 64 and a window size of 16384, instructions are not issued in 18% of the cycles due to serializing instructions. For the vortex benchmark, the fraction is only 13%. For all the remaining benchmarks, the fraction is less than 10%.

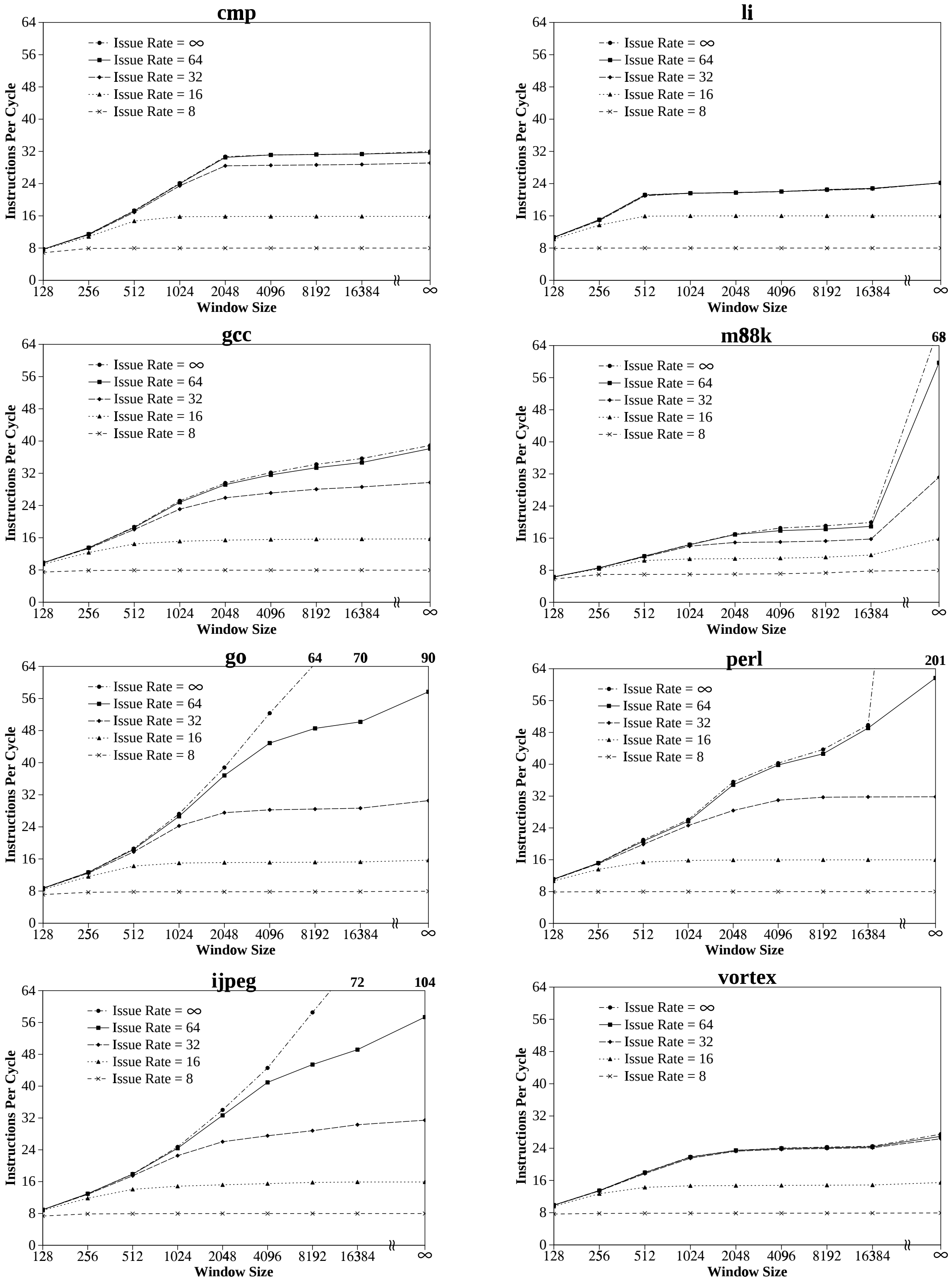


Figure 4.5: Instruction Level Parallelism—SPEC Benchmarks

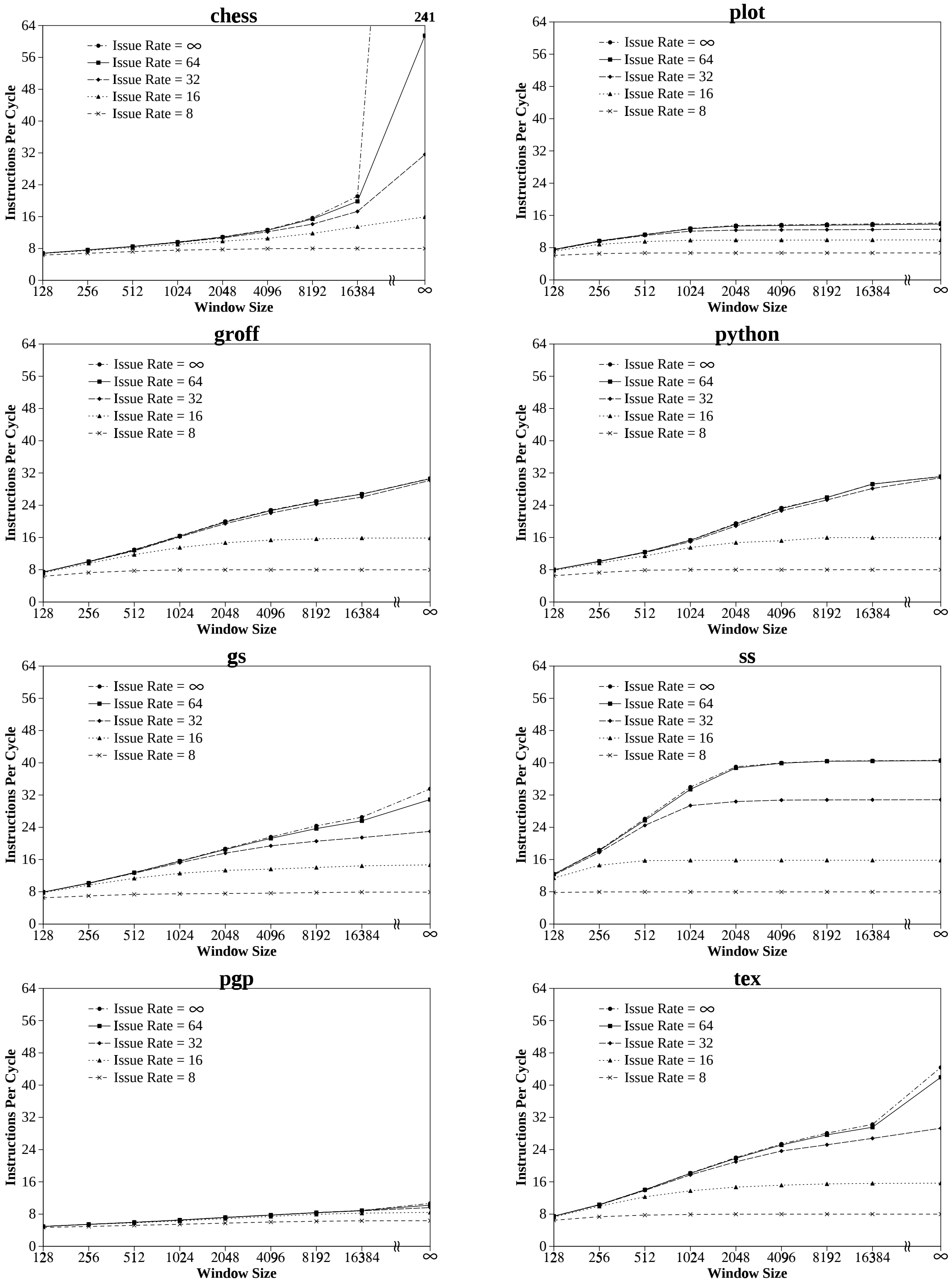


Figure 4.6: Instruction Level Parallelism—Non-SPEC Benchmarks

Figure 4.7 shows the results for all memory disambiguation techniques. This figure is for the average. Results for the individual benchmarks are provided in Figures A.1–A.16 of Appendix A. Each figure contains five graphs, with one graph for each issue rate. In each graph the performance is listed on the vertical axis. The window size is listed on the horizontal axis. There is one line for the complex oracle memory disambiguation technique, one line for the simple oracle memory disambiguation technique, and two lines for the dependency matrix memory disambiguation technique. The complex oracle and simple oracle memory disambiguation techniques are not affected by store type (i. e., unified or split). All machines that use these techniques implement split stores. The dependency matrix memory disambiguation technique is affected by store type. Consequently, there are two lines for this technique. The line labeled "Matrix (Split Stores)" is for machines that use dependency matrix and implement split stores. The line labeled "Matrix (Unified Stores)" is for machines that use dependency matrix and implement unified stores.

In some cases, the performances of the machines with infinite issue rates that used complex oracle and simple oracle exceeded 64 IPC. Whenever this occurred, the performances (rounded to the nearest integer) of the machine using complex oracle and the machine using simple oracle are listed at the top of the graph. The performance of the machine using complex oracle is always listed above the performance of the machine using simple oracle. For example, for the go benchmark (see Figure A.3), the performances of the machines with infinite issue rates exceeded 64 IPC at window sizes of 8192, 16384, and infinity. The performances of the machines that used complex oracle at these three window sizes are 64 IPC, 70 IPC, and 90 IPC, respectively. The performances of the machines that used simple oracle are 53 IPC, 60 IPC, and 81 IPC.

Figure 4.7 shows the performance averaged over all the benchmarks. Machines that used the memory disambiguation techniques in the speculative paradigm (i. e., complex oracle and simple oracle) performed significantly better than the machines that used the memory disambiguation technique in the non-speculative paradigm (i. e., dependency matrix) for issue rates of 16 and above. These machines perform better because loads are never forced to wait for the addresses of older stores to be computed. For the UDF machine that uses dependency matrix and implements split stores, 37% of the loads delay their result distributions because there are unknown store addresses. The performance of this machine is only 11 IPC. Note that this is the upper bound on the performance of *any* machine that

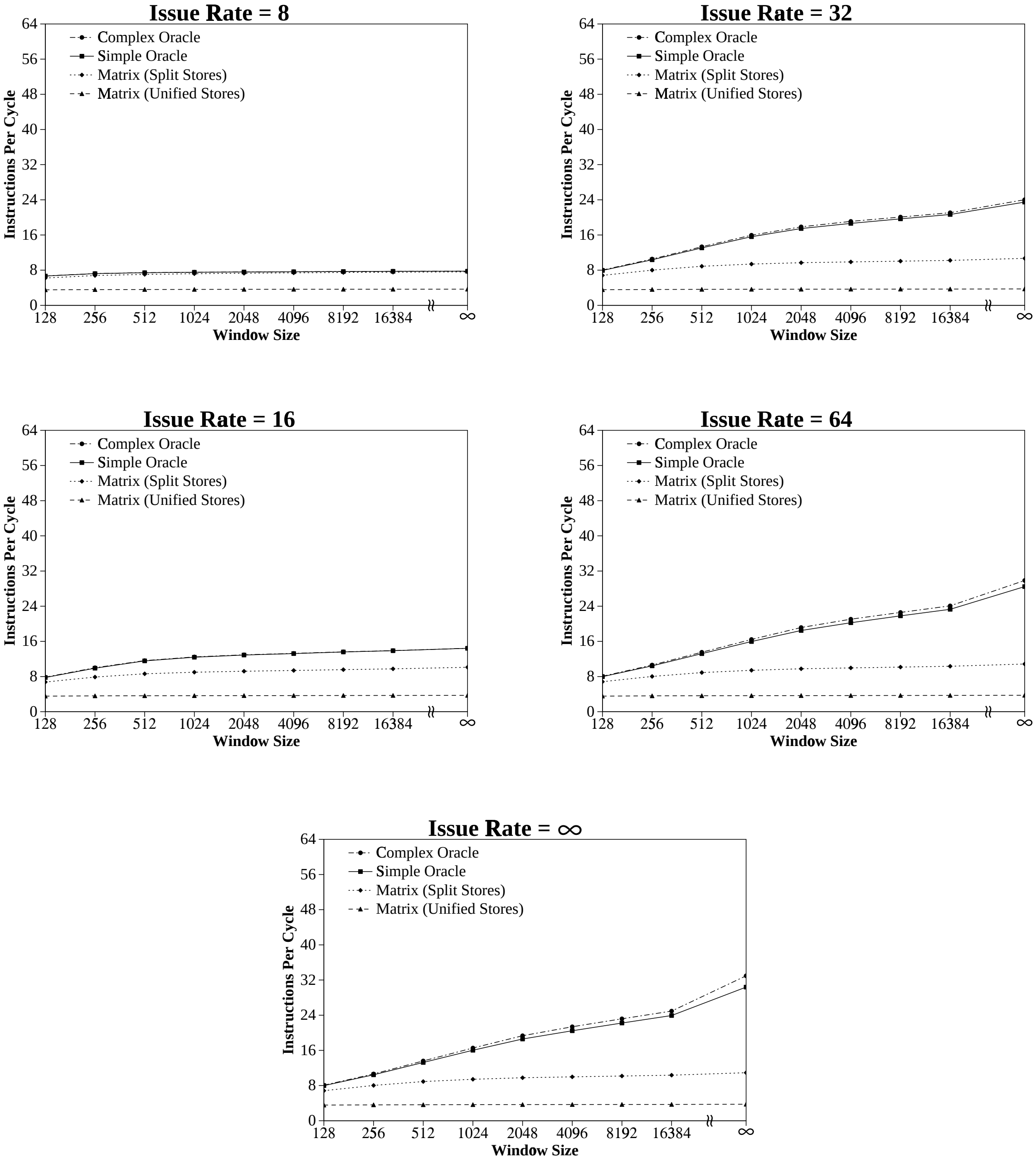


Figure 4.7: Memory Disambiguation Techniques—Harmonic Average

uses dependency matrix and that implements split stores. For the UDF machine that uses dependency matrix and implements unified stores, 62% of the loads delay their result distributions. The performance of this machine is only 3.7 IPC. This is the upper bound on the performance of any machine that uses dependency matrix and that implements unified stores.

Machines that used dependency matrix and that implemented split stores performed significantly better than the machines that used dependency matrix and that implemented unified stores. For an issue rate of 8 at any given window size, the performances of all machines—except for the machine that used dependency matrix and that implemented unified stores—were almost identical. Machines that implement unified stores perform poorly because the address of a store's memory access cannot be computed until the operand(s) that contain the store data are ready. As a result, store addresses remain unknown for longer period of times, and, for memory disambiguation techniques such as dependency matrix that follow the non-speculative paradigm, load result distributions are delayed more frequently. For example, for the UDF machines that use dependency matrix, 62% of loads delay their result distributions if unified stores are implemented, whereas only 37% delay their distributions if split stores are implemented.

For both complex oracle and simple oracle, a load that is independent of any stores in the active window obtains its result from the data cache and then immediately distributes this result to its dependent instructions. A load that is dependent on a store will have its data forwarded from the store. After the data has been bypassed to the load, the load immediately distributes its result. What distinguishes these two techniques from each other is when the store forwards its data to the dependent load. For complex oracle, the store forwards its data to the load as soon as the data is known. For simple oracle, the store forwards its data to the load only after the load's address, the store's address, and the store's data are known. If the load's address and the store's address are known before (or at the same time as) the store's data, both techniques forward the store data to the load at the same time, so there is no advantage to using complex oracle over simple oracle.

In Figure 4.7, machines that used simple oracle performed almost as well as the machines that used complex oracle. For the machine that used simple oracle and had an issue rate of 64 and a window size of 16384, only 26% of the (dynamic) load instructions would have distributed their results earlier if the machine had used complex oracle instead of simple oracle. That is, only 26% of loads were dependent on stores in the active window *and* had their bypasses delayed because the addresses of the load and/or store involved in the bypass were unknown. For the perl (Figure A.7) and groff (Figure A.10) benchmarks, machines that used simple oracle performed worse than the machines that used complex oracle. For the perl benchmark, for the machine that used simple oracle and had an issue rate of 64 and a window size of 16384, 34% of loads would have distributed their results earlier if the machine had used complex oracle instead of simple oracle. For the groff benchmark, for the machine with the same configuration, this fraction is 33%.

This fraction does not necessarily correlate to the speedup that complex oracle provides over simple oracle. Even if the fraction is small, the few loads that are delayed by a machine using simple oracle may be on the critical paths of dependency chains. Eliminating these stalls can increase the total amount of ILP (see the perl benchmark in Figure A.7 and the chess benchmark in Figure A.9), and the amount of ILP that can be exploited by a machine with a fixed sized window. For the go (Figure A.3) and python (Figure A.14) benchmarks, machines that used simple oracle performed worse than the machines that used complex oracle. For the go benchmark, for the machine that used simple oracle and had an issue rate of 64 and a window size of 16384, this fraction was only 21%, yet the speedup of the machine that used complex oracle over the machine that used simple oracle was 17%. For the python benchmark, for the machine that used simple oracle and had an issue rate of 64 and a window size of 4096, this fraction was 24% (a little below average), yet the speedup was 18% (well above average).

4.5 Summary

This chapter has shown that, if memory dependencies are handled aggressively, there is a significant amount of parallelism available in a single instruction stream. If memory dependencies are handled aggressively, via memory disambiguation techniques in the speculative paradigm, there is enough parallelism to sustain an execution rate of 33 IPC on a UDF machine. If memory dependencies are handled less aggressively, via memory disambiguation techniques in the non-speculative paradigm, there is only enough parallelism to sustain an execution rate of 11 IPC on a UDF machine. This chapter also measured the amount of parallelism that can be exploited by RDF machines with various configurations. For a machine with an issue rate of 16 and the simple oracle memory disambiguation technique, the average performance approaches 14.4 IPC as the window size approaches infinity. At a window size of 1024, the average performance is 12.4 IPC.

It may be possible to improve the performance of both the UDF and the RDF machines. Serializing instructions can significantly limit the amount of ILP. It might be possible to execute serializing instructions speculatively, but to make it "appear" as though they are executed non-speculatively. This might reduce the impact of serializing instructions. Future compilers may also transform programs to increase ILP. In addition, techniques such as memoization [10, 106] and value prediction [71] may be used to eliminate some of the dependencies in a program that limit the amount of ILP.

CHAPTER 5

Performance Bottlenecks

The previous chapter showed that the potential performance of an RDF machine with an issue rate of 16, a window size of 1024, and the simple oracle memory disambiguation technique is 12.4 IPC. This performance is far greater than that of any existing processors. Unfortunately, this *ideal* machine can't be built, because it has a perfect (100 percent hit rate) instruction cache, a perfect (omniscient) branch predictor, a perfect execution core (i. e., an execution core with an unbounded numbers of functional units, each of which can perform every desired operation), and a perfect single cycle data cache.

Chapter 6 will present my preliminary investigation of out-of-order fetch/decode/issue. My preliminary investigation uses the RDF machine described in the preceding paragraph. However, to make this machine more realistic, I gave it a real instruction cache, a real branch predictor, a real execution core, and a real data cache. The non-ideality of each of these four components bottlenecks the performance of this *real* machine, reducing its performance from 12.4 IPC to 3.0 IPC.

This chapter examines the performance bottlenecks that are created by having these non-ideal components. It looks at four bottlenecks: one bottleneck for each of the four non-ideal components in the RDF machine used for our preliminary investigation of out-of-order fetch/decode/issue. It shows how severe each of the four bottlenecks is, and how these bottlenecks prevent the processor from achieving its potential performance.

There are two reasons why it is important to look at performance bottlenecks. First, by examining bottlenecks, researchers can determine which bottlenecks are severe enough to warrant doing something about them. For example, in this chapter, I will show that instruction cache misses create one of the severe bottlenecks for a sixteen wide issue processor, and therefore this bottleneck must be dealt with. Second, given that all processor design teams have limited manpower, prudent investment of that manpower is essential for the processor to meet its performance goals and to ship on time. Processor architects and implementors can examine the processor's bottlenecks to pinpoint the low-hanging fruit; that is, areas of investment where large performance gains are likely with only a minimum expenditure of manpower.

This chapter is organized into four sections. Section 5.1 describes my methodology for studying performance bottlenecks. Section 5.2 describes my assumptions about data cache and instruction cache access times. Section 5.3 looks at each of the four bottlenecks. Section 5.4 provides a summary of the chapter.

5.1 Methodology

To study performance bottlenecks, I will use two machines: an *ideal* machine and a *real* machine. Both machines are RDF machines that have an issue rate of 16, a window size of 1024, the instruction class latencies specified in Table 4.1, and the simple oracle memory disambiguation technique. The ideal machine has a perfect instruction cache, a perfect branch predictor, a perfect execution core, and a perfect data cache. For the real machine, all four components (i. e., instruction cache, branch predictor, execution core, and data cache) are real.

Whenever a real component is used instead of a perfect component, a performance bottleneck is created. I will look at four bottlenecks—one bottleneck for each of the four components. The first bottleneck, created when a real instruction cache is used, is due to instruction cache misses. The second bottleneck, created when a real branch predictor is used, is due to branch mispredicts. The third bottleneck, created when a real execution core is used, is due to a lack of execution bandwidth. The fourth bottleneck, created when a real data cache is used, is due to data cache misses. For each of these four bottlenecks, I will assess its severity using two different approaches.

In the first approach, the severity of a bottleneck is assessed by adding that bottleneck to the ideal machine. When the bottleneck is added, the performance of the ideal machine drops according to the bottleneck's severity. If the bottleneck is the bottleneck that results from having a real 'X'; where 'X' is either an instruction cache, branch predictor, execution core, or data cache; the bottleneck is added by trading the perfect ‘X’ used by the ideal machine for a real ‘X’. By comparing the performance of the resulting machine, which has a real ‘X’, to the performance of the ideal machine, which has a perfect ‘X’, I can assess the severity of the bottleneck that results from having a real ‘X’.

This first approach is optimistic: it assumes that when the machine is built, there will be viable solutions for all bottlenecks except for the bottleneck in question. When determining a bottleneck’s severity, this approach does not account for the interactions between that bottleneck and other bottlenecks. For example, the bottleneck due to a lack of execution bandwidth interacts with the bottleneck due to branch mispredicts. In machines with real execution cores, the execution of some instructions is delayed due to contention among instructions for a limited number of functional units. If a delayed instruction feeds a dependency chain that resolves a branch, the resolution of a mispredicted branch may be delayed, worsening the bottleneck due to branch mispredicts. This first approach fails to account for the interaction between the bottleneck due to a lack of execution bandwidth and the bottleneck due to branch mispredicts.

In the second approach, the severity of a bottleneck is assessed by removing that bottleneck from the real machine. When the bottleneck is removed, the performance of the real machine increases according to the bottleneck’s severity. If the bottleneck is the bottleneck that results from having a real ‘X’; where ‘X’ is either an instruction cache, branch predictor, execution core, or data cache; the bottleneck is removed by trading the real ‘X’ used by the real machine for a perfect ‘X’. By comparing the performance of the resulting machine, which has a perfect ‘X’, to the performance of the real machine, which has a real ‘X’, I can assess the severity of the bottleneck that results from having a real ‘X’.

This second approach is pessimistic: it assumes that when the machine is built, there will not be viable solutions for bottlenecks other than the bottleneck in question. In this approach, the severity of a bottleneck is determined by assuming that the machine will have all of the other bottlenecks. Thus, unlike the first approach, this approach does account for the interactions between the bottleneck in question and other bottlenecks. Unfortunately, the performance advantage that results from removing a bottleneck is sometimes masked by interactions between that bottleneck and the other bottlenecks. For example, the bottleneck due to instruction cache misses interacts with the bottleneck due to a lack of execution bandwidth. In machines with real instruction caches, instruction cache misses reduce the rate at which instructions are supplied to the execution core; that is, they reduce the issue bandwidth. Execution bandwidth only becomes a bottleneck if it is less than the issue bandwidth. A machine that suffers from a large number of instruction cache misses has low issue bandwidth, and thus requires little execution bandwidth. Put another way, if the bottleneck due to instruction cache misses is the dominant bottleneck, removing the bottleneck due to a lack of execution bandwidth will provide little (or no) performance advantage.

The first approach assumes that constant progress is being made on reducing all bottlenecks. It identifies the bottleneck that will be the most severe in future CPU designs if significant progress is not made on reducing that bottleneck. The second approach is useful for pinpointing the low-hanging fruit. It can be used to assess the severity of the bottlenecks of an existing CPU design. [1] The component associated with the most severe bottleneck is the single component, that, if improved, will yield the largest increase in performance. The first approach is not as useful for pinpointing the low-hanging fruit, because an existing CPU design will have many bottlenecks that interact with each other, yet the approach does not account for these interactions. Hence, eliminating the bottleneck that the first approach identifies as the most severe may provide little (or no) performance advantage for an existing CPU design.

The four components used by the real machine are identical to those of the RDF machine that will be used in the next chapter for my preliminary investigation of out-of-order fetch/decode/issue. For the second approach, where the severity of a bottleneck is assessed by removing that bottleneck from the real machine, the bottleneck is removed by trading the real component for a perfect component. In the experiments in this chapter, I will also show what happens if the bottleneck is reduced rather than removed. Additionally, I show what happens if the bottleneck is made more severe. The bottleneck will be reduced (or made more severe) by trading the real component for another real component that is better (or worse). For example, to reduce the bottleneck due to branch mispredicts, I will trade the real branch predictor for another real branch predictor that has a lower mispredict rate. Each of the four default components (i. e., the components that are identical to those of the RDF machine used in the next chapter) is described in the paragraphs below.

[1] Either of the two approaches can be used to assess the severity of bottlenecks other than the four presented in this chapter. For example, they both can be used to assess the severity of the bottleneck due to instruction cache translation lookaside buffer (TLB) misses.

The default instruction cache is non-blocking, direct mapped, 16k bytes, with a 64 byte line size. The cache access requires one cycle. In the event of a cache miss, an additional 10 cycles are required to access the next level of cache and/or memory. In the experiments, I will vary the size, access time, and miss penalty of this default instruction cache.

The default branch predictor actually contains three separate predictors: one predictor for conditional branches, one predictor for indirect (or computed) branches, and one predictor for subroutine returns. The conditional branch predictor is a gshare [81] scheme which exclusive-ORs a 16-bit global history with the fetch address to select the appropriate pattern history table entry. Indirect (or computed) branch targets are predicted using the "tagless" variety of the pattern based predictor proposed by Chang, Hao, and Patt [19]. A 9-bit global history is used to select an entry in a table of indirect branch target addresses. To improve prediction accuracy, I added a single "hysteresis" bit to each entry in the table. The bit controls the replacement of the branch target address stored in that entry. [2] Subroutine returns are predicted using a 64 entry Return Address Stack (RAS). To model the real branch predictor, as the branches are encountered in the dynamic instruction stream, a prediction is made. This prediction is compared to the real outcome of the branch. If a prediction is incorrect, issue is stalled until the branch is resolved and instructions from the correct path are available for issuing. I assume six cycles between when a branch is resolved and when instructions from the correct path are available. Since one cycle is required to execute the branch, the minimum branch mispredict penalty is seven cycles. This mispredict penalty is identical to that of the Compaq Alpha 21264 [43]. In the experiments, I will vary the type of conditional branch predictor and the type of indirect branch predictor. I will also vary the sizes of the conditional branch predictor, indirect branch predictor, and RAS (subroutine return predictor); and the minimum branch mispredict penalty. I did not investigate the problem of Branch Target Buffer (BTB) misses or any possible solutions. I modeled a perfect (100 percent hit rate) BTB under the assumption that there will be a suitable solution.

The default execution core consists of sixteen fully pipelined functional units, where

[2] Whenever an entry provides the correct prediction for a branch, its hysteresis bit is set to 1. Whenever it provides an incorrect prediction, its hysteresis bit is set to 0. The branch target address stored in an entry can only be replaced if the entry provides an incorrect prediction and the hysteresis bit was 0 at the time the prediction was made.

each functional unit is capable of performing every desired operation. Each cycle, the oldest sixteen ready instructions are scheduled for execution. (This execution core is modeled by setting the dispatch rate to sixteen. The dispatch rate is the maximum number of instructions that can be scheduled for execution on functional units in a single cycle.) In the experiments, I will vary the number of functional units in the core. The number of instructions that can be scheduled for execution each cycle was always set equal to the number of functional units in the core.

The default data cache is non-blocking, direct mapped, 16k bytes, with a 64 byte line size. Loads require one cycle for address calculation and one cycle for cache access. In the event of a cache miss, loads require an additional 10 cycles for accessing the next level of cache and/or memory. (The load latency on a cache hit is 2 cycles. The load latency on a cache miss is 12 cycles.) In the experiments, I will vary the size, access time, and miss penalty of this default data cache. The load latency will be scaled according to the access time and miss penalty of the data cache.

5.2 Cache Access Time

One obvious way to reduce the severity of bottlenecks that result from cache misses (either instruction or data) is to build bigger caches. Bigger caches typically experience fewer misses. Unfortunately, bigger caches also take up more chip area, use more power, and have longer access times. To meet a CPU's power and cycle time requirements, it may be necessary to pipeline the access of a large cache over several cycles. There are some negative consequences to pipelining the instruction and data caches—consequences which may eliminate the benefit of a higher cache hit rate. Pipelining the instruction cache increases the depth of the front-end of the processor pipeline (i. e., the number of pipeline stages required for instruction fetch, decode, and issue), which increases the number of cycles that are required to recover from a mispredicted branch, and hence worsens the bottleneck due to branch mispredicts. Pipelining the data cache increases the latency of loads, which delays the execution of instructions that are dependent on loads. Loads feed dependency chains that resolve branches. Pipelining the data cache increases the time required to resolve a mispredicted branch, which worsens the bottleneck due to branch mispredicts.

The access time of a cache cannot be determined without knowledge of the process technology that it will be implemented in, and the circuit techniques that it will be implemented with. Both the process technology and the circuit techniques are continually evolving. The number of pipeline stages required to access the cache cannot be determined unless the cycle time of the machine is known. The choice of cycle time is based, in part, on the access time of the cache. (Cache size and cycle time should be selected together in order to maximize processor performance.) I expect that a machine with an issue rate of 16 and a window size of 1024 can be built within the next 5 to 10 years [99]. Note that this machine has the same issue rate and window size as my ideal machine and my real machine. Over the past 10 years, the maximum size of a cache that can be accessed in a single cycle has not significantly changed. Rather than trying to forecast what will happen to process technology, circuit techniques, and cycle time, I will assume that the maximum size of a cache that can be accessed in a single cycle will not change between now and when this machine is built. I will assume that a single cycle direct-mapped cache is limited to 16k bytes. (My default instruction and data caches are single cycle direct-mapped 16k byte caches.)

McFarland [77] and others [11, 76, 105] have pointed out an ominous trend: with each successive generation of process technology, gate delay shrinks a lot, but wire delay shrinks only a little. That is, gate delay is scaling quicker than wire delay. This trend has an important implication: the portion of a cache's access time that is due to wire delay increases with each new generation of process technology, because a significant portion of a cache's access time is due to wire delay, which does not scale as quickly as gate delay. Note that a cache's memory cells are arranged in a two-dimensional array. One wire of length $\sqrt{k}$ runs through each row of the array, and one wire of length $\sqrt{k}$ runs through each column. The number of cells in the array is proportional to k (i. e., $\sqrt{k} \times \sqrt{k}$). A cell is accessed via the wire that runs through its row and the wire that runs through its column. In the limit, where all of a cache's access time is due to wire delay, the access time of this cache with k memory cells is proportional to $\sqrt{k}$.

Throughout this dissertation, I will make two different assumptions about cache access times: an optimistic assumption and a pessimistic assumption. When appropriate, I will present experimental results using both assumptions. The optimistic assumption is that the access time of a cache does not increase as its size increases. The access time of all caches will be assumed to be 1 cycle. The pessimistic assumption is that all of the cache access time is due to wire delay, and, as a result, the access time is proportional to the square root of the size of the cache. I will assume that the access time of a 16k byte cache is 1 cycle, a 64k byte cache is 2 cycles, a 256k byte cache is 4 cycles, and a 1M byte cache is 8 cycles.

5.3 The Bottlenecks

This section looks at each of the four bottlenecks. Subsection 5.3.1 provides the measurement and analysis of the bottleneck due to instruction cache misses. Subsection 5.3.2 provides the measurement and analysis of the bottleneck due to branch mispredicts. Subsection 5.3.3 provides the measurement and analysis of the bottleneck due to a lack of execution bandwidth. Subsection 5.3.4 provides the measurement and analysis of the bottleneck due to data cache misses.

5.3.1 The Instruction Cache

In the RDF model of execution, instructions from the program's dynamic instruction stream are issued into the active window. The dynamic instruction stream originates from a perfect (100 percent hit rate) instruction cache. This perfect instruction cache can't be built, so real CPU designs must settle for instruction caches that are less than perfect. That is, they must settle for (real) instruction caches that occasionally experience misses.

Instruction cache misses reduce the supply of issue bandwidth. During an instruction cache miss, the front-end of the processor pipeline stalls while it waits for the instructions to be supplied by the next level of the cache/memory hierarchy. This results in cycles in which no instructions are issued into the active window.

To determine the severity of this bottleneck, I simulated several different machines. Figure 5.1 shows their performances averaged over all the benchmarks. There are five lines. One line shows the performance of the ideal machine with a perfect (100 percent hit rate) instruction cache. This machine provides an upper bound on the performances of machines with real instruction caches. The remaining lines plot the performances of ideal machines that have been augmented with real instruction caches. Each line represents a set of machines that all have the same penalty for an instruction cache miss. The miss penalty is a function of the cycle time and cache/memory hierarchy. It varies from implementation to implementation, so I have varied it from 6 to 32 cycles. Each point in a line plots the performance of an ideal machine with an instruction cache of the size listed on the horizontal axis. The instruction cache size was varied from 16k bytes to 1M byte. The instruction cache access required one cycle regardless of its size. That is, its access time was not scaled with its size.

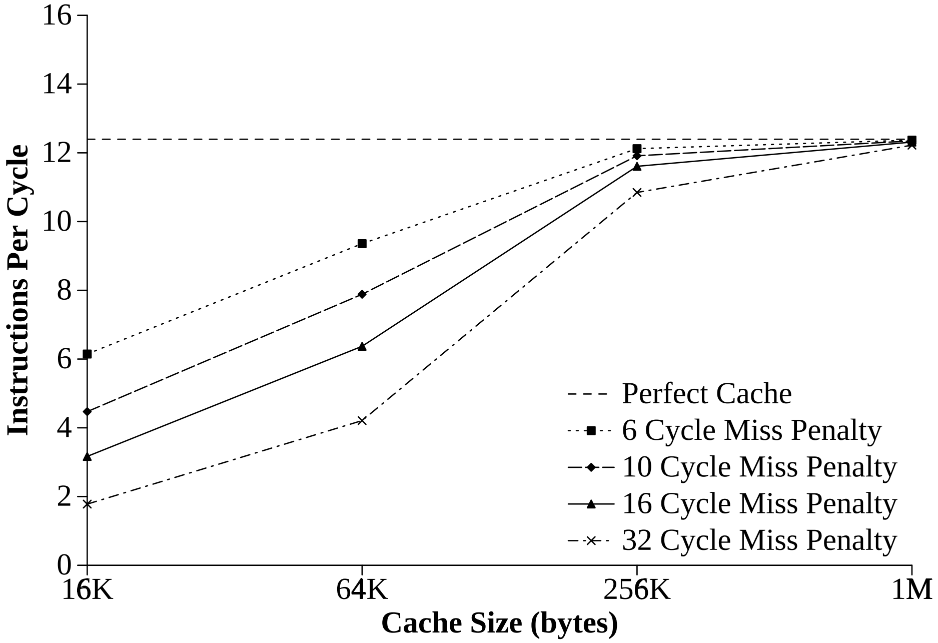


Figure 5.1: Ideal Machine with Varied Instruction Cache Size—Harmonic Average

The machines with larger instruction caches experience fewer instruction cache misses, and therefore have better performance. For the machines with 6 cycle miss penalties, performance increased from 6.15 IPC to 12.37 IPC as the instruction cache size was increased from 16k bytes to 1M byte. For the machines with 32 cycle miss penalties, performance increased from 1.78 IPC to 12.22 IPC. Also, the machines with low miss penalties have better performance than the machines with high miss penalties. For the machines with 16k byte instruction caches, performance increased from 1.78 IPC to 6.15 IPC as the miss penalty was decreased from 32 cycles to 6 cycles. Note that if the miss penalty is reduced all the way to 0 cycles, the latency on a cache miss is identical to the latency on a cache hit. As a consequence, the performance of a machine with a 0 cycle miss penalty, and an instruction cache of any size, would be identical to that of the machine with a perfect instruction cache.

The results indicate that for machines with either 16k byte or 64k byte instruction caches, instruction cache misses significantly bottleneck performance. The performance of the ideal machine with the perfect instruction cache is 12.39 IPC. The performances of the machines with 16k byte instruction caches are between 6.15 IPC (6 cycle miss penalty) and 1.78 IPC (32 cycle miss penalty). These machines lose between 50% and 86% of their potential performance due to instruction cache misses. The performances of the machines with 64k byte instruction caches are between 9.36 IPC and 4.21 IPC. They lose between 25% and 66% of their potential performance due to instruction cache misses.

For machines with either 256k byte or 1M byte instruction caches, instruction cache misses don't significantly bottleneck performance. These caches are probably large enough to hold the (average) program's working set. The worst case was for the machine with a 256k byte instruction cache and a 32 cycle miss penalty. The performance of this machine was only 12% worse than the performance of the machine with a perfect instruction cache. Results for the individual benchmarks are provided in Figure A.17 (SPEC) and Figure A.18 (Non-SPEC) of Appendix A.

The machines used to generate Figure 5.1 and Figures A.17–A.18 all had single cycle (i. e., non-pipelined) instruction caches and perfect branch predictors. Pipelining the instruction cache increases the depth of the front-end of the processor pipeline, which increases the number of cycles required to recover from a mispredicted branch. The only impact that pipelining the instruction cache has on performance is that it worsens the bottleneck due to branch mispredicts. [3] Since the machines used to generate Figure 5.1 and Figures A.17–A.18 had perfect branch predictors, pipelining their instruction caches has no impact on their performances. For this reason, I do not present results for machines with pipelined instruction caches (i. e., caches whose access times have been scaled with their sizes) and perfect branch predictors.

Figure 5.2 shows the performance averaged over all the benchmarks for several machines. One line shows the performance of the real machine that was given a perfect instruction cache. This machine provides an upper bound on the performances of machines with real instruction caches. The remaining lines plot the performances of real machines with real instruction caches. For these experiments, the instruction cache access required one cycle regardless of its size. That is, its access time was not scaled with its size. Because of this, the depth of the front-end of the processor pipeline did not depend on the instruction cache size, and, as a result, the minimum branch mispredict penalty was a constant 7 cycles for all machines.

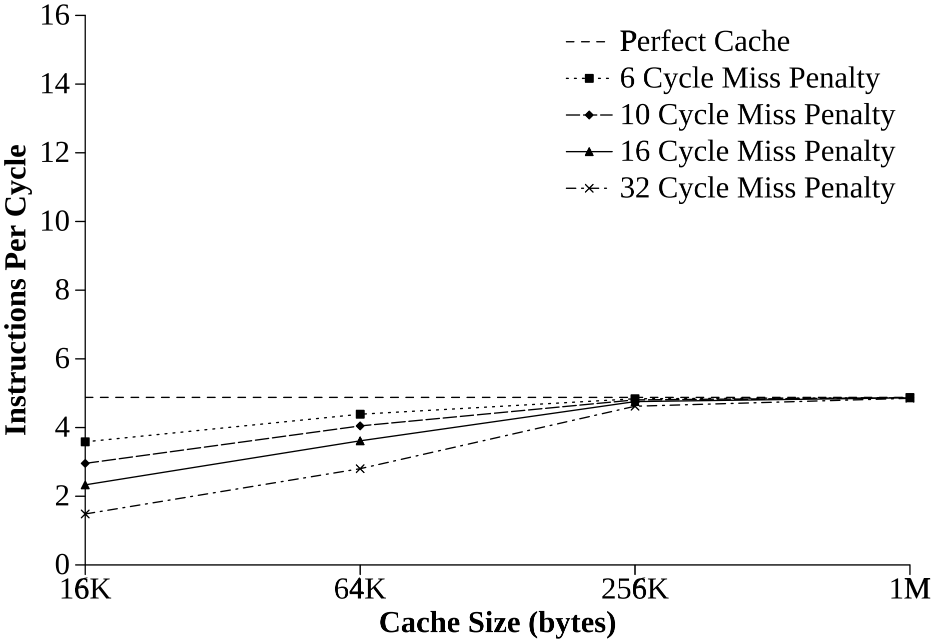


Figure 5.2: Real Machine with Varied Instruction Cache Size (Constant Mispredict Penalty)—Harmonic Average

[3] The number of issue cycles lost due to an instruction cache miss may increase if the cache is pipelined, worsening the bottleneck due to instruction cache misses. However, for all the machines used in this chapter and the next, the number of issue cycles lost is equal to the instruction cache miss penalty; i. e., the number of cycles lost does not depend on the cache access time.

The instruction cache for a real machine need not be as aggressive as the instruction cache for an ideal machine. The ideal machine with a 16k byte instruction cache loses between 50% (6 cycle miss penalty) and 86% (32 cycle miss penalty) of its potential performance due to instruction cache misses. For the same size instruction cache, the real machine loses between 27% (6 cycle miss penalty) and 70% (32 cycle miss penalty) of its potential performance. Additionally, the real machine is less sensitive to increasing miss penalty. When the miss penalty is increased from 6 cycles to 32 cycles, the performance of the ideal machine with a 16k byte instruction cache falls by 71%. For the same size instruction cache, the performance of the real machine only falls by 59%. The reason for this is that the real machine is hampered by bottlenecks other than the bottleneck due to instruction cache misses. As a result, instruction cache misses have less of an overall impact on the real machine than they do on the ideal machine, which is *only* hampered by the instruction cache bottleneck. Results for the individual benchmarks are provided in Figure A.19 (SPEC benchmarks) and Figure A.20 (Non-SPEC benchmarks) of Appendix A.

Figure 5.3 shows the performance averaged over all the benchmarks for the same machines that were used to generate Figure 5.2, except that for these machines, the access time of the instruction cache was scaled with its size. The access time was 1 cycle at 16k bytes, 2 cycles at 64k bytes, 4 cycles at 256k bytes, and 8 cycles at 1M byte. Because the access time was scaled, the depth of the front-end of the processor pipeline depends on the instruction cache size, and, as a result, the minimum branch mispredict penalty must be scaled according to the instruction cache size. The minimum mispredict penalty was 7 cycles at a cache size of 16k bytes, 8 cycles at 64k bytes, 10 cycles at 256k bytes, and 14 cycles at 1M byte. The machine with the perfect instruction cache is supposed to represent the ideal solution to the instruction cache bottleneck. The ideal solution is an instruction cache that has a 100 percent hit rate and a single cycle access time. (A single cycle access time is ideal because it results in the smallest possible minimum mispredict penalty.) For this reason, the machine with the perfect instruction cache could always access its cache in a single cycle, and its minimum mispredict penalty was always 7 cycles.

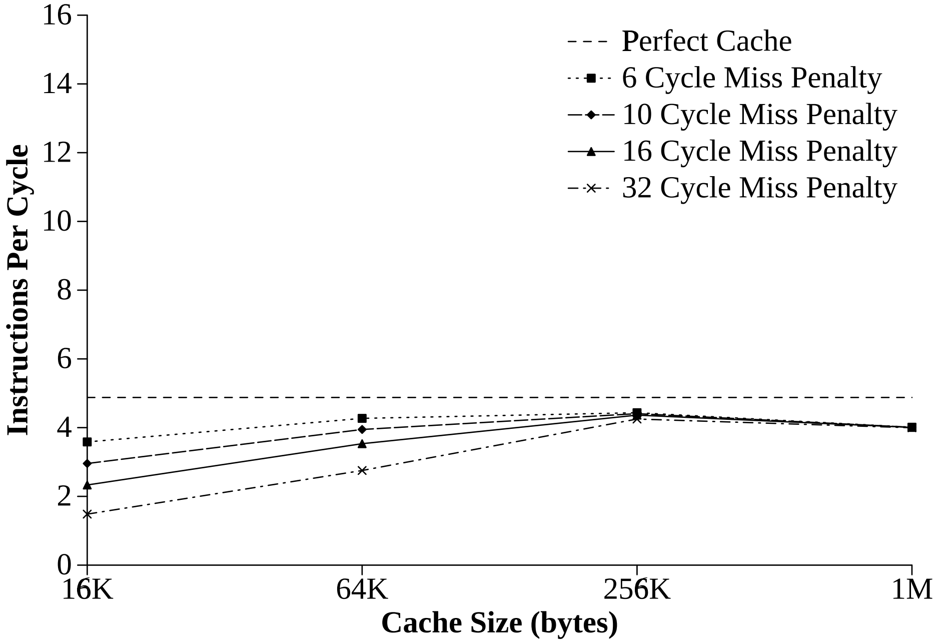


Figure 5.3: Real Machine with Varied Instruction Cache Size (Scaled Mispredict Penalty)—Harmonic Average

As the instruction cache size increases, fewer instruction cache misses occur, which reduces the severity of the instruction cache bottleneck. The minimum branch mispredict penalty also increases as the instruction cache size increases, which increases the severity of the bottleneck due to branch mispredicts. Beyond a certain point, increasing the size of the instruction cache will result in lower performance, because it will increase the severity of the bottleneck due to branch mispredicts more than it reduces the severity of the instruction cache bottleneck. For all machines, increasing the size of the instruction cache beyond 256k bytes results in lower performance. Results for the individual benchmarks are provided in Figure A.21 (SPEC) and Figure A.22 (Non-SPEC) of Appendix A.

5.3.2 The Branch Predictor

In the RDF model of execution, instructions from the program's dynamic instruction stream are issued into the active window. The dynamic instruction stream is created by an omniscient branch predictor that always knows the direction a branch will take. This omniscient branch predictor can't be built, so real CPU designs must settle for a branch predictor that is less than perfect. Fortunately, branch predictors are getting better and better as each day passes. Continuous research on branch prediction has led to the discovery of prediction algorithms with successively higher prediction accuracies. Additionally, this research has found ways of achieving a given prediction accuracy at successively lower implementation costs.

Branch mispredictions throttle the supply of useful issue bandwidth. To determine the severity of this bottleneck, I simulated several different branch predictors. All of the simulated predictors used real (i. e., not synthetic or artificial) prediction algorithms. Some of these predictors have implementation costs that are low enough that they could be implemented in today's processors. Some do not. However, as time marches on, transistor budgets increase. Predictors that are too expensive to implement for today's processors will not be expensive for tomorrow's processors. In addition, branch prediction research may discover a prediction algorithm that achieves the same prediction accuracy as one of these expensive predictors only at a much lower implementation cost.

Figure 5.4 shows the performance averaged over all the benchmarks for several different machines. There are five lines. Each line corresponds to a set of machines that all have the same minimum branch mispredict penalty. The minimum mispredict penalty is the minimum number of cycles between when a mispredicted branch is fetched and when that branch is executed (or resolved). It is equal to the minimum number of pipe stages required to fetch, decode, issue, and execute a branch. The number of pipe stages required to do this varies from implementation to implementation, so I have varied the minimum mispredict penalty from 4 to 14 cycles. The *actual* mispredict penalty for a particular branch is equal to the minimum mispredict penalty, plus the number of cycles the branch waits in the active window for its flow dependencies to be resolved. Each point in a line plots the performance of an ideal machine with the mispredict rate, in mispredicts per 1000 instructions, listed on the horizontal axis. Note that the reciprocal of this rate is the average number of instructions that are executed between mispredicted branches. The machines with non-zero mispredict rates are ideal machines that have been augmented with real branch predictors. The mispredict rates correspond to the mispredict rates of the real predictors. I will describe these real predictors later. The machine with a zero mispredict rate is the ideal machine with the omniscient branch predictor. It provides an upper bound on the performances of the machines with real branch predictors.

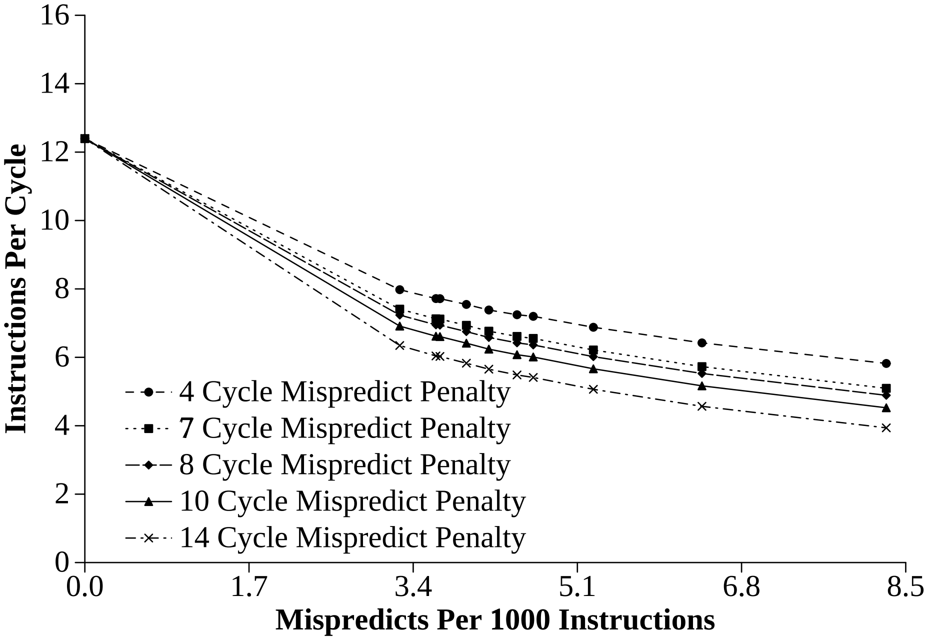


Figure 5.4: Ideal Machine withVaried Mispredict Rate—Harmonic Average

Branch mispredictions significantly bottleneck performance. The performance of the ideal machine with the omniscient branch predictor is 12.39 IPC. The performances of the machines with the best predictor, which has a miss rate of 3.3 mispredicts per 1000 instructions (or, alternatively, one mispredict for every 307 instructions), are between 7.98 IPC (4 cycle minimum mispredict penalty) and 6.35 IPC (14 cycle minimum mispredict penalty). These machines lose between 36% and 49% of their potential performance due to branch mispredictions. The machines with the worst predictor, which has a miss rate of 8.3 mispredicts per 1000 instructions (one mispredict for every 121 instructions), fared much worse. The performances of these machines are between 5.82 IPC and 3.94 IPC. They lose between 53% and 68% of their potential performance due to branch mispredictions.

As pipeline depths increase, the minimum mispredict penalty increases, and the amount of speculative work that must be thrown away in the event of a branch mispredict increases. Not surprisingly, as the minimum mispredict penalty increases, the performance decreases. When the minimum mispredict penalty is increased from 4 cycles to 14 cycles, the performance of the machine with a miss rate of 8.3 mispredicts per 1000 instructions falls from 5.82 IPC to 3.94 IPC, which corresponds to a drop in performance of 32%. The performance of the machine with a miss rate of 3.3 mispredicts per 1000 instructions falls from 7.98 IPC to 6.35 IPC, which corresponds to a drop in performance of only 20%.

Better branch predictors, which have lower mispredict rates, can be used to counter the performance degradation that results from the growing minimum mispredict penalty that is caused by increasing pipeline depths. For example, the performance drop of the machine with a miss rate of 8.3 mispredicts per 1000 instructions that results from increasing the minimum mispredict penalty from 4 cycles to 14 cycles can be completely erased by equipping the machine with any of the branch predictors that have mispredict rates lower than 4.0 mispredicts per 1000 instructions. In the limit, when the mispredict rate of the branch predictor is zero, the minimum mispredict penalty becomes irrelevant, because the predictor never mispredicts. Unfortunately, a branch predictor with a mispredict rate of zero will never be built. Also, as branch predictors get better, the useful issue bandwidth increases. As the issue bandwidth increases, the *actual* mispredict penalty increases, because branches wait longer in the active window for their dependencies to be resolved [15]. Hence, better predictors reduce the total number of branch mispredicts, but they also increase the penalty, or cost, associated with each mispredict. The best solution, of course, is to build a predictor with the lowest mispredict rate, and to minimize the mispredict penalty by pipelining the machine such that there are a minimum number of pipe stages between when a branch is fetched and when it is executed.

Results for the individual benchmarks are provided in Figure 5.5 (SPEC benchmarks) and Figure 5.6 (Non-SPEC benchmarks). Between benchmarks, there are wide variances in the mispredict rates of the branch predictors. For each benchmark, the scale on the horizontal axis, which lists mispredict rates, has been customized for the mispredict rates of the branch predictors for that benchmark.

Note that performance is not entirely dependent on the mispredict rate. The performance also depends on *which* branches are mispredicted. The cost associated with mispredicting a branch differs depending on the branch. Different branch predictors mispredict different branches. If one branch predictor has a mispredict rate that is greater than or equal to that of a second predictor, the performance of a machine using the first predictor may still exceed the performance of a machine using the second predictor if the first predictor generates fewer costly mispredicts than the second predictor. This effect can be seen in the chess, go, li, perl, python, tex, and vortex benchmarks, where the performance sometimes increases when the mispredict rate increases.

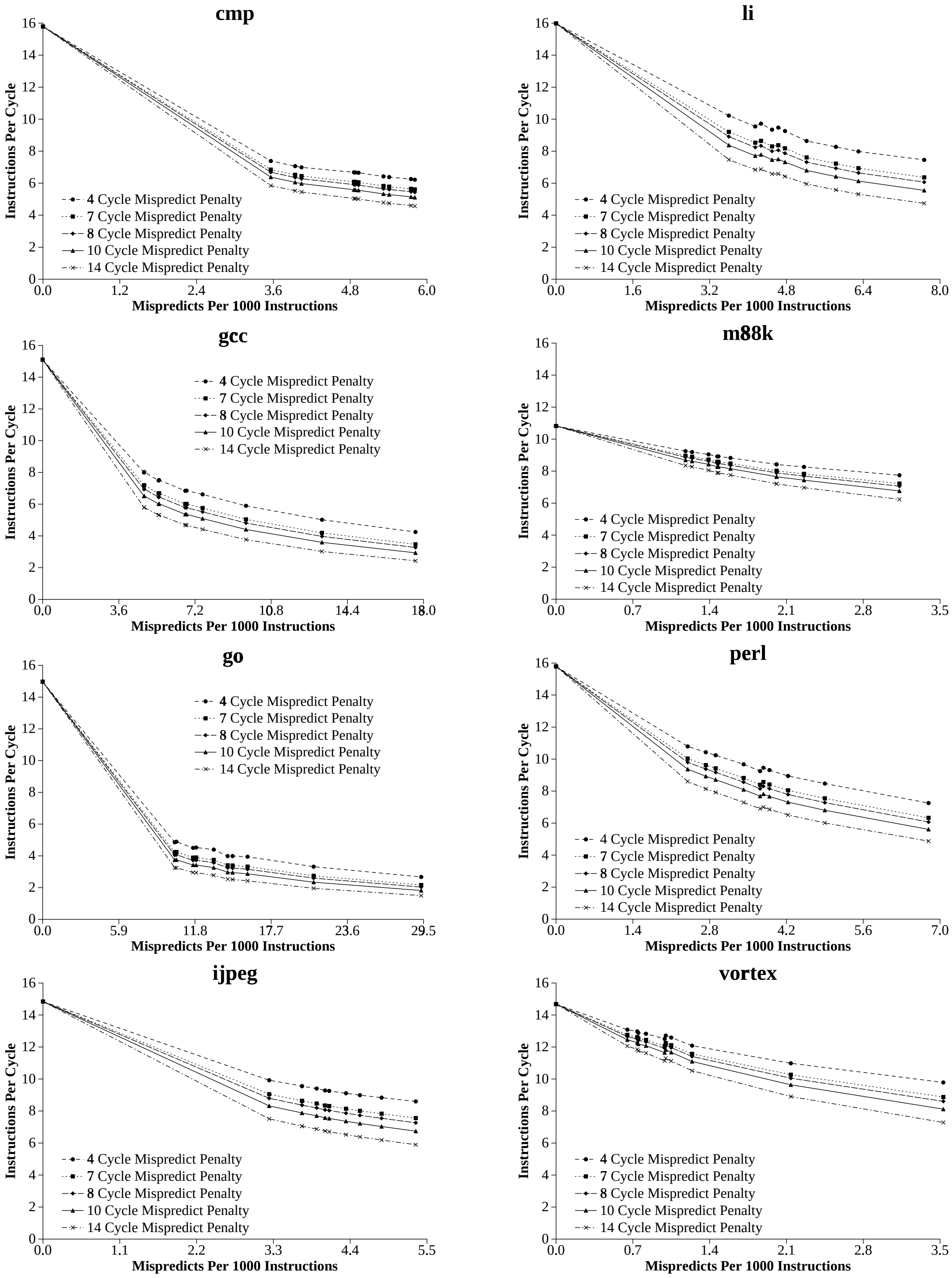


Figure 5.5: Ideal Machine withVaried Mispredict Rate—SPEC Benchmarks

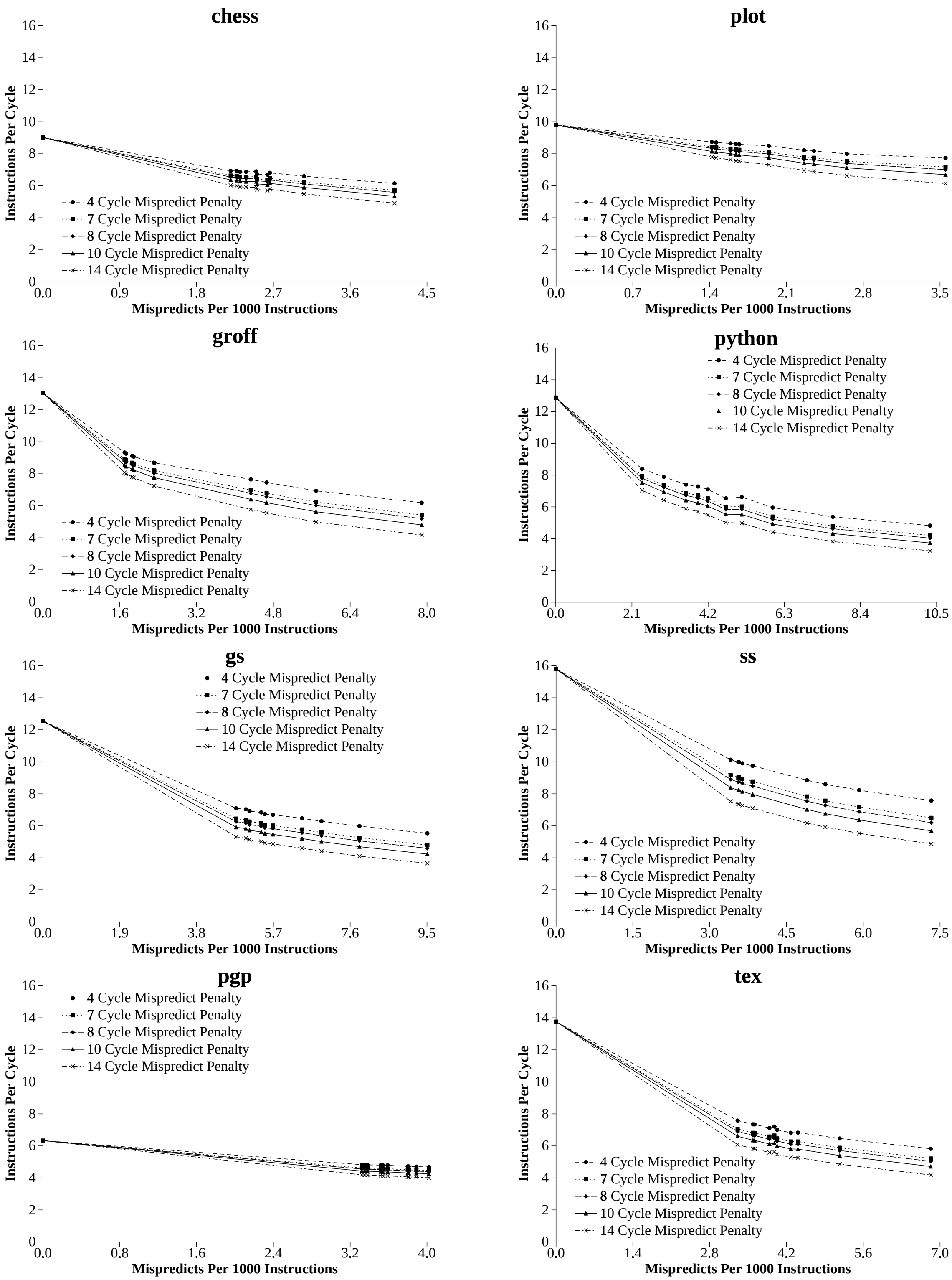


Figure 5.6: Ideal Machine with Varied Mispredict Rate—Non-SPEC Benchmarks

Figure 5.7 shows the performance averaged over all the benchmarks for several different machines. Each point plots the performance of the real machine (i. e., the machine with the real instruction cache, branch predictor, execution core, and data cache) with the mispredict rate listed on the horizontal axis. The machines with non-zero mispredict rates are real machines with real branch predictors. The mispredict rates correspond to the mispredict rates of the real predictors. The machine with a zero mispredict rate is the real machine that has been given the omniscient branch predictor. It provides an upper bound on the performances of the machines with real branch predictors.

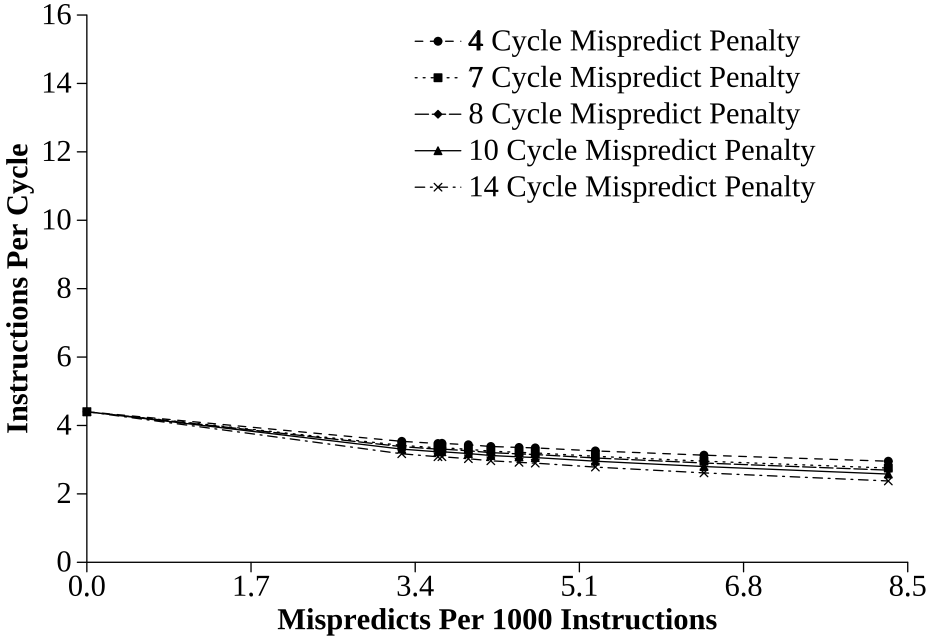


Figure 5.7: Real Machine withVaried Mispredict Rate—Harmonic Average

The branch predictor for the real machine need not be as aggressive as the branch predictor for the ideal machine. The ideal machine with a predictor that has a miss rate of 3.3 mispredicts per 1000 instructions loses between 36% (4 cycle minimum mispredict penalty) and 49% (14 cycle minimum mispredict penalty) of its potential performance due to branch mispredicts (see Figure 5.4). At the same mispredict rate, the real machine only loses between 20% and 28% of its potential performance. Additionally, the real machine is less sensitive to increasing minimum mispredict penalty. When the minimum mispredict penalty is increased from 4 cycles to 14 cycles, the performance of the ideal machine with a miss rate of 3.3 mispredicts per 1000 instructions falls by 20%. At the same miss rate, the performance of the real machine only falls by 10%.

The real machine is hampered by bottlenecks other than branch mispredicts. As a result, branch mispredicts have less of an overall impact on the real machine than they do on the ideal machine, which is only hampered by a single bottleneck: the bottleneck that results from branch mispredicts. The bottleneck that results from branch mispredicts is still severe for the real machine—severe enough to justify further research on branch prediction, aggressive branch predictor implementations, and pipelines that minimize the minimum mispredict penalty. However, this bottleneck is not as severe as it is for the ideal machine. Results for individual benchmarks are provided in Figure A.23 (SPEC benchmarks) and Figure A.24 (Non-SPEC benchmarks) of Appendix A.

Ten different branch predictors were used to generate the results for Figures 5.4–5.7 and Figures A.23–A.24. Each branch predictor actually contains three separate predictors: one predictor for conditional branches, one predictor for indirect (or computed) branches, and one predictor for subroutine returns. All predictors used a perfect (100 percent hit rate) Branch Target Buffer (BTB). The ten predictors were as follows:

1. The conditional branch predictor is a gshare [81] scheme which exclusive-ORs a 14-bit global history with the fetch address to select the appropriate 2-bit pattern history table (PHT) entry. Indirect branch targets are predicted using the "tagless" variety of the pattern based predictor proposed by Chang, Hao, and Patt [19]. A 7-bit global history is used to select an entry in a table of indirect branch target addresses. To improve prediction accuracy, I added a single "hysteresis" bit to each entry in the table. The bit controls the replacement of the branch target address stored in that entry. Subroutine returns are predicted using a 16 entry Return Address Stack (RAS).

2. This predictor is the default branch predictor used throughout this dissertation. It is identical to predictor number 1, except that the conditional branch predictor uses a 16-bit global history, the indirect branch predictor uses a 9-bit global history, and the RAS contains 64 entries.

3. This predictor is identical to predictor number 1, except that the conditional branch predictor uses an 18-bit global history, the indirect branch predictor uses an 11-bit global history, and the RAS contains 256 entries.

4. This predictor is identical to predictor number 1, except that the conditional branch predictor uses a 20-bit global history, the indirect branch predictor uses a 13-bit global history, and the RAS contains 1024 entries.

5. The conditional branch predictor is our Variable Length Path (VLP) predictor for conditional branches [121]. It forms a 16-bit index by hashing together selected target addresses from the 32 most recently encountered branches. The index selects the appropriate 2-bit PHT entry. Indirect branch targets are predicted using our VLP predictor for indirect branches. This predictor forms a 9-bit index by hashing together selected target addresses from the 32 the most recently encountered branches. The index is used to select an entry in a table of indirect branch target addresses. Like the indirect branch predictor used for predictor number 1, a "hysteresis" bit associated with each table entry is used to control the replacement of the branch target address stored in associated entry. Subroutine returns are predicted using a 64 entry RAS.

6. This predictor is identical to predictor number 5, except that the conditional VLP branch predictor generates an 18-bit index, the indirect VLP branch predictor generates an 11-bit index, and the RAS contains 256 entries.

7. This predictor is identical to predictor number 5, except that the conditional VLP branch predictor generates a 20-bit index, the indirect VLP branch predictor generates a 13-bit index, and the RAS contains 1024 entries.

8. The conditional branch predictor is a hybrid [81] consisting of a global predictor and a local (or per branch address) predictor. The global predictor is a VLP predictor. The local predictor is an SAg Two-Level Adaptive Branch Predictor [136]. For an SAg predictor, the first level of history is stored in a table of branch history registers, called the Branch History Table (BHT). To make a prediction, the lower bits of a branch's address are used to select an entry from the BHT. The history register stored in that entry is then used to select the appropriate PHT entry. To predict a branch in a single cycle [4], a single-bit lookahead prediction [134] is stored alongside each history register in the BHT. All the SAg predictors used in my experiments have a large number of branch history registers in their BHTs, so they are essentially "tagless" versions of

[4] The SAg predictor requires two sequential table accesses to make a prediction. Squeezing two accesses into a single cycle is difficult.

PAg predictors. The selection mechanism for the hybrid is an array of counters. Each branch is mapped to a counter via its address. The counter keeps track of which predictor (either the global [VLP] or the local [SAg]) is currently more accurate for the branch. When the branch is fetched, its counter is used to select the predictor that will (hopefully) provide the most accurate prediction.

For this particular conditional branch predictor, the VLP predictor component of the hybrid is the conditional VLP branch predictor used by predictor number 5. This VLP predictor uses a 16-bit index to access the PHT. The SAg predictor component of the hybrid uses a BHT with 4096 (2^{12}) 12-bit branch history registers, and a PHT that contains 3-bit saturating up-down counters. The selection mechanism for the hybrid is an array of 16384 (2^{14}) 2-bit up-down counters. The indirect branch targets for this predictor (i. e., predictor number 8) are predicted using the indirect VLP branch predictor used by predictor number 5. This VLP predictor uses a 9-bit index to access the table of indirect branch target addresses. Subroutine returns are predicted using a 64 entry RAS.

9. This predictor is identical to predictor number 8, except that the sizes of the conditional branch predictor, the indirect branch predictor, and the RAS have been increased. The VLP predictor component of the hybrid conditional branch predictor uses an 18-bit index to access its PHT. The SAg predictor component uses a BHT with 8192 (2^{13}) 13-bit branch history registers. The selection mechanism is an array of 131072 (2^{17}) counters. Indirect branch targets are predicted using an indirect VLP branch predictor that uses an 11-bit index to access the table of indirect branch target addresses. Subroutine returns are predicted using a 256 entry RAS.

10. This predictor is identical to predictor number 8, except that the sizes of the conditional branch predictor, the indirect branch predictor, and the RAS have been increased. The VLP predictor component of the hybrid conditional branch predictor uses a 20-bit index to access its PHT. The SAg predictor component uses a BHT with 32768 (2^{15}) 15-bit branch history registers. The selection mechanism is an array of 524288 (2^{19}) counters. Indirect branch targets are predicted using an indirect VLP branch predictor that uses a 13-bit index to access the table of indirect branch target addresses. Subroutine returns are predicted using a 1024 entry RAS.

Table 5.1 lists the sizes and miss rates for the ten predictors. The predictor number is given in the far left column. The column labeled "*Composite Predictor*" provides the size and miss rate of the (composite) predictor for the associated row. The miss rate is an average taken over all benchmarks. It is given in two different metrics. The first is the percent of *all* branches (including, for example, unconditional PC-relative branches, which don't require a prediction) that are mispredicted. The second is the number branch mispredicts per 1000 instructions, or, mispredicts per kilo instruction (pki).

	Cond. Predictor			*Indirect Predictor*			*Return Predictor*			*Composite Predictor*		
		miss			miss			miss				
	size	rate	miss	size	rate	miss	size	rate	miss	size	miss rate	
#	kbyte	%	ratio	kbyte	%	ratio	kbyte	%	ratio	kbyte	%	pki
1	4	6.24	.835	1/2	30.96	.119	1/16	2.86	.047	5	5.54	8.30
2	16	5.02	.855	2	26.58	.125	1/4	0.92	.020	18	4.34	6.39
3	64	4.27	.866	8	23.14	.122	1	0.42	.012	73	3.64	5.27
4	256	3.89	.879	32	20.84	.116	4	0.17	.005	292	3.26	4.64
5	16	3.76	.888	2	13.56	.084	1/4	0.92	.028	18	3.13	4.48
6	64	3.48	.916	8	11.36	.069	1	0.42	.015	73	2.80	3.95
7	256	3.31	.929	32	10.77	.064	4	0.17	.006	292	2.62	3.68
8	28	3.46	.880	2	13.56	.090	1/4	0.92	.030	30	2.92	4.18
9	113	3.15	.908	8	11.36	.075	1	0.42	.017	122	2.57	3.64
10	460	2.87	.920	32	10.77	.073	4	0.17	.007	496	2.31	3.26

Table 5.1: Branch Predictor Sizes, Miss Rates, and Miss Ratios

Table 5.1 also lists the sizes, miss rates, and miss ratios of the three predictors (conditional, indirect, and subroutine return) that make up each of the ten composite predictors. The miss rate is the percentage of the predictor's predictions that are incorrect. For example, for predictor number 1, the conditional branch predictor mispredicts 6.24% of the conditional branches. The miss ratio is the ratio of the number of branches mispredicted by the predictor to the total number of mispredicts. For example, the conditional branch predictor is responsible for 835 out of every 1000 mispredicts generated by predictor number 1. If branch mispredicts bottleneck a machine's performance, the miss ratios indicate which portion of the mispredicts each predictor is responsible for. Predictors that are responsible for large numbers of mispredicts should be targeted for further optimization.

The sizes given in Table 5.1 do not include the memory associated with the recovery mechanisms for branch prediction storage structures. (Fetching branches along a mispredicted path pollutes the branch prediction storage structures, and, if recovery mechanisms aren't provided to undo this pollution, increases the mispredict rate [62].) For the conditional branch predictors that are hybrids, each entry in the BHT for the SAg predictor component contains a branch history register *and* a single-bit lookahead prediction [134]. For the indirect branch predictors, the size of each entry in the table that contains the branch target addresses was assumed to be 32 bits. Although branch target addresses are 64 bits in the Alpha AXP architecture, only the lower 32 bits are stored in the predictor table. The upper 32 bits are taken from the current fetch address. In addition, the lower 2 bits of every branch target address are guaranteed to be 0, since memory is byte addressable, and instructions are aligned at 4-byte boundaries. One of these 2 bits is used to store the hysteresis bit. For the RAS (subroutine return predictor), the size of each entry was also assumed to be 32 bits. Only the lower 32 bits of an address were stored in the RAS. The upper 32 were taken from the fetch address.

5.3.3 The Execution Core

In the RDF model of execution, the execution core is perfect. That is, it consists of an unbounded number of functional units. Since there are an unbounded number of functional units, each instruction in the active window may be assigned to its own functional unit. As soon as an instruction's flow dependencies have been resolved, it is scheduled for execution on its assigned functional unit. It is guaranteed that the assigned functional unit will be available (i. e., not busy), since no other instructions use it. As a result, execution of the instruction commences without delay.

Practical CPU designs have more realistic execution cores. Each of their execution cores consists of a bounded number of functional units. Bounding the number of functional units limits the amount of execution bandwidth the core can provide, and limiting the amount of execution bandwidth bottlenecks performance.

To determine the severity of this bottleneck, I simulated execution cores of various sizes. The size of an execution core specifies the number of functional units it contains. An execution core of size N consists of N fully pipelined functional units, where each functional unit is capable of performing every desired operation. [5] Each cycle, the oldest N ready instructions are scheduled for execution. (This execution core was modeled by setting the dispatch rate to N. The dispatch rate is the maximum number of instructions that can be scheduled for execution on functional units in a single cycle.)

This *oldest first* scheduling heuristic gives higher priority to the instructions that appear earlier in the dynamic instruction stream; that is, it gives higher priority to those instructions that will be retired first. It attempts to hasten instruction retirement by greedily selecting the oldest N ready instructions. Hastening retirement causes the window to empty quicker.

However, the greedy decision made by the oldest first scheduling heuristic —i. e., select the oldest N ready instructions—is not always the best decision. A ready instruction that is not one of the oldest N ready instructions may have a large number of instructions that are dependent on it. It may also belong to a dependency chain that resolves a branch, or to a critical dependency chain in the program. Because this instruction is not (initially) selected by the oldest first scheduling heuristic, the resolution of its dependent instructions' flow dependencies will be delayed. As a result, in future cycles, there may be fewer ready instructions. If the instruction belongs to a dependency chain that resolves a branch, the resolution of a mispredicted branch may be delayed. If it belongs to a critical dependency chain in the program, the execution of that chain will be delayed. In all these cases, performance may suffer because the decision made by the oldest first scheduling heuristic was not the best possible decision.

The optimal scheduling algorithm always makes the best possible decision when deciding which N of the ready instructions should be scheduled for execution on functional units. The best possible decision is the one which minimizes the overall execution time. This decision can only be made by knowing the *entire* dataflow graph for the program being executed. A program's dataflow graph can only be constructed when all the instructions that comprise the program's dynamic instruction stream are known, and when all the dependencies between those instructions are known. The instructions that comprise the

[5] I assume that a machine with a properly selected, cost-effective functional unit configuration performs almost as well as a machine with the ideal configuration; i. e., the configuration where all functional units are fully pipelined and capable of performing every desired operation. Jourdan, Sainrat, and Litaize [61] found that for an issue rate of eight instructions per cycle, the performance of a machine with a cost-effective configuration, one where only half the functional units could execute loads and stores, was only 9% lower than that of a machine with the ideal configuration.

program's dynamic instruction stream can only be known after all the branches in the program have executed. All the dependencies between those instructions can only be known after the addresses of all loads and stores have been computed. This information is almost never known when the hardware is making scheduling decisions. Since the hardware needs this information to construct the program's dataflow graph, and the hardware needs the dataflow graph to implement the optimal scheduling algorithm, it is impossible to implement the optimal scheduling algorithm in hardware. In addition, the optimal scheduling algorithm is NP-hard [6], so implementing it in software (or a simulator) is not feasible.

Due to Amdahl's law [5], the performance of a machine with an execution core consisting of N functional units can be significantly less than N IPC, even if the optimal scheduling algorithm is used, and if the performance of a machine with a perfect execution core is greater than N IPC. Consider a program that consists of 40 dynamic instructions. The execution latency of each of the 40 instructions is one cycle. The first 8 instructions are serially dependent; i. e., instruction I_8 is dependent on instruction I_7, instruction I_7 is dependent on instruction I_6, $\ldots$, and instruction I_2 is dependent on instruction I_1. Regardless of the number of functional units, these instructions require 8 cycles to execute. The last 32 instructions (I_9–I_{40}) are only dependent on instruction I_8. On a machine with an execution core consisting of 2 functional units, 16 cycles are required to execute the last 32 instructions. For this machine, 24 cycles are required to execute the entire program, so the performance is 1.67 IPC (40 instructions ÷ 24 cycles). On a machine with a perfect execution core, the last 32 instructions are all executed in the same cycle. The performance of this machine is 4.44 IPC (40 instructions ÷ 9 cycles). Hence, the performance of the machine with an execution core consisting of 2 functional units was less than 2 IPC, even though the optimal scheduling algorithm was used and the machine with a perfect execution core could exploit enough parallelism to obtain a performance of 4.44 IPC.

[6] The problem of finding the optimal schedule is at least as hard as the PRECEDENCE CONSTRAINED SCHEDULING problem [38], which is NP-complete. For a set of tasks, a partial order on the tasks, a fixed number of processors, and an overall deadline, the PRECEDENCE CONSTRAINED SCHEDULING problem determines whether there is a schedule for running the tasks on the processors that meets the deadline and obeys all the ordering (or precedence) constraints. For the problem of finding the optimal schedule, the instructions (or nodes) in the dataflow graph are the tasks, the dependencies between those instructions specify the ordering constraints, and each functional unit corresponds to a processor.

Figure 5.8 shows the performance averaged over all the benchmarks for several different machines. Each point on the line labeled “Real Execution Core” plots the performance of an ideal machine that has been augmented with a real execution core of a given size. The sizes of the execution cores, which are listed on the horizontal axis, were varied from 1 to 32. The line labeled “Perfect Execution Core” is for the ideal machine with a perfect execution core. It provides an upper bound on the performances of the machines with real execution cores.

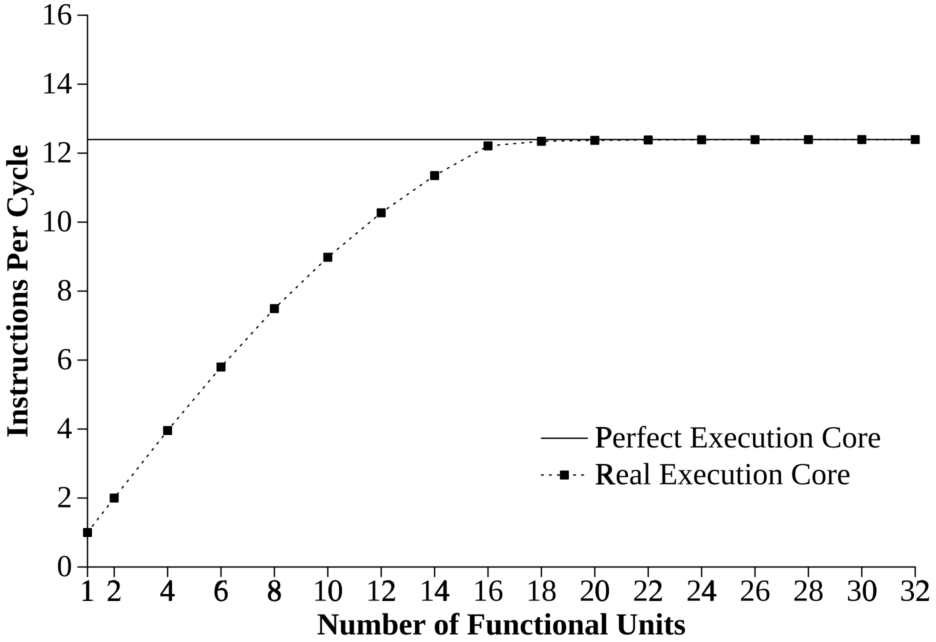


Figure 5.8: Ideal Machine with Varied Execution Core Size—Harmonic Average

As a consequence of using the oldest first scheduling heuristic rather than the optimal scheduling algorithm, and as a consequence of Amdahl’s law, the performance of a machine with an execution core of size N is always less than N IPC. The performance of the machine with an execution core of size 1 is 1.00 IPC (this number has been rounded to three significant digits). Since this machine has only one functional unit, its peak performance is 1 IPC. Thus, its actual performance is (nearly) equal to its peak performance. The performance of the machine with an execution core of size 16 is 12.21 IPC, which is only a 1.5% drop in performance from the machine using the perfect execution core. Thus, for a 16 wide issue machine, an execution core consisting of 16 functional units can eliminate the performance bottleneck that results from a lack of execution bandwidth. Finally, the performance of the machine with an execution core of size 32 is 12.39 IPC. This performance is the same as that of the machine with a perfect execution core. Results for the individual benchmarks are provided in Figure A.25 (SPEC benchmarks) and Figure A.26 (Non-SPEC benchmarks) of Appendix A.

Figure 5.9 shows the performance averaged over all the benchmarks for several machines. Each point on the line labeled "Real Execution Core" plots the performance of the real machine (i. e., the machine with the real instruction cache, branch predictor, execution core, and data cache) with an execution core of the size listed on the horizontal axis. The line labeled "Perfect Execution Core" is for the real machine that was given a perfect execution core. It provides an upper bound on the performances of the machines with real execution cores.

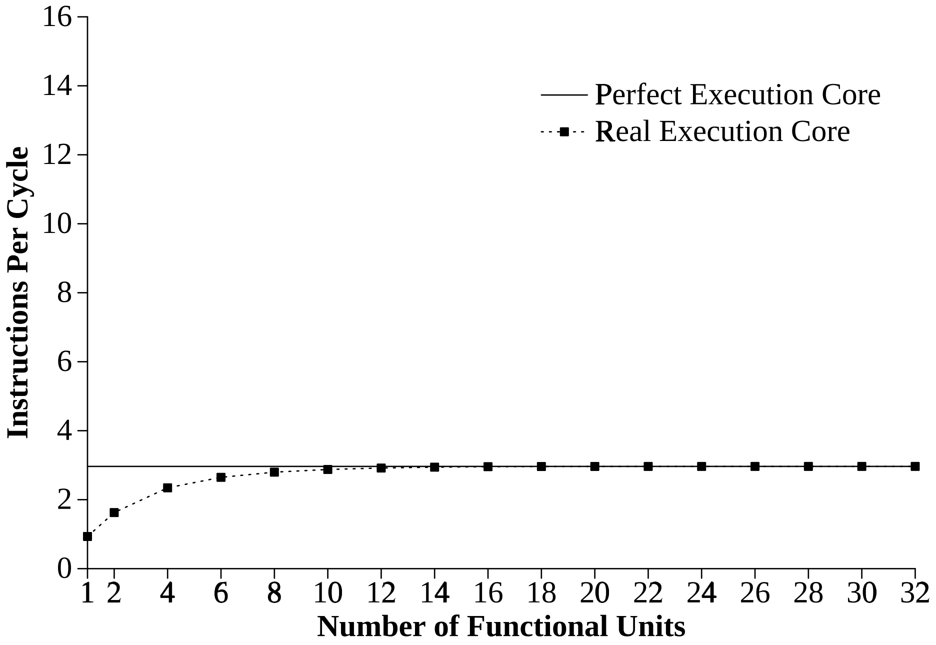


Figure 5.9: Real Machine with Varied Execution Core Size—Harmonic Average

The execution core for a real machine need not be as aggressive as the core for an ideal machine. The ideal machine requires a core of size 14 or greater to achieve 90% of the performance of the (ideal) machine with a perfect core (see Figure 5.8). The real machine only requires a core of size 8 to achieve 90% of the performance of the real machine with a perfect core. Results for the individual benchmarks are provided in Figure A.27 (SPEC benchmarks) and Figure A.28 (Non-SPEC benchmarks) of Appendix A.

5.3.4 The Data Cache

Data caches are an effective way of reducing the amount of time required to execute a load instruction. When a load instruction finds its data in the cache, the latency of the load is equal to the time required to perform the address calculation plus the time required to perform the cache access. This latency is typically only 2 or 3 cycles. When a load instruction does not find its data in the cache, the data must be retrieved from the next level of the cache/memory hierarchy. This adds many extra cycles to the latency of the load in addition to the cycles required to perform the address calculation and probe the data cache. In the worst case, the data may have to be retrieved from main memory. With CPU clock rates quickly approaching 1 GHz, and DRAM access times between 30 and 120 nanoseconds [24], a minimum of 30–120 cycles are required just to access the DRAM. When other factors are considered, such as bus arbitration time and the time to transmit the data across the bus, the total memory access time can be much worse.

Data cache misses bottleneck machine performance because a data cache miss retards the retirement of the load that generated the miss, and because the instructions that are dependent on a load that misses must wait until the miss has been serviced before they can start executing [12]. A load cannot retire until it has completed execution. On a data cache miss, the load cannot retire until its data has been returned from the next level of the cache/memory hierarchy. Because instructions must be retired in order, when a load that has missed in the data cache becomes the oldest instruction in the active window, it blocks the retirement of all instructions until the miss has been serviced. If it takes a long time to service the miss, the window fills up, which prevents new instructions from being issued into the window, which degrades performance. Instructions that are dependent on a load that misses must wait until the miss has been serviced before they can start executing. If a dependent instructions belongs to a dependency chain that resolves a branch, the resolution of that branch will be delayed. This is particularly harmful if the branch has been mispredicted. A dependent instruction may belong to one of the program's critical dependency chains. Hence, a data cache miss may reduce the amount of exploitable parallelism in the program.

Out-of-order execution and non-blocking data caches—which are used for all of the experiments in this dissertation—can be used to reduce the performance penalty that results from data cache misses. On a processor that executes instructions in order, the instructions following a load that misses in the data cache cannot execute until the miss has been serviced. On a processor with out-of-order execution, the instructions following the load are free to execute as long as they are not dependent on the result of the load. Thus, out-of-order execution allows the processor to hide some of the penalty associated with the cache miss by allowing it to execute instructions that are independent of the load. Blocking data caches lockup on a cache miss. That is, they don't accept any new requests until the miss has been serviced and the request that caused the miss has been satisfied. Non-blocking data caches [69] don't lockup on a cache miss. They continue to accept new requests while the miss is being serviced. If the new requests hit in the data cache, their data is returned to their associated loads. If the new requests miss, their misses can be serviced in parallel with the original miss. Hence, a non-blocking data cache reduces the cost of a miss by allowing new requests to be processed concurrently with the servicing of the miss.

To determine the severity of the bottleneck that results from data cache misses, I simulated several different machines. Figure 5.10 shows their performances averaged over all the benchmarks. There are five lines. One line shows the performance of the ideal machine with a perfect (100 percent hit rate) data cache. This machine provides an upper bound on the performances of machines with real data caches. The remaining lines plot the performances of ideal machines that have been augmented with real data caches. Each line represents a set of machines that all have the same penalty for a data cache miss. The miss penalty is a function of the cycle time and cache/memory hierarchy. It varies from implementation to implementation, so I have varied it from 6 to 32 cycles. Each point in a line plots the performance of an ideal machine with a data cache of the size listed on the horizontal axis. The data cache size was varied from 16k bytes to 1M byte. The data cache access required one cycle regardless of its size. That is, its access time was not scaled with its size. Because of this, the load latency on a cache hit was a constant 2 cycles (one cycle was required for the address calculation, and one cycle was required for the cache access) for all machines.

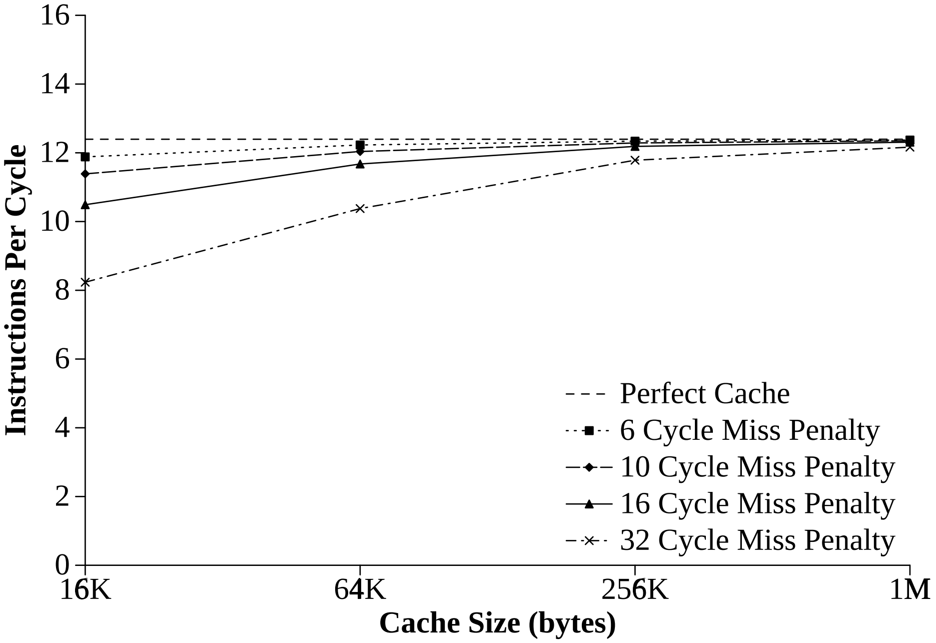


Figure 5.10: Ideal Machine with Varied Data Cache Size (Constant Load Latency)—Harmonic Average

Not surprisingly, the machines with larger data caches experience fewer data cache misses, and therefore have better performance. For the machines with 6 cycle miss penalties, performance increased from 11.88 IPC to 12.37 IPC as the data cache size was increased from 16k bytes to 1M byte. For the machines with 32 cycle miss penalties, performance increased from 8.23 IPC to 12.16 IPC. Also, the machines with low miss penalties have better performance than the machines with high miss penalties. For the machines with 16k byte data caches, performance increased from 8.23 IPC to 11.88 IPC as the miss penalty was decreased from 32 cycles to 6 cycles. Note that if the miss penalty is reduced all the way to 0 cycles, the load latency on a cache miss is identical to the load latency on a cache hit. As a consequence, the performance of a machine with a 0 cycle miss penalty, and a data cache of any size, would be identical to that of the machine with a perfect data cache. These results indicate that, for miss penalties of 16 cycles or less, and for the benchmarks I studied, the bottleneck that results from data cache misses is not very severe. The worst case was for the machine with a 16k byte data cache with a 16 cycle miss penalty. The performance of this machine was only 15% worse than the performance of the machine with a perfect data cache. For a miss penalty of 32 cycles, the bottleneck was severe for the machine with a 16k byte data cache, but not for the machines with data caches larger than 16k bytes. The machine with a 16k byte data cache lost 34% of its potential performance due to data cache misses. The machines with data caches larger than 16k bytes lost at most 16% of their potential performance. Results for the individual benchmarks are provided in Figure A.29 (SPEC) and Figure A.30 (Non-SPEC) of Appendix A.

Figure 5.11 shows the performance averaged over all the benchmarks for the same machines that were used to generate Figure 5.10, except that for these machines, the access time of the data cache was scaled with its size. The access time was 1 cycle at 16k bytes, 2 cycles at 64k bytes, 4 cycles at 256k bytes, and 8 cycles at 1M byte. Because the access time was scaled, the load latency on a cache hit was also scaled. The load latency on a cache hit was equal to one cycle, which was required to perform the address calculation, plus the number of cycles required to access the data cache. The load latency on a cache miss was equal to one cycle (for the address calculation), plus the number of cycles required to probe (i. e., access) the data cache, plus the data cache miss penalty. The machine with the perfect data cache is supposed to represent the ideal solution to the data cache bottleneck. The ideal solution is a data cache that has a 100 percent hit rate and the smallest possible access time. I assumed one cycle was the smallest possible access time. For this reason, the machine with the perfect data cache could alway access its cache in a single cycle, so its load latency was always 2 cycles.

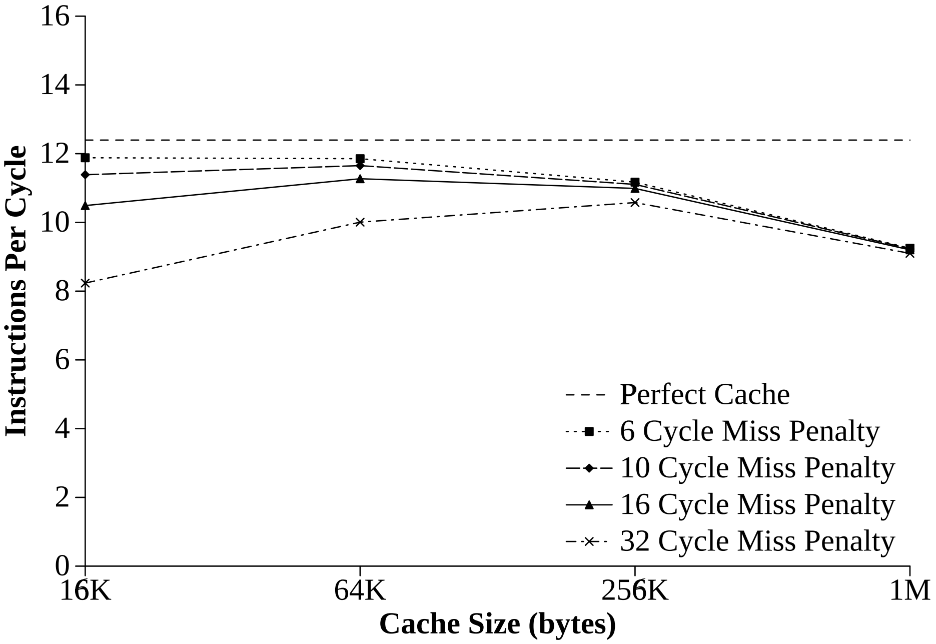


Figure 5.11: Ideal Machine with Varied Data Cache Size (Scaled Load Latency)—Harmonic Average

As the data cache size increases, fewer data cache misses occur, which reduces the severity of the data cache bottleneck. Unfortunately, the load latency on cache hits (and on cache misses) also increases as the data cache size increases. Increasing the load latency bottlenecks performance in the same way that data cache misses bottleneck performance: it retards the retirement of loads, and it delays the execution of instructions that are dependent on loads. Beyond a certain point, increasing the size of the data cache will result in lower performance, because the cost of the increased load latency will outweigh the benefit of the reduced number of cache misses. For the machines with 32 cycle miss penalties, increasing the size of the data cache beyond 256k bytes results in lower performance. For the machines with 6 cycle miss penalties, the data cache bottleneck is much less severe, so the point at which the performance is harmed if the cache size is increased occurs at a smaller data cache size. For these machines, increasing the size of the data cache beyond even 16k bytes results in lower performance. Results for individual benchmarks are provided in Figure A.31 (SPEC) and Figure A.32 (Non-SPEC) of Appendix A.

Figure 5.12 shows the performance averaged over all the benchmarks for several machines. One line shows the performance of the real machine (i. e., the machine with the real instruction cache, branch predictor, execution core, and data cache) that was given a perfect data cache. This machine provides an upper bound on the performances of machines with real data caches. The remaining lines plot the performances of real machines with real data caches. For these experiments, the load latency on a cache hit was a constant 2 cycles. That is, the data cache access time (and hence the load latency) was not scaled with the data cache size.

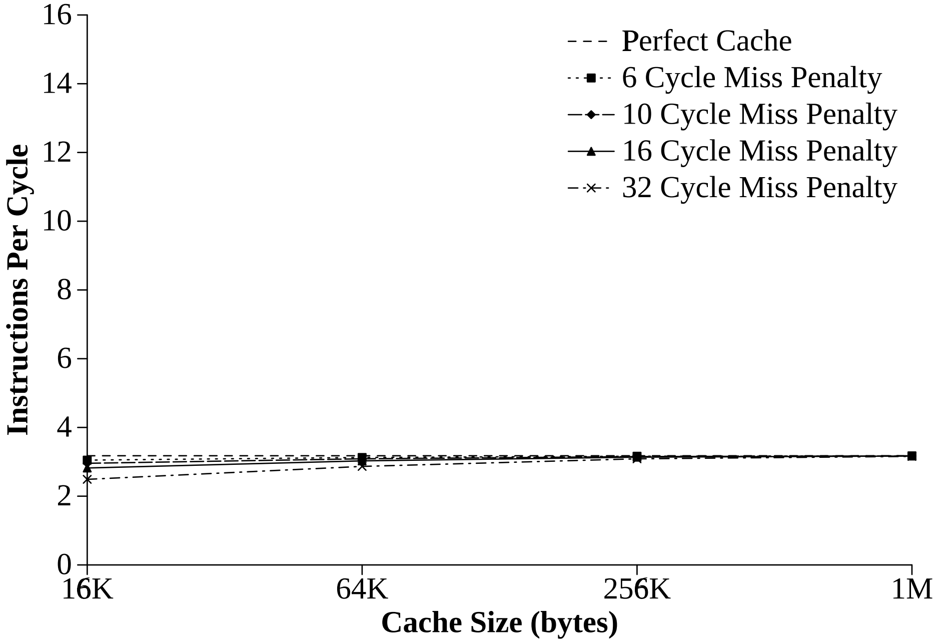


Figure 5.12: Real Machine with Varied Data Cache Size (Constant Load Latency)—Harmonic Average

The data cache for a real machine need not be as aggressive as for an ideal machine. The ideal machine with a 16k byte data cache loses between 4% (6 cycle miss penalty) and 34% (32 cycle miss penalty) of its potential performance due to data cache misses. For the same size data cache, the real machine loses between 4% (6 cycle miss penalty) and 22% (32 cycle miss penalty) of its potential performance. Additionally, the real machine is less sensitive to increasing miss penalty. When the miss penalty is increased from 6 cycles to 32 cycles, the performance of the ideal machine with a 16k byte data cache falls by 31%. For the same size data cache, the performance of the real machine only falls by 18%.

The real machine is hampered by bottlenecks other than the bottleneck that results from data cache misses. As a result, data cache misses have less of an overall impact on the real machine than they do on the ideal machine, which is *only* hampered by the data cache bottleneck. The bottleneck that results from data cache misses is more severe than the bottleneck that results from a lack of execution bandwidth (assuming that a processor that can issue 16 instructions per cycle has between 8 and 16 functional units), but less severe than the bottlenecks that result from branch mispredictions and instruction cache misses. Branch mispredictions and instruction cache misses are catastrophic events: they prevent the instructions that follow the mispredicted branch, or the instructions that follow the instruction that missed in the instruction cache, from being issued and executed until the mispredict has been resolved or the cache miss has been serviced. Data cache misses, on the other hand, do not prevent the instructions that logically follow the load that missed from being issued and executed. Note that data cache misses worsen the bottleneck that results from branch mispredictions. Loads feed dependency chains that resolve branches. When a load misses in the data cache, it may delay the resolution of a mispredicted branch. Hence, eliminating data cache misses can be very important, since branch mispredictions are a major performance limiter. Results for the individual benchmarks are provided in Figure A.33 (SPEC benchmarks) and Figure A.34 (Non-SPEC benchmarks) of Appendix A.

Finally, Figure 5.13 shows the performance averaged over all the benchmarks for the same machines that were used to generate Figure 5.12, except that for these machines, the access time of the data cache was scaled with its size. For the machines with 6 cycle miss penalties, the highest performing machine had a 16k byte data cache. For the machines with miss penalties greater than 6 cycles, the highest performing machines were those with 64k byte data caches. Results for the individual benchmarks are provided in Figure A.35 (SPEC) and Figure A.36 (Non-SPEC) of Appendix A.

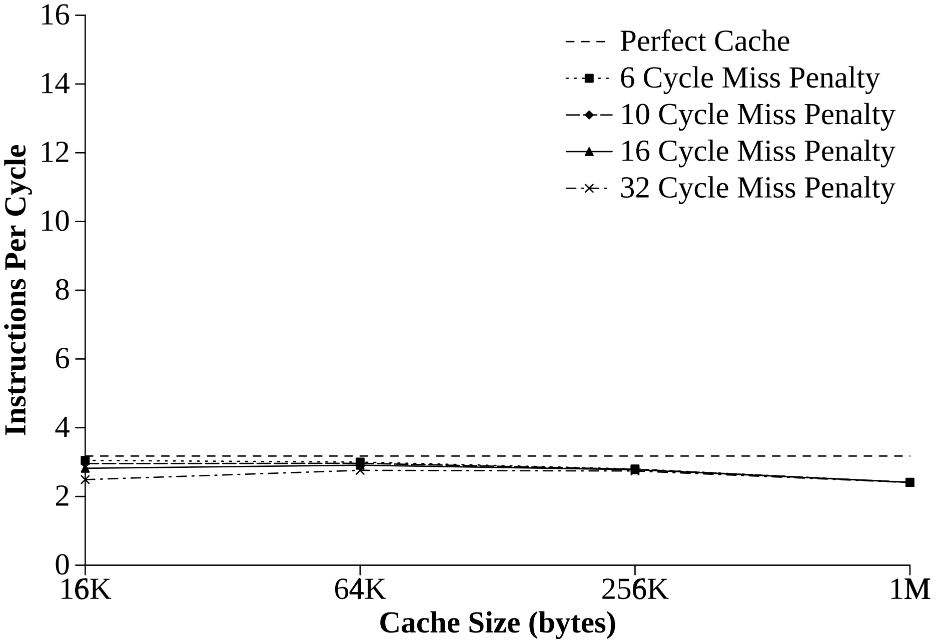


Figure 5.13: Real Machine with Varied Data Cache Size (Scaled Load Latency)—Harmonic Average

5.4 Summary

Figure 5.14 provides a summary of the four performance bottlenecks. It plots the performance averaged over all the benchmarks for 10 different machines. The solid line shows the performance of the ideal machine; i. e., the machine with a perfect instruction cache, a perfect branch predictor, a perfect execution core, and a perfect data cache. The dotted line shows the performance of the real machine; i. e., the machine with the real instruction cache, the real branch predictor, the real execution core, and the real data cache. The performance of the ideal machine, which has none of the four performance bottlenecks, is 12.4 IPC. The performance of the real machine, which has all four performance bottlenecks, is 3.0 IPC. Thus, 76% of the potential performance is lost due to the four bottlenecks.

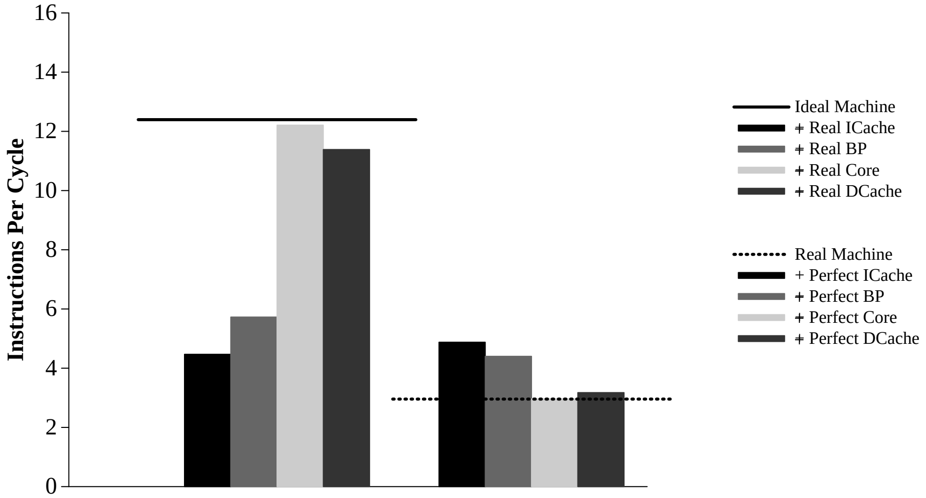


Figure 5.14: Performance Bottlenecks Summary—Harmonic Average

The ideal machine provides an upper bound on the performances of the other 9 machines. Underneath the solid line that plots the performance of this machine are four bars. Each of these bars plots the performance of the ideal machine when one of the bottlenecks is added. The bottleneck is added by trading the perfect 'X' used by the ideal machine for the real 'X' used by the real machine, where 'X' is either an instruction cache, branch predictor, execution core, or data cache. When a bottleneck is added, the performance drops according to the severity of the bottleneck. I will call this method of determining a bottleneck's severity (i. e., the method where a bottleneck's severity is determined by adding that bottleneck to the ideal machine) the *ideal machine method.* The bottlenecks, in decreasing order of severity, are the bottleneck due to instruction cache misses, the bottleneck due to branch mispredicts, the bottleneck due to data cache misses, and the bottleneck due to a lack of execution bandwidth. The performance degradations that result from these bottlenecks are 64%, 54%, 8%, and 1%, respectively.

The real machine provides a lower bound on the performances of the other 9 machines. The dotted line that plots the performance of this machine passes through four bars. Each of these bars plots the performance of the real machine when one of the bottlenecks is removed. The bottleneck is removed by trading the real 'X' used by the real machine for a perfect 'X', where 'X' is either a instruction cache, branch predictor, execution core, or data cache. When a bottleneck is removed, the performance increases according to the severity of the bottleneck. I will call this method of determining a bottleneck's severity (i. e., the method where a bottleneck's severity is determined by removing that bottleneck from the real machine) the *real machine method*. The bottlenecks, in decreasing order of severity, are the bottleneck due to instruction cache misses, the bottleneck due to branch mispredicts, the bottleneck due to data cache misses, and the bottleneck due to a lack of execution bandwidth. The performance increases that result from removing these bottlenecks are 65%, 49%, 7%, and 0%, respectively.

The ideal machine method for determining a bottleneck's severity does not account for the interactions between that bottleneck and other bottlenecks. The real machine method does. The ideal machine method is optimistic: it assumes that when the machine is built, there will be viable solutions for all bottlenecks except for the bottleneck in question. The severity of a bottleneck is determined by assuming that the machine will have no other bottlenecks. Because the severity is determined in this way, this method does not account for the interactions between bottlenecks. (An example of an interaction between bottlenecks is that data cache misses worsen the bottleneck that results from branch mispredictions. This example was discussed in Section 5.3.4). The real machine method, on the other hand, is pessimistic: it assumes that when the machine is built, there will not be viable solutions for bottlenecks other than the bottleneck in question. The severity of a bottleneck is determined by assuming that the machine will have all of the other bottlenecks. Because the severity is determined in this way, this method accounts for the interactions between bottlenecks.

In Figure 5.14, the most severe bottleneck was the bottleneck due to instruction cache misses. This was true regardless of the method used to determine the severity of the bottlenecks. In addition, the order of the bottlenecks, in terms of severity, was the same regardless of the method used to rank them. This is not always the case. For the li benchmark (shown in Figure 5.15 along with the rest of the SPEC benchmarks) and the chess benchmark (shown in Figure 5.16 along with the rest of the Non-SPEC benchmarks), the order of the bottlenecks depends on the method used to rank them.

The bottleneck due to branch mispredicts is extremely severe for both these benchmarks. Branch mispredicts are more costly for the real machines than they are for the ideal machines, because the real machines have real data caches, and data cache misses increase the resolution time of mispredicted branches. Because of this, the real machine method may assign a higher level of "severity" to the bottleneck due to branch mispredicts than the ideal machine method does. This phenomenon can be seen in the chess benchmark: the ideal machine method ranks the bottleneck due to branch mispredicts as less severe than the bottleneck due to instruction cache misses, but the real machine method ranks the bottleneck due to branch mispredicts as more severe than the bottleneck due to instruction cache misses. Also note that adding a perfect data cache to the real machine not only eliminates the bottleneck due to data cache misses, it also reduces the resolution time of mispredicted branches, which reduces the severity of the bottleneck due to branch mispredicts. Because of this, the real machine method may assign a higher level of "severity" to the bottleneck due to data cache misses. This phenomenon can be seen in the li benchmark: the ideal machine method ranks the bottleneck due to data cache misses as less severe than the bottleneck due to instruction cache misses, but the real machine method ranks bottleneck due to data cache misses more severe than the bottleneck due to instruction cache misses.

For the cmp benchmark (see Figure 5.15), performance only falls by 2% when the real data cache is added to the ideal machine, which seems to indicate that data cache misses do not significantly bottleneck performance. Yet, when a perfect data cache is added to the real machine, performance increases by 21%, which indicates that data cache misses are a significant bottleneck. This discrepancy is again caused by the interaction between two bottlenecks: data cache misses increase the resolution time of mispredicted branches, which worsens the bottleneck due to branch mispredicts. The real machine with the perfect data cache has a real branch predictor. Adding the perfect data cache to the real machine reduces the resolution time of mispredicted branches, which reduces the severity of the bottleneck due to branch mispredicts. The ideal machine with the real data cache has a perfect branch predictor, so data cache misses don't worsen the bottleneck due to branch mispredicts, since there are no branch mispredicts. Note that the bottleneck due to branch mispredicts is severe for the cmp benchmark, so reducing the resolution time of mispredicted branches yields a large performance gain.

The results in this chapter show that the bottleneck due to instruction cache misses and the bottleneck due to branch mispredicts are significant, whereas the bottleneck due to a lack of execution bandwidth and the bottleneck due to data cache misses are not. The bottleneck due to instruction cache misses is more severe than the bottleneck due to branch mispredicts. The performance of the ideal machine drops by 64% when the bottleneck due to instruction cache misses is added, and the performance of the real machine increases by 65% when that bottleneck is removed. For the bottleneck due to branch mispredicts, these percentages are 54% and 49%, respectively. In the remaining chapters of this dissertation, I will show how out-of-order fetch, decode, and issue reduces the bottleneck due to instruction cache misses.

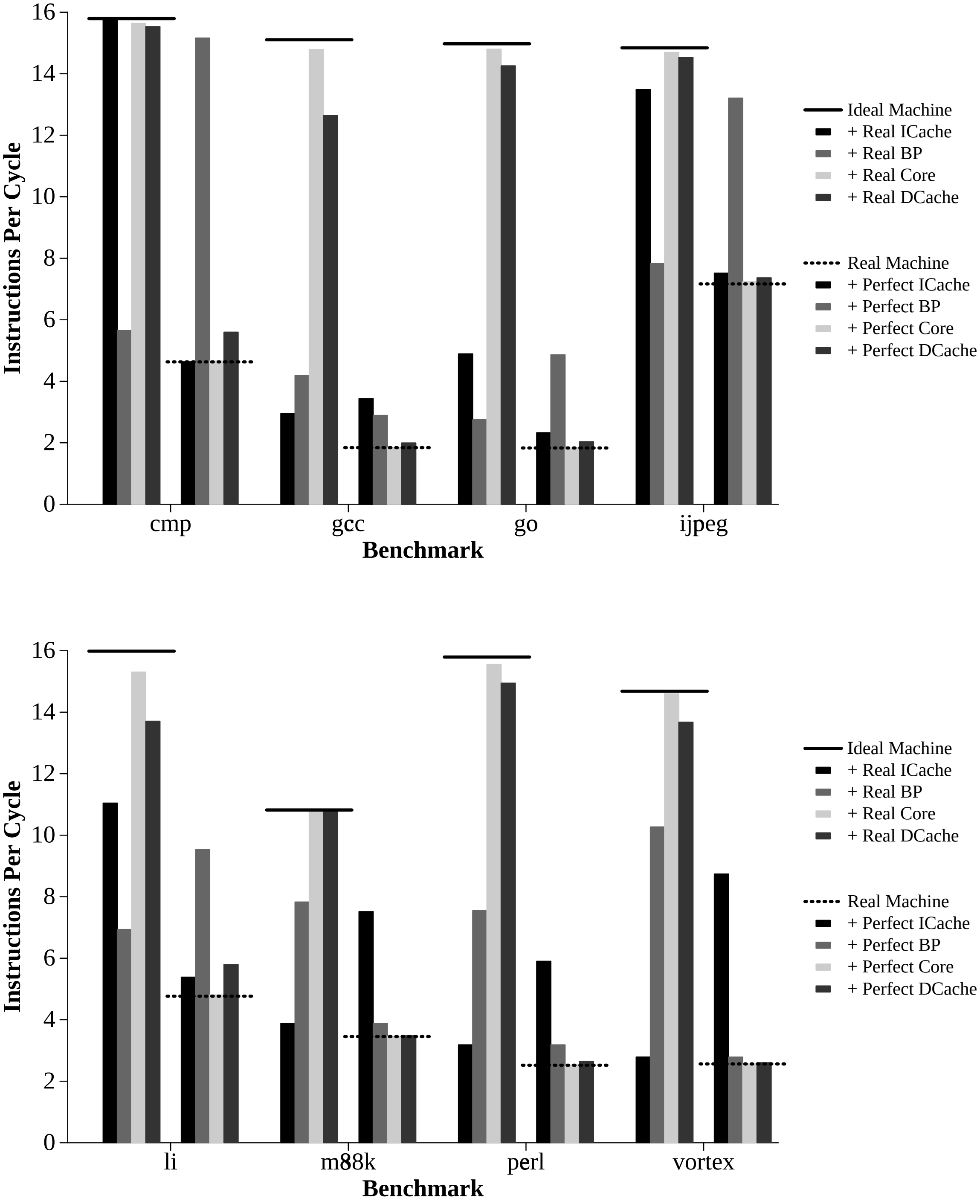


Figure 5.15: Performance Bottlenecks Summary—SPEC Benchmarks

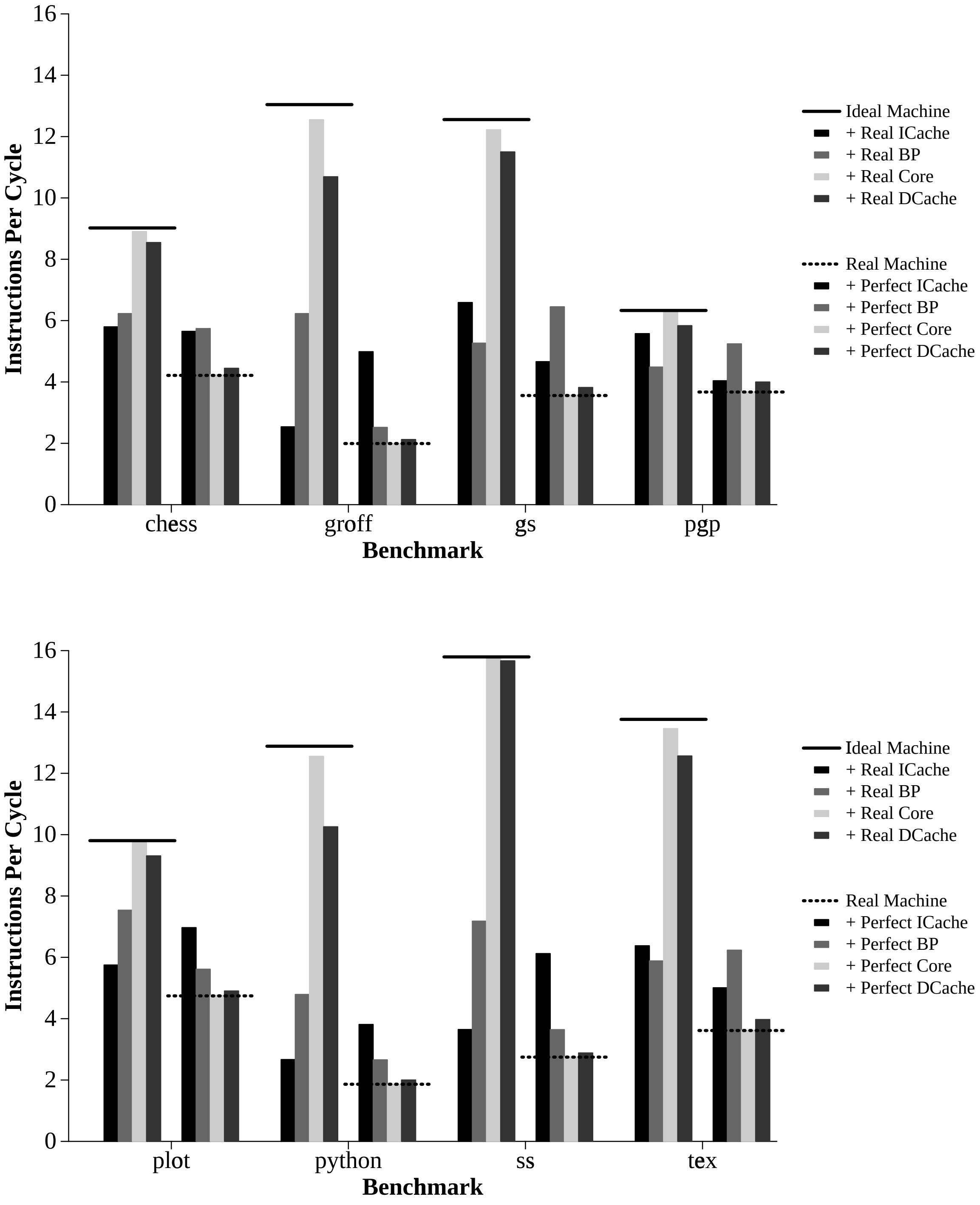


Figure 5.16: Performance Bottlenecks Summary—Non-SPEC Benchmarks

CHAPTER 6

Out-of-Order Fetch, Decode, and Issue: Concept and Preliminary Results

Chapter 5 showed that instruction cache misses significantly bottleneck performance. This chapter shows a way to reduce this bottleneck: out-of-order fetch, decode, and issue.

The chapter first reintroduces the concept of out-of-order fetch and the concept of out-of-order fetch/decode/issue (both concepts are variants of the concept of out-of-order fetch, decode, and issue), then describes the problem of unknown register dependencies that results from out-of-order fetch/decode/issue, and then describes some possible solutions to this problem. It also presents some preliminary results that show that the instruction cache bottleneck for a sixteen wide issue machine can be nearly eliminated with out-of-order fetch/decode/issue.

These preliminary results were generated using the RDF simulator. The RDF simulator models an abstract machine: an RDF machine that has been augmented with a real instruction cache, a real branch predictor, a real execution core, and a real data cache. One major drawback of this simulator is that it cannot distinguish between fetch and issue. As a result, only out-of-order fetch/decode/issue can be modeled with this simulator. Out-of-order fetch cannot be modeled. The RDF simulator is used to calculate the performance potential of out-of-order fetch/decode/issue. It is also used to determine which solutions to the unknown register dependency problem are worthwhile. Only the worthwhile solutions are implemented in the full simulator. The actual performance benefits of both out-of-order fetch and out-of-order fetch/decode/issue are calculated using the full simulator, which models a more realistic machine. These results are presented in Chapter 8.

6.1 Concept

Out-of-order fetch reduces the performance penalty caused by instruction cache misses. I will use the term *fetch block* to refer to the group of instructions that are brought into the processor by an instruction cache fetch. Upon encountering an instruction cache miss, a conventional processor will wait until the instruction cache miss is serviced before continuing to fetch, decode, and issue any new fetch blocks. A processor with out-of-order fetch temporarily ignores the block associated with the instruction cache miss, and attempts to fetch the blocks that follow that block. The addresses of these blocks are generated by the branch predictor. (I will use the term *branch predictor* to refer to all the next fetch address generation logic.) Because the branch predictor does not depend on the instructions in the current block to generate the address of the next block to be fetched, it can continue to make predictions even when the current block misses in the instruction cache. Thus, the processor can skip the block that missed in the instruction cache and continue fetching the following block using the address generated by the branch predictor. After the instruction cache miss is serviced, the processor can fetch, decode, and issue the skipped block.

Consider the example in Figure 6.1. It shows a graph consisting of five fetch blocks A–E. Assume that every cycle the processor is able to fetch a new block. In the first cycle, the processor fetches block A and the predictor generates address B for the next block. Suppose in the second cycle, the processor's fetch of block B results in an instruction cache miss. The predictor can still generate the address for the next block, block C. The processor will fetch nothing in the second cycle, but can attempt to fetch block C in the third cycle, followed by block D in the fourth cycle (regardless of whether C hit in the cache or not). Once the instruction cache miss is serviced, the processor can return to the point of the miss and fetch, decode, and issue the skipped block(s), starting with block B. Thus, out-of-order fetch allows the processor to fetch useful work in the presence of instruction cache misses. In the worst case, where all the following fetches result in instruction cache misses, out-of-order fetch still provides the performance benefit of prefetching.

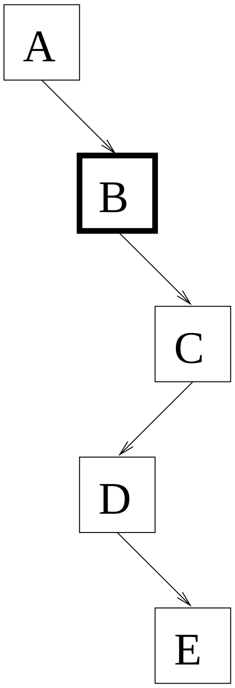


Figure 6.1: Out-of-Order Fetch Example

Out-of-order fetch/decode/issue also reduces the performance penalty caused by instruction cache misses by allowing the processor to fetch useful work in the presence of instruction cache misses. For conventional processors, processors with out-of-order fetch, and processors with out-of-order fetch/decode/issue, the branch predictor predicts the sequence of fetch blocks that comprise the dynamic instruction stream. In conventional processors, fetch requests for these blocks are initiated and completed in program order. As the fetch requests complete, they write their data into the processor's fetch buffer. For conventional processors, blocks are always inserted into the fetch buffer in the predicted program order. In processors with either out-of-order fetch or out-of-order fetch/decode/issue, fetch requests are initiated in program order, but may complete out of program order if there are instruction cache misses. As a result, fetch blocks may be inserted into the fetch buffer out of order. What distinguishes out-of-order fetch from out-of-order fetch/decode/issue is the order in which fetch blocks may be *removed* from the fetch buffer. For out-of-order fetch, blocks are always removed from the fetch buffer in program order. For out-of-order fetch/decode/issue, blocks may be removed from the fetch buffer out-of-order. Blocks are decoded and issued into the reservation stations in the order that they are removed from the fetch buffer. Hence, for out-of-order fetch, instructions are always decoded and issued in program order. For out-of-order fetch/decode/issue, instructions may be decoded and issue out of program order.

The example in Figure 6.1 assumed that the processor could only fetch one block per cycle. Restricting fetch to one block per cycle severely limits the performance of wide issue processors. Recent research has discovered several techniques for allowing a processor to fetch multiple blocks per cycle. These techniques include VLIW tree instructions [90], branch alignment [16], trace scheduling [34], superblocks [51], hyperblocks [74, 75], block-structured ISAs [46, 47, 82–84], trace caches [35, 84, 85, 98, 102, 108] [1], branch address caches [133], collapsing buffers [23], subgraph predictors [31], multiple-block ahead predictor [111], and path address caches [87]. Many of these techniques (for example, block-structured ISAs and trace caches) store multiple copies of the same fetch block in the cache. This exacerbates the instruction cache bottleneck. Other techniques (for example, subgraph predictors and path address caches) require branch predictors specialized for multiple block fetch. Throughout this dissertation, I use an idealized technique for multiple block fetch that does not require multiple copies of fetch blocks to be stored in the cache or require a specialized branch predictor.

[1] Throughout this dissertation, I assume that instructions are fetched from an instruction cache, and that out-of-order fetch and out-of-order fetch/decode/issue are used to lessen the performance penalty that results from instruction cache misses. However, if instructions are (primarily) fetched from a trace cache, out-of-order fetch and out-of-order fetch/decode/issue are used to lessen the impact of trace cache misses rather than instruction cache misses.

6.2 Creating the Hole

Conventional processors—and processors with out-of-order fetch—decode and issue instructions in program order. As the instructions are decoded, the dependencies communicated between instructions via registers are computed. Because the instructions are decoded in order, these dependencies can be computed exactly. Once the dependencies have been computed, register renaming is used to eliminate the anti and output dependencies.

Processors that implement out-of-order fetch/decode/issue have to deal with a problem that is similar to the unknown address problem, which was described in Section 4.2. Unlike the unknown address problem, which deals with the dependencies communicated between instructions via memory, this problem deals with the dependencies communicated via registers. Because the instructions are not always decoded in order, an instruction can be decoded and issued into the active window before the instructions it is dependent on. When this occurs, the instruction's register dependencies will be computed incorrectly. The instruction must then wait until the instructions it is dependent on have been decoded and issued before its register dependencies can be correctly computed and repaired.

When out-of-order fetch/decode/issue results in an instruction being issued that is dependent on an instruction that has not yet been issued into the active window, the machine's representation of the dataflow graph is said to have a "hole" in it. The hole is a piece of the dataflow graph about which the machine has no knowledge of the register dependencies. The dataflow graph may have multiple holes in it. Each hole in the dataflow graph corresponds to a sequence of fetch blocks that are older than the youngest fetch block in the active window and that are not in the active window because those blocks missed in the instruction cache. (Fetch block X is older than fetch block Y if, in the sequence of fetch blocks that comprise the dynamic instruction stream, block X occurs before block Y.)

For example, assume that the sequence of fetch blocks B_1–B_9 comprises a segment of the dynamic instruction stream. Block B_X is older than block B_Y if X is less than Y. Assume blocks B_2, B_3, B_6, B_7, and B_9 missed in the instruction cache, and their instructions have not been issued into the active window. Assume the instructions from the remaining blocks (B_1, B_4, B_5, and B_8) have been issued into the active window. There are two holes in the dataflow graph: one for the sequence of blocks B_2–B_3, and one for the sequence of blocks B_6–B_7. Block B_9 is not older than the youngest fetch block in the active window

(i. e., block B_8), so there is no hole corresponding to this block.

For now, consider the case where there is a single hole in the dataflow graph. The hole is comprised of all instructions from fetch blocks skipped over as a result of instruction cache misses. The instructions issued before the skipped blocks are called *pre-hole* instructions, the instructions from the skipped blocks are called *hole* instructions, and the instructions issued after the skipped blocks are called *post-hole* instructions. The hole is created as the fetch blocks are being skipped over. The hole is filled in as the cache misses are serviced and the missing instructions become available for decode and issue. The hole disappears after all the instructions from the skipped blocks have been decoded and issued into the active window. Note that each hole instruction uses two issue cycles: one for the creation of the hole, and one to fill the hole in.

Consider the example in Figure 6.2. Assume that blocks A, C, D, and E have been issued into the active window, and that block B, which suffered an instruction cache miss, has not yet been issued into the active window. The instructions from block A are the pre-hole instructions, the yet-to-be-issued instructions from block B are the hole instructions, and instructions from blocks C, D, and E are the post-hole instructions.

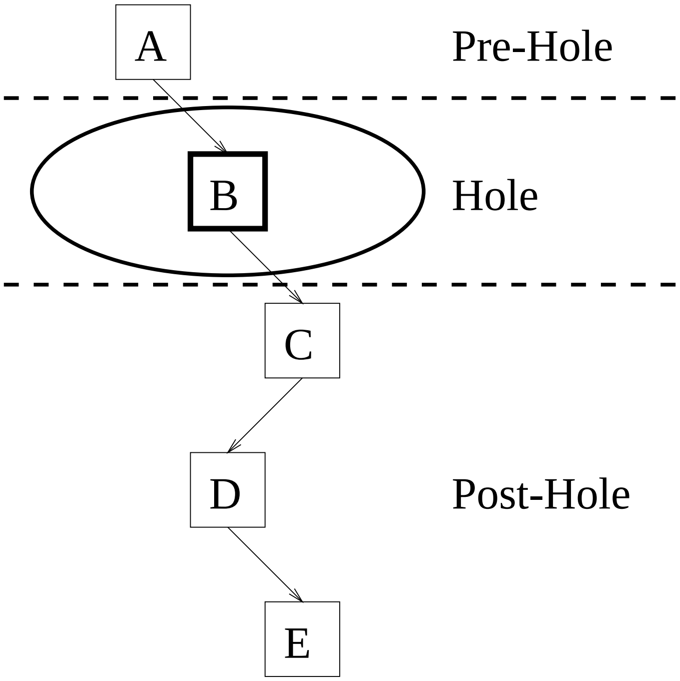


Figure 6.2: The Dataflow Graph Hole

When there are multiple holes in the dataflow graph, the terms pre-hole, hole, and post-hole are all used in reference to the *oldest* hole in the dataflow graph. (Hole X is older than hole Y if hole X corresponds to a sequence of fetch blocks that are older than the sequence of fetch blocks that correspond to hole Y.) The instructions issued before the skipped blocks that comprise the oldest hole are the pre-hole instructions, the instructions from the skipped blocks are the hole instructions, and the instructions issued after the skipped blocks are the post-hole instructions. When the oldest hole disappears, the next oldest hole (if there is one) becomes the new oldest hole, and new sets of instructions become the pre-hole, hole, and post-hole instructions. In Figure 6.2, if block D also suffered an instruction cache miss, and as a result has not yet been issued into the active window, the instructions from block A are still the pre-hole instructions, the instructions from block B are still the hole instructions, and the instructions from blocks C, D, and E are still the post-hole instructions.

6.3 Dependency Handling Techniques

I invented several techniques for solving the data dependency problems that result from having holes in the dataflow graph. Like the memory disambiguation techniques, which were described in Section 4.2, these techniques follow one of two basic paradigms: either the non-speculative paradigm or the speculative paradigm. Techniques in the non-speculative paradigm do not speculate on the dependencies of post-hole instructions. The techniques in the speculative paradigm do.

6.3.1 Non-Speculative Paradigm

Techniques in the non-speculative paradigm never execute a post-hole instruction until they know for certain which instructions the post-hole instruction is dependent on. A post-hole instruction that might be dependent on hole instructions is forced to wait until those hole instructions have been decoded and issued into the active window. Once its dependencies are known for certain, the post-hole instruction is allowed to execute. The advantage of using techniques in this paradigm is that, when a post-hole instruction executes, its source operands are guaranteed to contain the correct data. Because of this, when a post-hole instruction distributes its result, that result is guaranteed to contain the correct data. The disadvantage of using techniques in this paradigm is that some post-hole instructions are forced to wait for hole instructions they are not dependent on, which needlessly delays their execution and result distribution.

I invented three techniques that follow the non-speculative paradigm. I call these three techniques *assume dependence*, *classification*, and *mask cache.*

Assume Dependence

The assume dependence technique pessimistically predicts (or assumes) that all post-hole instructions are dependent on hole instructions. Post-hole instructions are not allowed to execute until the hole disappears. This unnecessarily delays the execution of all post-hole instructions that are not dependent on results generated by hole instructions. While this technique allows the fetch unit to perform prefetching during an instruction cache miss, and allows early insertion of the post-hole instructions into the reservation stations, the processor will not take advantage of any parallelism exposed by out-of-order fetch/decode/issue.

This technique deals with register dependencies in a way that is analogous to the way that the dependency matrix memory disambiguation technique deals with memory dependencies. For dependency matrix, a load is not allowed to distribute its result until the addresses of all previous memory writes are known. For assume dependence, a post-hole instruction is not allowed to execute and distribute its result until the addresses (i. e., register numbers) of all previous register writes are known.

Classification

With the classification technique, post-hole instructions are classified as either being independent of any hole instructions, or possibly dependent on hole instructions. An instruction is classified as independent if (1) it has no register source operands (e. g., all its source operands are literal constants specified by the instruction), or (2) the values for all of its register source operands are generated by other independent instructions. Otherwise, an instruction is classified as possibly dependent.

Consider the example in Figure 6.3. It shows a section of a dataflow graph with a hole in it. The arrows entering the blob labeled “HOLE INSTRUCTIONS” represent the register values produced by the pre-hole instructions. The blob represents the hole, and contains the hole instructions. The arrows that emanate from the blob represent the register values produced by either the pre-hole instructions or the hole instructions. Since there is a hole in the dataflow graph, it is impossible for the machine to determine which of these values are produced by the pre-hole instructions, and which are produced by the hole instructions. The instructions below the blob are the post-hole instructions.

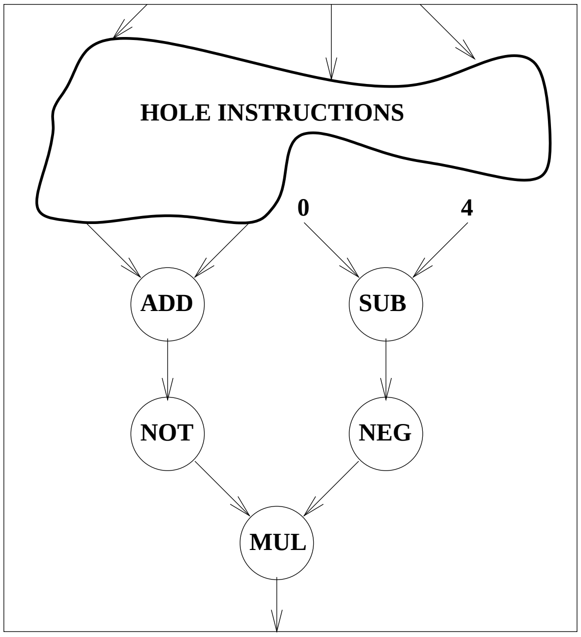


Figure 6.3: Classification Example

The SUB instruction has no register source operands. Both its source operands are literal constants. It is classified as independent. The NEG instruction has a single register source operand whose value is produced by the SUB instruction. Therefore, the NEG instruction is also classified as independent. The ADD instruction has two register source operands whose values are produced by either pre-hole or hole instructions. The ADD instruction is therefore classified as being possibly dependent. The NOT instruction consumes the value produced by the ADD instruction. Thus, the NOT instruction is also possibly dependent. Finally, the MUL instruction consumes the values produced by the NOT and NEG instructions. Since the NOT instruction is possibly dependent, the MUL instruction is also possibly dependent.

By definition, possibly dependent instructions might be dependent on hole instructions. These instructions are not allowed to execute until the hole disappears. This unnecessarily delays the execution of all possibly dependent instructions that are not dependent on results generated by hole instructions. Independent instructions, on the other hand, are guaranteed to be independent of any hole instructions. These instructions are allowed to execute as soon as they become ready. The advantage of this technique over the assume dependence technique is that some of the post-hole instructions (i. e., the instructions classified as independent) are allowed to execute before the hole disappears. This allows the processor to exploit some of the extra parallelism exposed by out-of-order fetch/decode/issue.

This technique deals with register dependencies in a way that is analogous to the way that the most aggressive memory disambiguation technique that follows the non-speculative memory disambiguation paradigm deals with memory dependencies. The memory disambiguation technique was described algorithmically by Patt et al. [101]. For this technique, a load that is dependent on a store is not allowed to distribute its result until the address of the memory write of the store, and the addresses of all memory writes younger than the store but older than the load are known. For the classification technique, a post-hole instruction that is dependent on instruction X is not allowed to execute and distribute its result until the address (i. e., register number) of the register write of instruction X, and the addresses of all register writes younger than instruction X but older than the post-hole instruction are known.

Mask Cache

For the mask cache technique, hardware keeps track of which registers are written by the hole instructions. The hardware uses this information to determine which post-hole instructions are dependent on hole instructions. The hardware prevents post-hole instructions that are dependent on hole instructions from being scheduled for execution. Post-hole instructions that are not dependent on hole instructions can be scheduled for execution once their flow dependencies have been satisfied. When the instructions that comprise the hole become available for decode and issue, the hardware fills in the hole and correctly establishes all the flow dependencies. Post-hole instructions that were dependent on hole instructions are scheduled for execution when the hole instructions they depend on have been decoded and issued, and all the post-hole instructions' flow dependencies have been satisfied.

To accomplish this, destination register identifiers are recorded in a separate structure, so that the flow dependencies can be identified. For handling an instruction cache miss, only the flow dependencies crossing the skipped fetch block must be represented. This is done by storing the dependency information for each of the instructions of the block in a separate cache, called the mask cache. If an instruction writes a register, the mask cache stores the architectural register number of this register. For the Alpha AXP ISA, which has 64 architectural registers, with two registers hard-wired to 0, 6 bits are needed to specify the dependency information for each instruction. To store an Alpha AXP instruction in the instruction cache requires 32 bits, which is approximately 5 times the number of bits required to store the dependency information of an Alpha AXP instruction in the mask cache.

When creating the hole, the dependency information read from the mask cache is used to rename the destination registers of the hole instructions. That is, even though the hole instructions are not available for decode and issue, their destination registers are renamed using the mask cache information. Later, when the hole instructions are available for decode and issue, and the hardware is filling the hole, the hardware will need to know how the architectural destination registers were mapped to physical registers when the hole was created in order to properly set up the rename tags of the instructions being inserted into the hole. Depending on the register renaming scheme, the hardware may need to save the rename tags in some structure when creating the hole in order to accomplish this.

Figure 6.4 shows the organization of an instruction fetch mechanism incorporating a mask cache for a scalar processor. For superscalar processors, the fetch mechanism must be able to provide multiple instructions per cycle. For the processors modeled in the remaining experiments in this dissertation, the fetch mechanism can provide up to 16 instructions per cycle. However, to make things easier to understand, the illustrated fetch mechanism can only fetch up to one instruction per cycle.

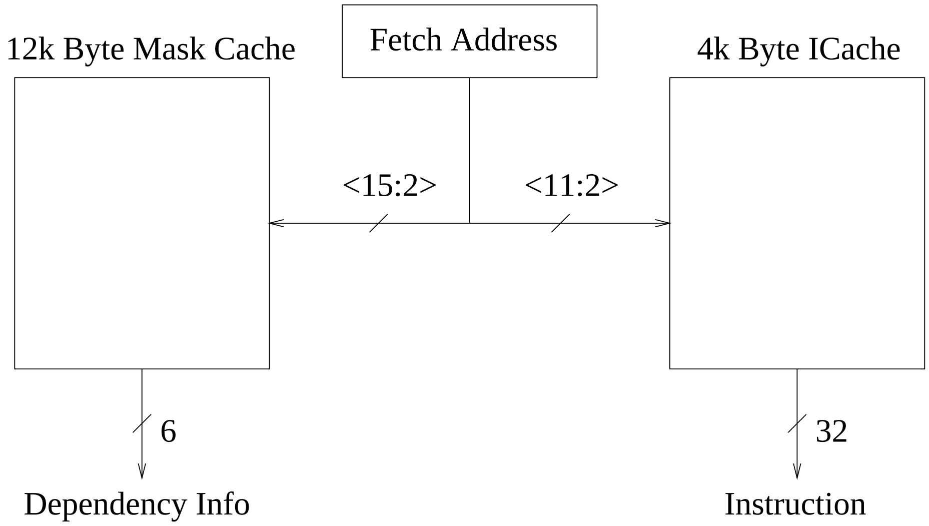


Figure 6.4: Instruction Fetch Mechanism Incorporating a Mask Cache

The illustrated instruction cache is 4k bytes and direct mapped. Each instruction is 32 bits, or 4 bytes, so the cache can hold 1k instructions. The dependency information for each instruction is stored in the mask cache. The dependency information indicates which architectural register, if any, is updated by the instruction. For the Alpha AXP ISA, which has 64 architectural registers, 6 bits are needed to record this dependency information. The mask cache illustrated is 12k bytes and direct mapped. Since the dependency information for one instruction requires only 6 bits of storage, the mask cache can hold the dependency information for 16k instructions.

The mask cache is accessed in parallel with the instruction cache. On an instruction cache hit, everything proceeds as in a conventional processor. If there is a miss in the instruction cache, but a hit in the mask cache, the dependency information from the mask cache is used to enable out-of-order fetch/decode/issue of the instructions that missed in the instruction cache. If there is a miss in both caches, the dependency information required for out-of-order fetch/decode/issue is not available. At this point, fetch, decode, and issue can stall, or one of the other dependency handling techniques (in either the non-speculative or the speculative paradigms), which don't require a mask cache for out-of-order fetch/decode/issue, can be used to enable out-of-order fetch/decode/issue. Table 6.1 summarizes the actions taken by the processor for all three cases. In the table, FDI stands for Fetch/Decode/Issue.

ICache	Mask Cache	Action
Hit	—	In-Order FDI
Miss	Hit	Out-of-Order FDI
Miss	Miss	Stall FDI *or* Out-of-Order FDI

Table 6.1: Summary of Actions for a Processor with a Mask Cache

The advantage of this technique over the assume dependence and classification techniques is that, as long as the dependency information for each of the hole instructions is resident in the mask cache, the execution of post-hole instructions is never unnecessarily delayed. This allows the processor to exploit all the extra parallelism exposed by out-of-order fetch/decode/issue. The primary disadvantage of this technique is that some of the processor's transistor budget must be allocated to the mask cache. This may reduce the number of transistors that can be budgeted for the instruction cache, which results in a smaller instruction cache with a higher miss rate. Hence, when out-of-order fetch/decode/issue is implemented using the mask cache technique, the number of instruction cache misses may increase. However, out-of-order fetch/decode/issue allows the processor to better tolerate instruction cache misses, which counteracts the impact of the extra instruction cache misses.

6.3.2 Speculative Paradigm

When it is uncertain whether or not a post-hole instruction is dependent on a hole instruction, techniques in the speculative paradigm will predict whether or not the post-hole instruction is dependent. If a post-hole instruction is predicted to be independent of any hole instructions, it is allowed to execute as soon as it becomes ready. If a post-hole instruction is predicted to be dependent, it is not allowed to schedule for execution until the hole instructions it depends on have been decoded and issued, and all of its flow dependencies have been satisfied. The machine can wait for the hole to disappear in order to determine which instructions the post-hole instruction is dependent on. Once the hole disappears, all the post-hole instruction's (register) dependencies can be fully resolved.

The prediction is verified when all of the post-hole instruction's dependencies have been resolved. If the prediction was correct, no further action is required. If the prediction was incorrect, there are two cases to consider. First, the post-hole instruction has not yet executed. Since all the post-hole instruction's dependencies are known at this point, the instructions that produce the values consumed by the post-hole instruction can be identified. The dependency information (e. g., rename tags) of the post-hole instruction is simply updated so that it identifies the correct producers. The only real harm done in this case is that the execution of the post-hole instruction was (possibly) delayed longer than it needed to be. For the second case, the post-hole instruction has already executed. Since the prediction was incorrect, the post-hole instruction probably executed using source operands that contained incorrect data. Because of this, the post-hole instruction may have distributed a result containing incorrect data. Instructions that are dependent on the post-hole instruction may have executed using this incorrect data, producing and distributing still more incorrect data. Recovering from this mispredict may be trickier. The machine must re-execute the post-hole instruction and any instructions that executed using incorrect data.

The advantage of using techniques in this paradigm is that, assuming the predictions are mostly correct, post-hole instructions are rarely forced to wait for hole instructions they are not dependent on. As a result, the distribution of post-hole instruction results is almost never needlessly delayed. The disadvantage, of course, is that the predictions are sometimes wrong. And when a prediction is wrong, a post-hole instruction may execute using source operands that contain incorrect data, and, as a result, the post-hole instruction may distribute incorrect data.

I invented two techniques that follow the speculative paradigm. I call these two techniques *assume independence* and *oracle*.

Assume Independence

The assume independence technique optimistically predicts (or assumes) that *all* post-hole instructions are independent of the hole instructions. This allows some of the post-hole instructions to be executed before the hole disappears. With this technique, post-hole instructions that are dependent on hole instructions may initially execute using source operands that contain incorrect data. However, an instruction re-execution scheme is used to ensure that these instructions will eventually be executed using the correct source operand data. The disadvantage of this technique is that some post-hole instructions may temporarily generate incorrect results, potentially causing branches to resolve incorrectly, or memory requests that would not otherwise occur. Also, instructions producing incorrect results may contend for resources needed by instructions executing correctly. The advantage of this technique is that post-hole instructions that are not dependent on hole instructions can execute during the processing of the instruction cache miss.

This technique uses an instruction re-execution scheme to ensure that a post-hole instruction that initially executed using incorrect source operand data is eventually executed using the correct source operand data. When a post-hole instruction's dependencies are resolved, any flow dependencies that were incorrectly established are corrected. This is accomplished by fixing all of the instruction's incorrect rename tags. Any instruction whose rename tags are fixed is forced to re-execute. When the data associated with the rename tags for all source operands becomes available, the instruction can be immediately sent to a functional unit and executed. When an instruction from a hole executes and distributes its result, any post-hole instructions that initially executed using incorrect values will be re-executed correctly. All instructions executed incorrectly will eventually be re-executed as the correct values propagate down the dependence graph.

This technique deals with register dependencies in a way that is analogous to the way that the blind (or naive) speculation memory disambiguation technique deals with memory dependencies. For blind speculation, when it is uncertain whether or not a load is dependent on a store in the active window, the technique *always* predicts that the load is independent. For assume independence, when it is uncertain whether or not a post-hole instruction is dependent on a hole instruction, the technique *always* predicts that the post-hole instruction is independent.

Oracle

For the oracle technique, the hardware uses an oracle to predict whether or not a post-hole instruction is dependent on a hole instruction. The oracle always provides correct predictions. The oracle is equivalent to a perfect (100 percent hit rate) mask cache. A post-hole instruction that is predicted to be independent of hole instructions can be scheduled for execution once its flow dependencies have been satisfied. A post-hole instruction that is predicted to be dependent is scheduled for execution when the hole instructions it depends on have been decoded and issued, and its flow dependencies have been satisfied. The oracle can't be built, so this technique cannot actually be implemented in hardware. I use this technique because it ideally handles the data dependency problems that result from having holes in the dataflow graph. That is, the oracle technique is the optimal dependency handling technique. This technique provides an upper bound on the performance of any machine that implements out-of-order fetch/decode/issue.

6.4 Experimental Results

This section presents some preliminary results that show that the instruction cache bottleneck for a sixteen wide issue machine can be nearly eliminated with out-of-order fetch/decode/issue. These preliminary results were generated using an RDF model that was augmented with a real instruction cache, a real branch predictor, a real execution core, and a real data cache.

All RDF machines modeled in this chapter have a window size of 1024 instructions, an issue rate of sixteen instructions per cycle, and the instruction class latencies specified in Table 4.1. They all use the simple oracle memory disambiguation technique. For conventional machines (i. e., machines that fetch, decode, and issue instructions in program order), instructions are issued and retired in the order they appear in the dynamic instruction stream. For machines that implement out-of-order fetch/decode/issue, instructions may be issued out-of-order, but they are always retired in order.

The default instruction cache is non-blocking, direct mapped, 16k bytes, with a 64 byte line size. The cache access requires one cycle. In the event of a cache miss, an additional 10 cycles are required to access the next level of cache and/or memory. In the experiments in this chapter, I will vary the size, access time, set associativity, and miss penalty of this default instruction cache. The performance benefit of out-of-order fetch/decode/issue strongly depends on the cache miss rate and the cache miss penalty. The miss rate is a function of the cache size and the benchmark being simulated. The size of my default instruction cache may be rather small for tomorrow's machines. Then again, the size of my benchmarks may be rather small for tomorrow's machines. The miss penalty is a function of the cycle time and cache/memory hierarchy. The miss penalty of my default instruction cache may be too small, too large, or just right depending on the cycle times and cache/memory hierarchies of tomorrow's machines. Note that allowing a machine to perform out-of-order fetch/decode/issue opens up new possibilities in the design of the cache/memory hierarchy. For example, without out-of-order fetch/decode/issue, a machine may need both an on-chip second level cache and an off-chip third level cache to meet its performance goals. With out-of-order fetch/decode/issue, the machine may no longer need the on-chip second level cache to meet its performance goals.

The conditional branch predictor is a gshare [81] scheme which exclusive-ORs a 16-bit global history with the fetch address to select the appropriate pattern history table entry. Indirect (or computed) branch targets are predicted using the "tagless" variety of the pattern based predictor proposed by Chang, Hao, and Patt [19]. A 9-bit global history is used to select an entry in a table of indirect branch target addresses. To improve prediction accuracy, I added a single "hysteresis" bit to each entry in the table. The bit controls the replacement of the branch target address stored in that entry. [2] Subroutine returns are predicted using a 64 entry Return Address Stack. To model the real branch predictor, as the branches are encountered in the dynamic instruction stream, a prediction is made. This prediction is compared to the real outcome of the branch. If a prediction is incorrect, issue is stalled until the branch is resolved and instructions from the correct path are available for issuing. I assume six cycles between when a branch is resolved and when instructions from the correct path are available. Since one cycle is required to execute the branch, the minimum branch mispredict penalty is seven cycles. This mispredict penalty is identical to that of the Compaq Alpha 21264 [43]. I did not investigate the problem of BTB misses or any possible solutions. I modeled a perfect (100 percent hit rate) BTB under the assumption that there will be a suitable solution.

The execution core consists of sixteen fully pipelined functional units, where each functional unit is capable of performing every desired operation. Each cycle, the oldest sixteen ready instructions are scheduled for execution. (This execution core is modeled by setting the dispatch rate to sixteen. The dispatch rate is the maximum number of instructions that can be scheduled for execution on functional units in a single cycle.) In addition, the maximum number of instructions that can be retired in a single cycle is sixteen.

The data cache is non-blocking, direct mapped, 16k bytes, with a 64 byte line size. All loads require one cycle for address calculation, and one cycle for cache access. In the event of a cache miss, loads require an additional 10 cycles for accessing the next level of cache and/or memory. (The load latency on a cache hit is 2 cycles. The load latency on a cache miss is 12 cycles.)

There are two major drawbacks of the RDF simulator, which is used to generate the results in this chapter. The first was described at the beginning of this chapter: the RDF simulator cannot distinguish between fetch and issue. As a result, only out-of-order fetch/decode/issue can be modeled with this simulator. Out-of-order fetch cannot be modeled. However, the performance of a machine that implements out-of-order fetch/decode/issue is an upper bound on the performance of a machine that implements out-of-order fetch. Hence, the results presented in this chapter for machines that implement

[2] Whenever an entry provides the correct prediction for a branch, its hysteresis bit is set to 1. Whenever it provides an incorrect prediction, its hysteresis bit is set to 0. The branch target address stored in an entry can only be replaced if the entry provides an incorrect prediction and the hysteresis bit was 0 at the time the prediction was made.

out-of-order fetch/decode/issue should be treated as the upper bounds on performance for machines that implement out-of-order fetch. The second major drawback of the RDF simulator is that the assume independence dependency handling technique cannot be modeled on it. Fortunately, the oracle dependency handling technique can be modeled on the RDF simulator, and this technique provides an upper bound on the performance of a machine that implements out-of-order fetch/decode/issue using assume independence. Experiments for machines that implement out-of-order fetch and experiments for machines that implement out-of-order fetch/decode/issue using assume independence will be presented in Chapter 8.

6.4.1 Varied Dependency Handling Technique

To demonstrate the performance potential of out-of-order fetch/decode/issue, and to determine which dependency handling techniques are worthwhile, I simulated six machines.

The first machine has the default instruction cache and does not support out-of-order fetch/decode/issue. It is a conventional machine and serves as the baseline.

The second machine implements out-of-order fetch/decode/issue using the mask cache dependency handling technique. I assumed that this machine had a fixed cache budget from which an instruction cache and mask cache could be allocated. The cache budget was set to 16k bytes—the size of the default instruction cache. This allows the performance of the second machine to be fairly compared to the performance of any machine that uses the default instruction cache and doesn't require a mask cache. Half of the cache budget was allocated to build an 8k byte instruction cache and the other half was allocated to build an 8k byte mask cache. For the mask cache technique, when there is a mask cache miss, either fetch, decode, and issue can stall, or one of the other dependency handling techniques (i. e., assume dependence, classification, or assume independence) can be used to enable out-of-order fetch/decode/issue. To obtain an upper bound on the performance of machines that implement out-of-order fetch/decode/issue by allocating half of their cache budget for a mask cache, this machine uses the oracle technique to enable out-of-order fetch/decode/issue on a mask cache miss. The oracle required for the oracle technique can't be built, so this machine cannot actually be implemented in hardware. However, the results will show that this machine does not perform as well as a machine that can be implemented: a machine that implements out-of-order fetch/decode/issue using only the assume dependence technique.

The second machine is identical to a machine that implements out-of-order fetch/decode/issue using a perfect mask cache. The oracle that is used to enable out-of-order fetch/decode/issue on a mask cache miss is equivalent to a perfect (100 percent hit rate) mask cache. On a mask cache hit, the mask cache provides the dependency information required for out-of-order fetch/decode/issue. On a mask cache miss, the oracle provides the dependency information that would have come from the mask cache had there been a mask cache hit. Thus, regardless of whether or not there is a hit in the mask cache, the machine obtains the dependency information required for out-of-order fetch/decode/issue. In the figures and in the remaining text, the second machine will be referred to as the machine that implements out-of-order fetch/decode/issue using a perfect mask cache.

The third and fourth machines implement out-of-order fetch/decode/issue using realistic (i. e., implementable) dependency handling techniques. The third machine uses the assume dependence technique and the fourth machine uses the classification technique. Both of these machines use the default instruction cache.

The fifth machine also uses the default instruction cache. It implements out-of-order fetch/decode/issue using the oracle technique. The oracle required for this technique can't be built, so this machine cannot actually be implemented in hardware. The primary reason for simulating this machine is that it provides the upper bound on the performance of a machine that implements out-of-order fetch/decode/issue using the assume independence technique. (The assume independence technique cannot be modeled on the RDF simulator.) This machine also provides the upper bound on the performance of all machines that implement out-of-order fetch/decode/issue.

The last machine has a perfect instruction cache. Hence, it cannot be implemented. This machine provides an upper bound on the performance of any technique (code reordering, prefetching, ...) that tries to deal with the problem of instruction cache misses.

Figure 6.5 shows the performance, in Instructions Per Cycle (IPC), averaged over all the benchmarks (both SPEC and Non-SPEC) for each of the six machines. Results for the individual benchmarks are provided at the end of this section in Figure 6.6 (SPEC benchmarks) and Figure 6.7 (Non-SPEC benchmarks). In the figures, OOO FDI stands for Out-Of-Order Fetch/Decode/Issue. The machines with out-of-order fetch/decode/issue achieve average speedups of 48% (for perfect mask cache), 49% (for assume dependence and classification), and 52% (for oracle) over the baseline machine. The performance of these machines also comes within 9%–11% of 4.88 IPC, which is the performance of the machine with the perfect instruction cache.

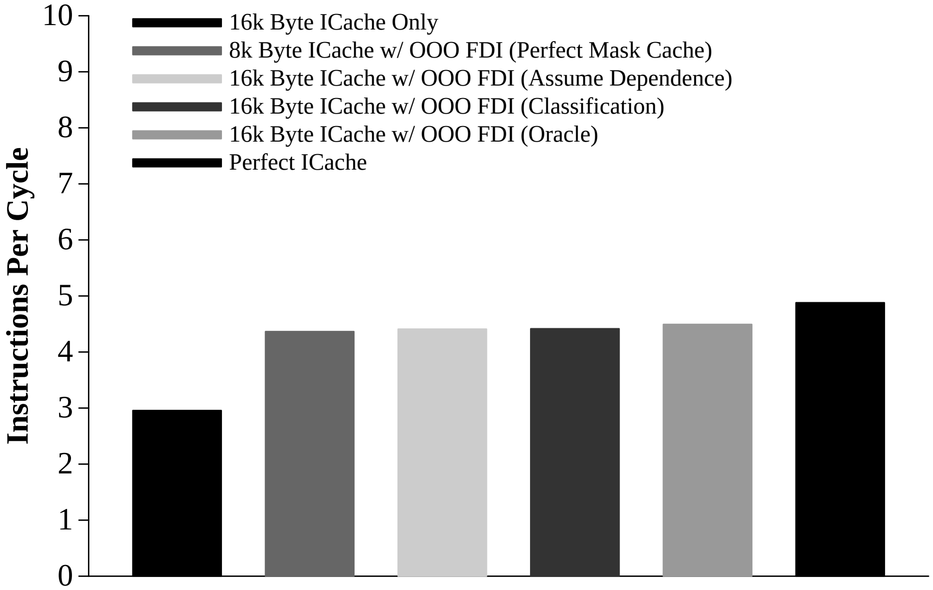


Figure 6.5: Dependency Handling Techniques—Harmonic Average

The performances of the four machines with out-of-order fetch/decode/issue are nearly identical. The machine that uses the perfect mask cache has the poorest performance. This machine sacrifices half of its cache budget to build the perfect mask cache, and, as a result, its instruction cache can only be half as big as the instruction caches of the other three machines. Consequently, it suffers from more instruction cache misses than the other machines. These extra instruction cache misses put machines with mask caches at a disadvantage when compared to machines that don't require mask caches. For this reason, in the remaining experiments in this dissertation, I won't simulate any machines that implement out-of-order fetch/decode/issue using the mask cache dependency handling technique.

The machine that uses the classification dependency handling technique does not perform noticeably better than the machine that uses the assume dependence technique. The advantage of the classification technique over the assume dependence technique is that the post-hole instructions that are classified as independent are allowed to execute before the hole disappears. However, very few post-hole instructions are actually classified as being independent. On average (over all the benchmarks), only 10% of all post-hole instructions are classified as independent. As a result, the classification technique provides little benefit over the assume dependence technique.

The implementation of the classification technique is more difficult than the implementation of assume dependence. For the post-hole instructions that are classified as being possibly dependent, the classification technique requires the same hardware as that used to implement the assume dependence technique. In addition to this hardware, the classification technique also needs hardware that classifies the post-hole instructions, and hardware that prevents the possibly dependent post-hole instructions from being scheduled for execution before the hole disappears, but allows the independent post-hole instructions to be scheduled for execution. Since the classification technique provides no noticeable performance advantage over the assume dependence technique, in the remaining experiments in this dissertation, I won't simulate any machines that implement out-of-order fetch/decode/issue using the classification technique.

The machine that uses the oracle dependency handling technique did not perform significantly better than the machine that uses the assume dependence technique. The performance, averaged over all the benchmarks, of the machine that uses the oracle technique was only 2% better than that of the machine that uses assume dependence. The assume dependence technique allows the fetch unit to perform prefetching during an instruction cache miss, and allows early insertion of the post-hole instructions into the reservation stations, but it does not allow the processor to take advantage of any parallelism exposed by out-of-order fetch/decode/issue. The oracle technique, on the other hand, allows the processor to take advantage of this parallelism. Out-of-order fetch/decode/issue does expose additional parallelism. When the oracle technique is used, a significant number of post-hole instructions become schedulable before the hole disappears. On average, 35% of the instructions issued while there are holes in the dataflow graph become schedulable before those holes disappear. Taking advantage of this additional parallelism, however, does not significantly improve the performance. One possible reason for this may be that the post-hole instructions that become schedulable before the hole disappears rarely belong to the program's critical dependency chains.

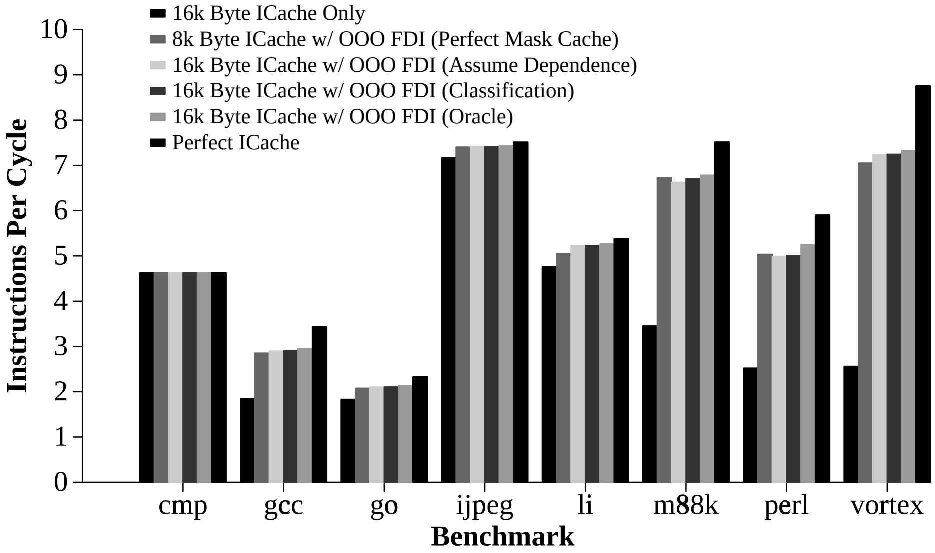


Figure 6.6: Dependency Handling Techniques—SPEC Benchmarks

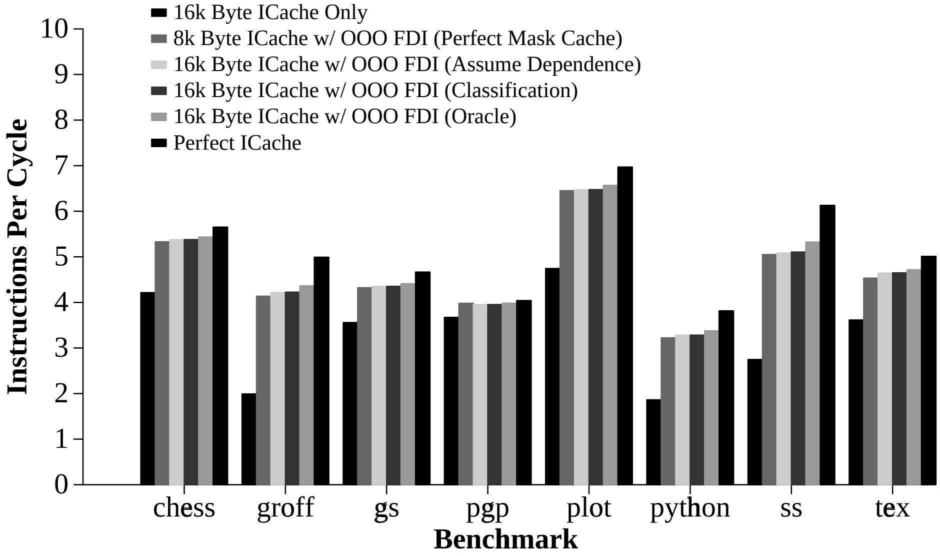


Figure 6.7: Dependency Handling Techniques—Non-SPEC Benchmarks

6.4.2 Impact of Procedure Reordering

Code re-ordering techniques can be used in combination with out-of-order fetch/decode/issue in order to further reduce the performance penalty that results from instruction cache misses. Code re-ordering is used to eliminate some of the instruction cache misses by re-ordering the instructions in a program. Out-of-order fetch/decode/issue is used to tolerate the remaining misses.

Figure 6.8 shows the performance averaged over all the benchmarks for three machines: the conventional (baseline) machine, the machine that implements out-of-order fetch/decode/issue using the assume dependence technique, and the machine with a perfect instruction cache. There are two bars for each machine, except for the machine with the perfect instruction cache. One bar plots the performance of the machine running the versions of the benchmark executables that were created without profiling. This bar is labeled “Not Reordered” since the procedures in the executables were not reordered to allow the programs to use the instruction cache more efficiently. The other bar plots the performance of the machine running the versions of the benchmark executables that were created with profiling. It is labeled “Reordered”. The performance of the machine with the perfect instruction cache does not depend on the version of the benchmark executables the machine is running. Consequently, there is only one bar for this machine.

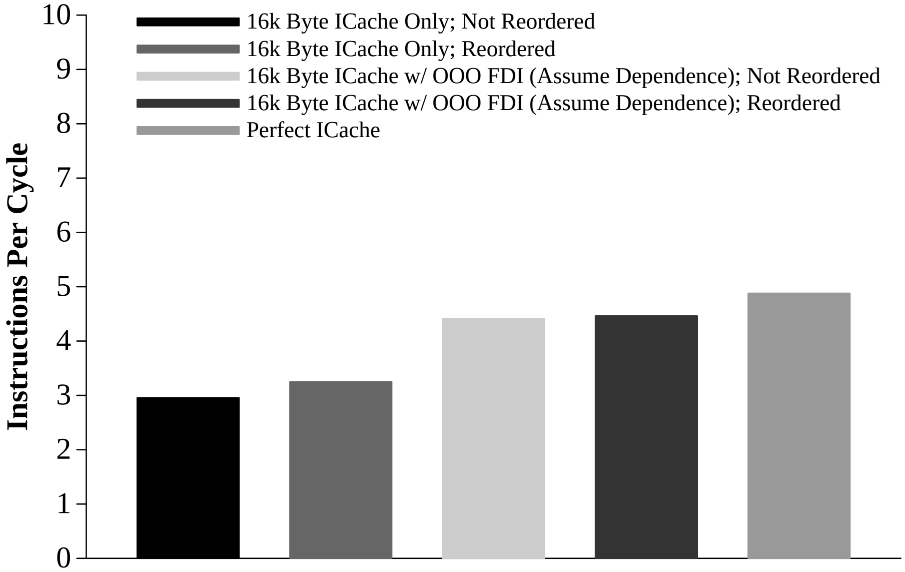


Figure 6.8: Impact of Out-of-Order Fetch/Decode/Issue and Procedure Reordering—Harmonic Average

Procedure reordering improves the performance of the baseline machine by 10%, and the performance of the out-of-order fetch/decode/issue machine by 1%. Out-of-order fetch/decode/issue improves the performance of a machine running the benchmark executables created without profiling by 49%, and the performance of a machine running the benchmark executables created with profiling by 37%. If both procedure reordering and out-of-order fetch/decode/issue are used, the performance is improved by 51%. The results for the individual benchmarks are provided in Figure A.37 (SPEC benchmarks) and Figure A.38 (Non-SPEC benchmarks) of Appendix A.

6.4.3 Varied Instruction Cache Size

For the rest of the experiments in this chapter, I will only consider the performance of four different types of machines. The first type are machines that have perfect instruction caches. The second type are machines that implement out-of-order fetch/decode/issue using the oracle dependency handling technique. The third type are machines that implement out-of-order fetch/decode/issue using the assume dependence technique. The fourth type are conventional machines that have a real instruction cache and that do not support out-of-order fetch/decode/issue.

Figure 6.9 shows the performance averaged over all the benchmarks for each of the four types of machines. The instruction cache size for each type of machine was varied from 4k bytes to 64k bytes. The instruction cache access required one cycle regardless of its size. That is, its access time was not scaled with its size. Because of this, the depth of the front-end of the processor pipeline (i. e., the number of pipeline stages required for instruction fetch, decode, and issue) did not depend on the instruction cache size, and, as a result, the minimum branch mispredict penalty was a constant 7 cycles for all machines. At all instruction cache sizes, the performance of the machine that uses the oracle technique is virtually identical to the performance of the machine that uses assume dependence. As the instruction cache size increases, fewer instruction cache misses occur and thus there are fewer opportunities for out-of-order fetch/decode/issue. Because of this, the performance benefit of out-of-order fetch/decode/issue decreases as the instruction cache size increases. At an instruction cache size of 4k bytes, the machine that implements out-of-order fetch/decode/issue using the assume dependence technique achieves a 109% gain in performance over the baseline machine. This gain falls to 15% when the instruction cache size is increased to 64k bytes. (I would like to note that most of my benchmarks fit within the 64k byte instruction cache. Many real applications do not fit within a 64k byte instruction cache. For these applications, out-of-order fetch/decode/issue could still provide a substantial benefit when the instruction cache size is 64k bytes.) The performance of the baseline machine drops by 51% as the instruction cache size is reduced from 64k bytes to 4k bytes. The performance of the machine that implements out-of-order fetch/decode/issue using the assume dependence technique drops by only 11% as the instruction cache size is reduced from 64k bytes to 4k bytes. Results for individual benchmarks are provided in Figure A.39 (SPEC) and Figure A.40 (Non-SPEC) of Appendix A.

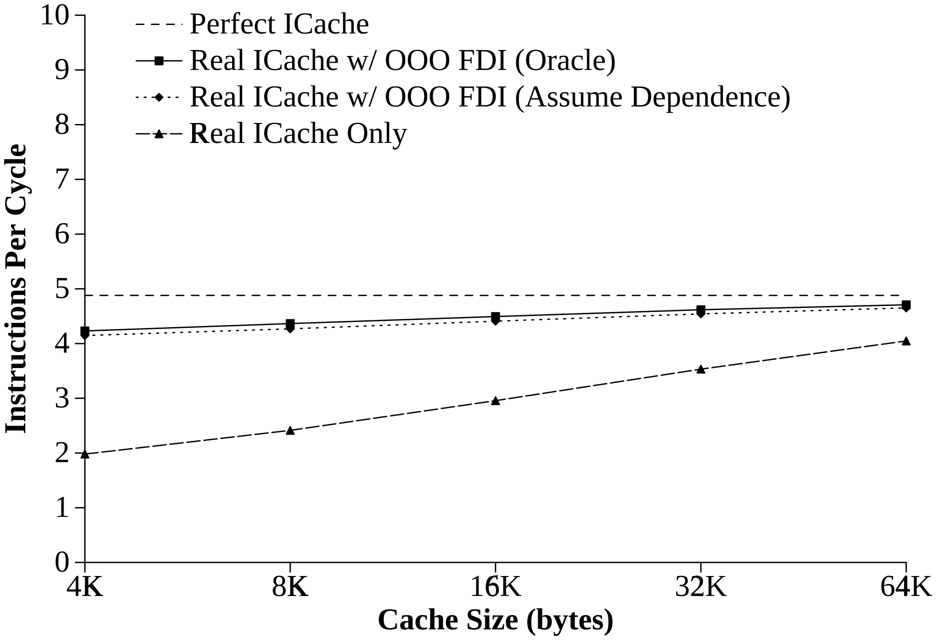


Figure 6.9: Varied Instruction Cache Size (Constant Mispredict Penalty)—Harmonic Average

Figure 6.10 shows the performance averaged over all the benchmarks for each of the four types of machines as the size of the instruction cache was varied from 16k bytes to 1M byte. The access time of the instruction cache was scaled with its size. The access time was 1 cycle at 16k bytes, 2 cycles at 64k bytes, 4 cycles at 256k bytes, and 8 cycles at 1M byte. Because the access time was scaled, the depth of the front-end of the processor pipeline depends on the instruction cache size, and, as a result, the minimum branch mispredict penalty must be scaled according to the instruction cache size. The minimum mispredict penalty was 7 cycles at a cache size of 16k bytes, 8 cycles at 64k bytes, 10 cycles at 256k bytes, and 14 cycles at 1M byte. The machine with the perfect instruction cache is supposed to represent the ideal solution to the instruction cache bottleneck. The ideal solution is an instruction cache that has a 100 percent hit rate and a single cycle access time. (A single cycle access time is ideal because it results in the smallest possible minimum mispredict penalty.) For this reason, the machine with the perfect instruction cache could alway access its cache in a single cycle, and its minimum mispredict penalty was always 7 cycles.

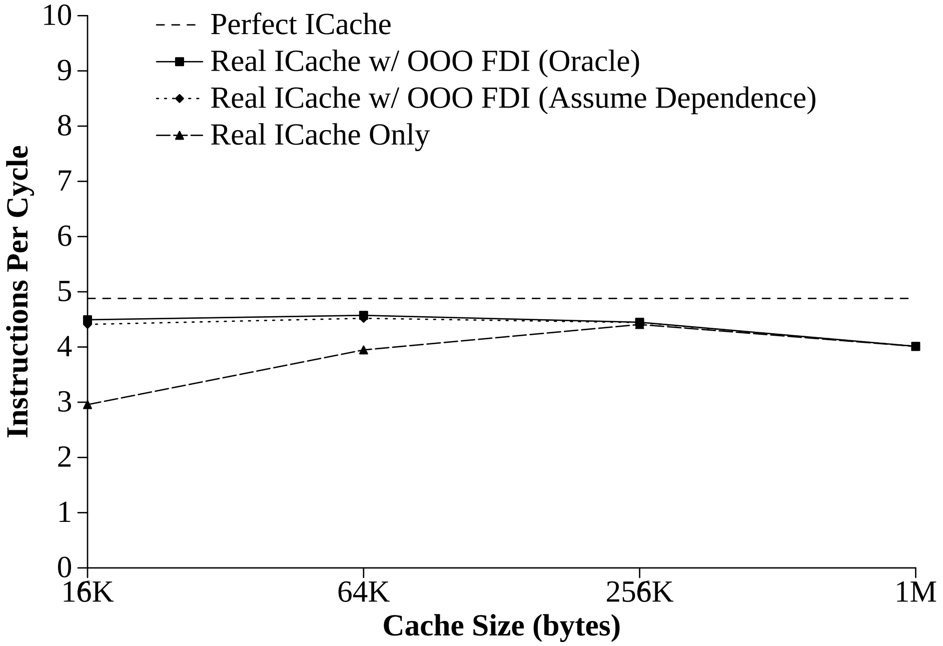


Figure 6.10: Varied Instruction Cache Size (Scaled Mispredict Penalty)—Harmonic Average

As the instruction cache size increases, fewer instruction cache misses occur, which reduces the severity of the instruction cache bottleneck. The minimum branch mispredict penalty also increases as the instruction cache size increases, which increases the severity of the bottleneck that results from mispredicted branches. Beyond a certain point, increasing the size of the instruction cache will result in lower performance, because it will increase the severity of the bottleneck that results from mispredicted branches more than it reduces the severity of the instruction cache bottleneck. For the baseline machine, increasing the size of the instruction cache beyond 256k bytes results in lower performance. For the machines that implement out-of-order fetch/decode/issue, the instruction cache bottleneck is much less severe, so the point at which the performance is harmed if the cache size is increased occurs at a smaller instruction cache size. For these machines, increasing the size of the instruction cache beyond 64k bytes results in lower performance. Results for individual benchmarks are provided in Figure A.41 (SPEC) and Figure A.42 (Non-SPEC) of Appendix A.

6.4.4 Varied Instruction Cache Associativity

Figure 6.11 shows the performance averaged over all the benchmarks for each of the four machines. The instruction cache associativity for each machine was varied from 1 (i. e., direct-mapped) to 8. Even for 8-way set associative caches, machines with out-of-order fetch/decode/issue achieved speedups of 21% (for assume dependence) and 23% (for oracle) over the baseline machine. Results for the individual benchmarks are provided in Figure A.43 (SPEC benchmarks) and Figure A.44 (Non-SPEC benchmarks) of Appendix A.

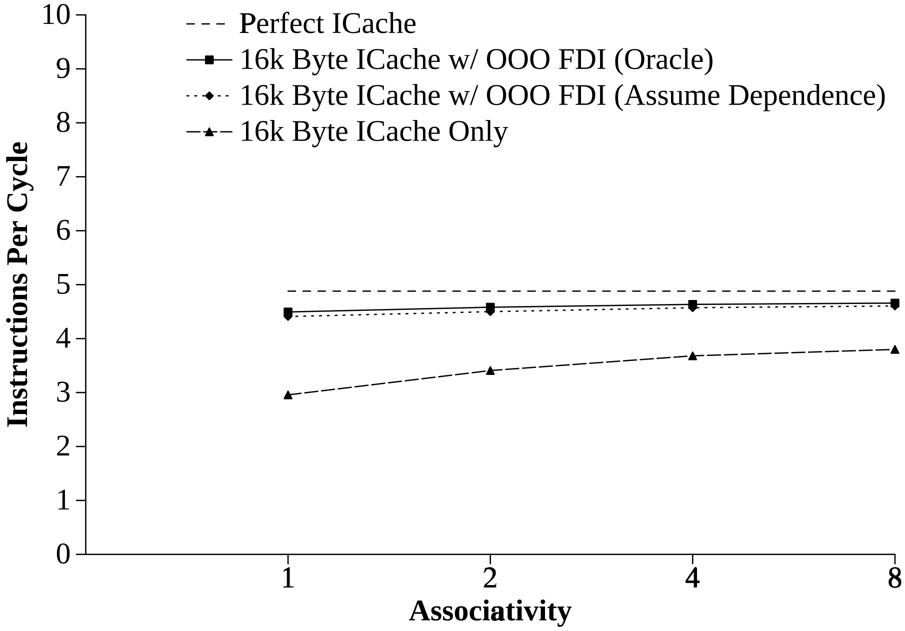


Figure 6.11: Varied Instruction Cache Associativity—Harmonic Average

For highly associative instruction caches, compulsory and capacity misses account for the lion's share of the misses. Conflict misses are eliminated by increasing the associativity of the cache. If the associativity is high enough, most of the conflict misses are eliminated, and, of the remaining misses, most are either compulsory or capacity misses. Compulsory misses occur when an instruction is referenced for the first time. For these misses, the only benefit that out-of-order fetch/decode/issue provides is that of prefetching. Capacity misses can occur during the processing of a seldom used subroutine. Out-of-order fetch/decode/issue hides the latency associated with these misses by fetching, decoding, issuing, and, for some dependency handling techniques, executing, the instructions following the subroutine's return while the misses are being serviced.

6.4.5 Varied Instruction Cache Miss Penalty

The performance benefit of out-of-order fetch/decode/issue depends on the penalty incurred on instruction cache misses. For each of the four machines, Figure 6.12 shows the performance averaged over all the benchmarks. For each machine, the instruction cache miss penalty was varied from 6 to 32 cycles. For the machine that implements out-of-order fetch/decode/issue using the oracle dependency handling technique, the performance gain over the baseline machine ranges from 29% for an instruction cache with a 6 cycle miss penalty, to 154% for an instruction cache with a 32 cycle miss penalty. For the machine that implements out-of-order fetch/decode/issue using assume dependence, the gain ranges from 28% (6 cycle miss penalty) to 142% (32 cycle miss penalty). In fact, the performances of both machines that implement out-of-order fetch/decode/issue and that have 32 cycle miss penalties exceed the performance of the baseline machine that has a 6 cycle miss penalty. Results for the individual benchmarks are provided in Figure A.45 (SPEC benchmarks) and Figure A.46 (Non-SPEC benchmarks) of Appendix A.

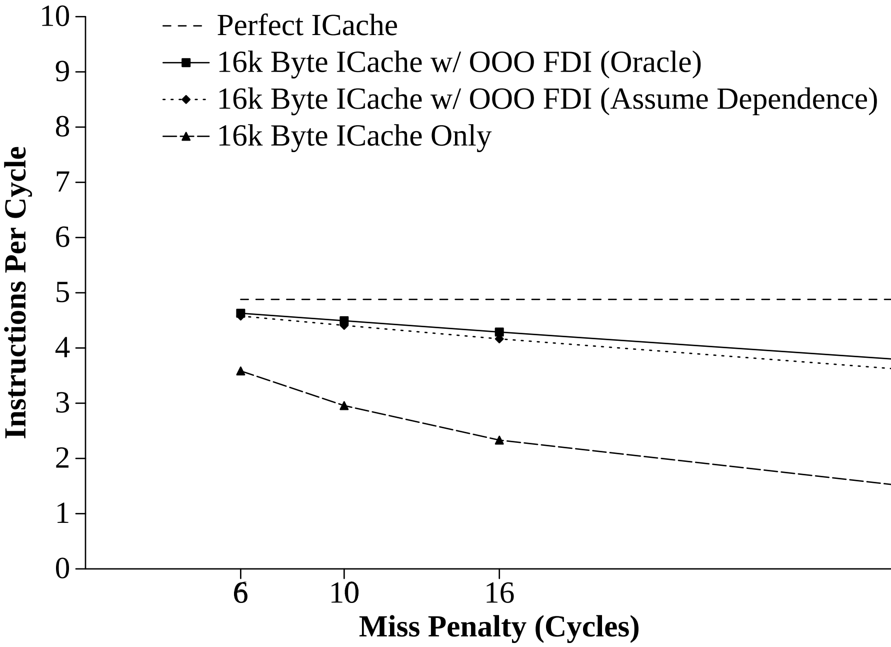


Figure 6.12: Varied Instruction Cache Miss Penalty—Harmonic Average

6.5 Summary

Of the four realistic dependency handling techniques—assume dependence, classification, mask cache, and assume independence—only two, assume dependence and assume independence, may be worthwhile. The classification technique does not perform noticeably better than the assume dependence technique, and its implementation is more difficult than the implementation of assume dependence. The mask cache technique always performs worse than the assume dependence technique, even if a perfect (100 percent hit rate) mask cache is implemented. For the remainder of this dissertation, the only realistic dependency handling techniques that I will investigate are the assume dependence and assume independence techniques.

Instruction cache misses significantly impact processor performance. When a perfect instruction cache is used, the average performance of the default sixteen wide issue processor is 4.88 Instructions Per Cycle (IPC). However, when a real instruction cache is used, the performance drops by 39% to 2.96 IPC.

Out-of-order fetch/decode/issue can be used to reduce this performance degradation. A processor that uses a real instruction cache and that implements out-of-order fetch/decode/issue using assume dependence has an average performance of 4.41 IPC, which is only a 9.6% drop in performance from a processor using a perfect instruction cache. To put it another way, its performance is 49% higher than a machine with a real instruction cache that doesn't implement out-of-order fetch/decode/issue. The performance of this processor is close to that of a processor that implements out-of-order fetch/decode/issue using an ideal dependency handling technique (i. e., oracle). The performance of this latter processor is 4.49 IPC, which is only an 7.9% drop in performance from a processor using a perfect instruction cache.

CHAPTER 7

Out-of-Order Fetch, Decode, and Issue: Implementation

This chapter describes one possible implementation of out-of-order fetch and one possible implementation of out-of-order fetch/decode/issue. The chapter is organized into three sections. Section 7.1 presents the base microarchitecture that will be used to demonstrate the implementations of out-of-order fetch and out-of-order fetch/decode/issue. This is by no means the only microarchitecture upon which out-of-order fetch and out-of-order fetch/decode/issue can be implemented. Section 7.2 presents the apparatus required for implementing out-of-order fetch. Out-of-order fetch/decode/issue requires out-of-order fetch. Ergo, the apparatus required for out-of-order fetch is also required for out-of-order fetch/decode/issue. Section 7.3 presents the remaining apparatus required for implementing out-of-order fetch/decode/issue.

7.1 Base Microarchitecture

The base microarchitecture is an HPS microarchitecture. (The HPS microarchitecture was described in Section 3.1.) Figure 7.1 shows a block diagram of the base microarchitecture. The base microarchitecture is a more aggressive version of the microarchitecture presented by Butler [15]: it can fetch up to 3 fetch blocks per cycle, whereas Butler's microarchitecture can fetch at most 1 fetch block per cycle. The base microarchitecture, which is modeled with the full simulator, is more realistic than the simple microarchitecture used for the experiments in the previous chapters. However, it is still only a model, and not the real thing. Not everything that would be a part of a real microarchitecture is modeled, and not everything that is modeled is modeled with complete accuracy.

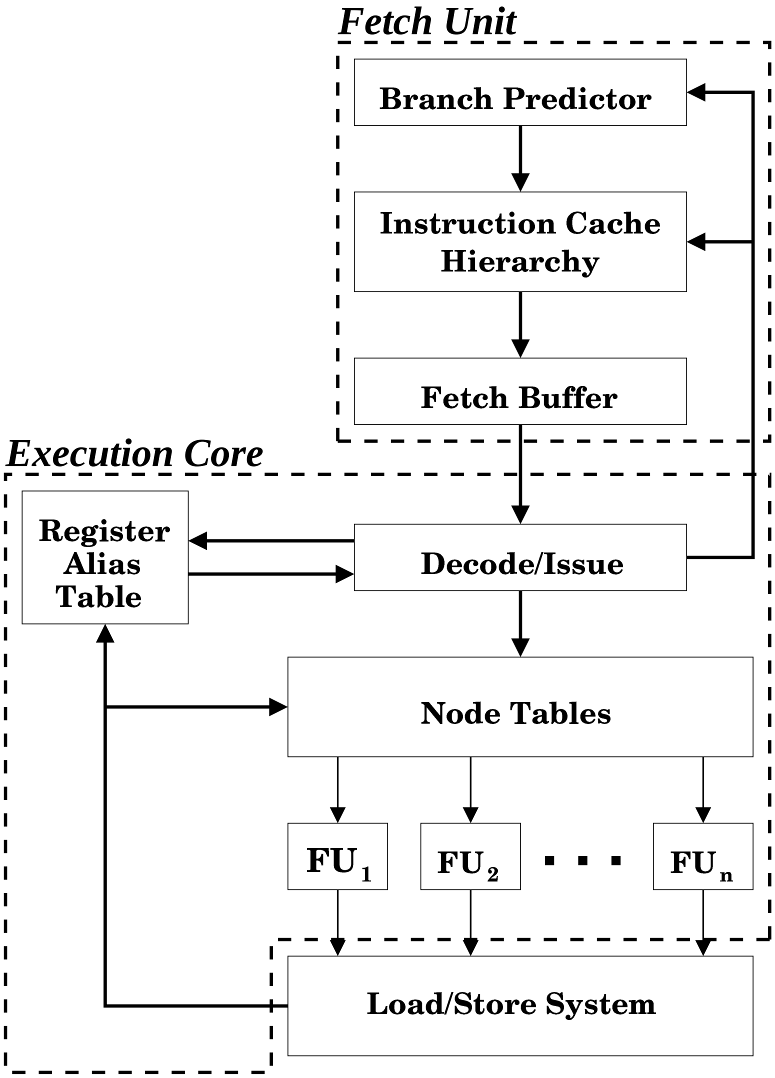


Figure 7.1: Block Diagram of the Base Microarchitecture

The pipeline of the base microarchitecture consists of 5 stages: fetch, decode, issue, execute, and retire. Each stage takes at least one cycle. The paragraph below summarizes each of these stages. More details will be presented in Sections 7.1.1–7.1.4.

In the fetch stage, the fetch unit accesses the instruction cache with a set of fetch addresses that are generated by the branch predictor. The instructions returned by the instruction cache are then written into the fetch buffer. Every cycle, the branch predictor generates the set of fetch addresses that will be needed to access the instruction cache in the next cycle. It generates these addresses using the set of fetch addresses that are being used in the current cycle to access the instruction cache. In the decode stage, instructions are removed from the fetch buffer and decoded into data flow nodes suitable for issue. In the issue stage, the decoded instructions are written into the Node Tables. If the source operands for a particular instruction are available, and if the instruction has priority over all other firable instructions awaiting dispatch to the needed functional unit, the instruction is immediately dispatched. Otherwise, the instruction must wait in its Node Table until all of its source operands become available and it becomes the highest priority firable instruction. Once an instruction has been dispatched, it enters the execute stage. In the execute stage, the functional unit executes the instruction and then distributes the instruction's result to the Register Alias Table and to the other instructions in the Node Tables awaiting that result. And finally, in the retire stage, instructions are retired from the active window. Instructions are always retired in program order. As the instructions are retired, their results are committed to the architectural (non-speculative) state, and any processor resources (e. g., physical registers and Node Table entries) that are still allocated to them are deallocated.

The base microarchitecture is a 16 wide machine. It was designed to fetch, on average, 16 instructions per cycle. It can decode, issue, dispatch, execute, and retire a maximum of 16 instructions per cycle. In principle, the microarchitecture can fetch up to 48 instructions per cycle. However, due to the small size of the average fetch block (4–6 instructions), and a fetch limit of 3 blocks per cycle, this peak fetch rate is rarely achieved.

The base microarchitecture is described in more detail in Sections 7.1.1–7.1.4. Section 7.1.1 describes the fetch unit (see Figure 7.1). Section 7.1.2 describes the execution core. Section 7.1.3 describes the load/store system. And Section 7.1.4 describes the main memory architecture.

7.1.1 Fetch Unit

Figure 7.2 is a block diagram of the fetch unit. The branch predictor predicts the sequence of fetch blocks that comprises the dynamic instruction stream. For each fetch block in the predicted sequence, the predictor produces one or more cache read requests in order to obtain the instructions that belong to that fetch block. Each request reads a single cache line from the first level instruction cache. The request also specifies which instructions from that cache line are contained in the fetch block. Fetch blocks that are contained entirely within a single cache line only generate a single request. Fetch blocks that span multiple cache lines generate one request for each cache line that contains a piece of the fetch block. Any given request only returns instructions belonging to a particular fetch block. That is, a request never returns instructions belonging to two (or more) fetch blocks. If there are two fetch blocks in a particular cache line, and they form a piece of the sequence of fetch blocks that comprises the dynamic instruction stream, the branch predictor will generate two requests for these fetch blocks: one for the first fetch block in the cache line, and one for the second fetch block in the cache line. Essentially, the branch predictor predicts the sequence of first level instruction cache lines, that contain the sequence of fetch blocks, that comprises the dynamic instruction stream.

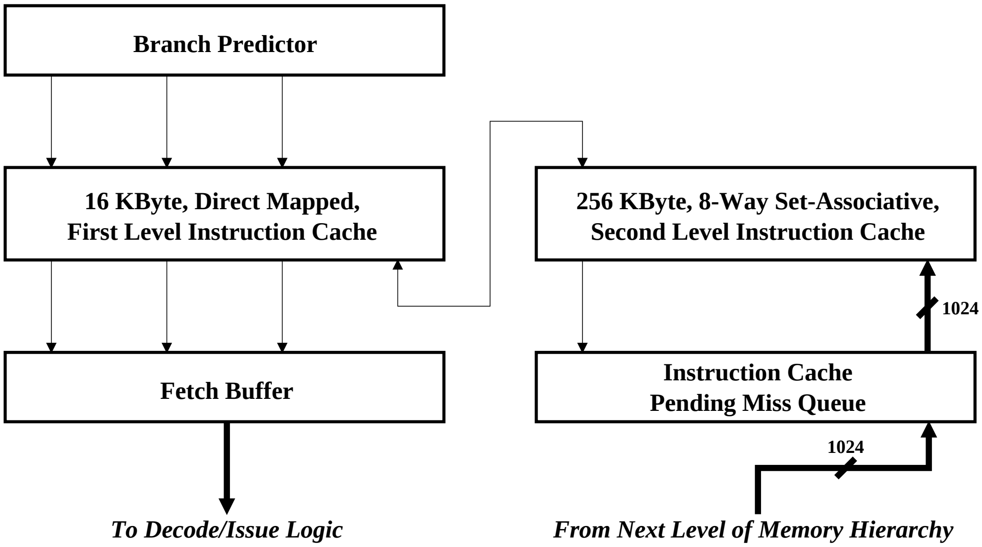

Branch Predictor
16 KByte, Direct Mapped, First Level Instruction Cache
256 KByte, 8-Way Set-Associative, Second Level Instruction Cache
1024
Fetch Buffer
Instruction Cache Pending Miss Queue
1024
To Decode/Issue Logic
From Next Level of Memory Hierarchy

Figure 7.2: Block Diagram of the Fetch Unit

The branch predictor predicts only a piece of the sequence of first level instruction cache lines each cycle. The sequence is predicted in program order. That is, the sequence of fetch blocks that comprises the dynamic instruction stream is predicted in program order, and, for each fetch block, read requests for first level instruction cache lines that contain the pieces of that fetch block are produced in program order. State variables are used to keep track of the current point in the sequence. Each time a piece of the sequence is predicted, these state variables are updated. In certain situations—for example, when there is a branch mispredict [60, 62, 113] or an instruction cache miss—the branch predictor is forced to restart at an earlier point in the sequence. To restart the branch predictor, the state variables are simply set to the values they had at that earlier point.

Each cycle, the branch predictor can predict up to 3 cache lines in the sequence of first level instruction cache lines. For each of these cache lines, a read request is issued to the first level instruction cache. On a cache hit, the instructions in the cache line that belong to the fetch block are written into the fetch buffer. On a cache miss, the cache line is fetched from the lower levels of the memory hierarchy. During this fetch, the branch predictor and the first level instruction cache stall. In addition, all read requests for cache lines logically following the cache line that missed are flushed from the machine. After the fetch completes, the cache line is installed in the first level instruction cache and the instructions in the cache line that belong to the fetch block are written into the fetch buffer. Also, since the read requests for cache lines logically following the cache line that missed were flushed, the branch predictor is forced to re-predict those line; i. e., the branch predictor is forced to restart at an earlier point in the sequence. After all of this has been accomplished, both the branch predictor and the first level instruction cache are restarted.

For the base microarchitecture and those derived from it, the first level instruction cache is a non-blocking, direct mapped, 16k byte cache, with a 64 byte line size and a single cycle latency. The cache is triple ported and can service up to 3 requests per cycle. A request may either by a cache read or a cache fill. At most 1 cache fill can be serviced per cycle. (A cache fill—which occurs in response to a cache miss—writes a new line, including both tag and data, into the cache.) The cache is modeled as being truly triple ported. Thus, unlike multi-banked (or interleaved) caches, there are no bank conflicts.

The fetch buffer has 8 entries. The branch predictor predicts, in program order, the sequence of first level instruction cache lines that comprises the dynamic instruction stream. The read requests for these cache lines are also generated in program order. As each read request is generated, it is allocated an entry in the fetch buffer. When a read completes, the data is written into the entry that was allocated to the read request. Consequently, each entry must be large enough to store a single first level instruction cache line.

For the base microarchitecture, the entries of the fetch buffer form a FIFO. Reads complete in the order that their requests were generated; i. e., in program order. As they complete, they write their data (or insert their data) into the fetch buffer entries that were allocated to their requests. The decode/issue logic removes the instructions stored in the fetch buffer in program order. When all the instructions stored in a fetch buffer entry have been removed by the decode/issue logic, the fetch buffer entry is deallocated. Hence, the data obtained via read requests is inserted into and removed from the fetch buffer in the order that those read requests were generated. For the microarchitecture that supports out-of-order fetch, and for the microarchitecture that supports out-of-order fetch/decode/issue, the fetch buffer does not function as a FIFO. Details on how the fetch buffer works for these microarchitectures will be provided later in this chapter.

Although each fetch buffer entry is large enough to store a single first level instruction cache line, this doesn't need to be the case. Each fetch buffer entry was made large enough to store a single first level instruction cache line in order to simplify the allocation of fetch buffer storage to the read requests generated via the branch predictor. A single read request can request all the instructions in a first level instruction cache line; i. e., 16 instructions. However, the typical read request is only for 4–6 instructions. Hence, a fetch buffer entry that can store an entire first level instruction cache line is overkill.

If the amount of chip area occupied by the fetch buffer is a concern, the number of instructions that can be stored in a fetch buffer entry can be reduced. This will reduce the amount of chip area occupied by each fetch buffer entry, and reduce the amount of storage wasted by each fetch buffer entry. (Storage is wasted because a fetch buffer entry can store up to 16 instructions, but typically stores only 4–6 instructions.) To partially compensate for the resulting loss in fetch buffer storage, the number of fetch buffer entries may be increased. Note that even though the overall amount of fetch buffer storage is reduced, the remaining storage is better utilized, so the amount of effective (or useful) fetch buffer storage may remain constant, or, in the case where the number of fetch buffer entries is increased, the effective fetch buffer storage may actually increase. Also note that a read request will need to be allocated multiple fetch buffer entries if the number of requested instructions is greater than the number of instructions that can be stored in a single fetch buffer entry. This will complicate the fetch buffer design.

The second level instruction cache is a fully pipelined, non-blocking, 8-way set-associative, 256k byte cache, with a 128 byte line size and an 8 cycle latency. Its single port is used to service the cache read requests that originate from the first level instruction cache and the cache fill requests that originate from the Instruction Cache Pending Miss Queue. At most one request, either a cache read or a cache fill, can be serviced per cycle.

Cache read requests that miss in the second level instruction cache are placed in the Instruction Cache Pending Miss Queue (Instruction Cache PMQ). The PMQ can store up to 16 miss requests. For each miss request, the PMQ obtains the sought after data from the lower levels of the memory hierarchy and inserts the data into the fetch buffer. The PMQ also coordinates the cache fills that are brought about by these cache misses.

In addition to being non-blocking, the first and second level instruction caches have non-stalling pipelines. That is, once a request has been passed to the first or second level instruction cache, the request advances through the cache's pipeline at a rate of one pipe stage per cycle. To prevent the pipelines from stalling, the read requests generated via the branch predictor are not issued to the first level instruction cache unless (1) they are guaranteed a fetch buffer entry in which to write their data, and (2) they are guaranteed an entry in the Instruction Cache PMQ. Before each read request is issued to the first level instruction cache, it is pre-allocated a fetch buffer entry and a PMQ entry. All read requests require a fetch buffer entry, but not all read requests require a PMQ entry. Only read requests that miss in both the first and second level instruction caches require a PMQ entry. As soon as a read request experiences a hit in one of the caches, its PMQ entry is deallocated.

The first level instruction cache can accept up to 3 read requests per cycle. Read requests that miss in the first level instruction cache are passed on to the second level instruction cache. Thus, up to 3 read requests per cycle can be passed on to the second level instruction cache. Unfortunately, the second level instruction cache can only accept at most one read request per cycle. To solve this problem, a small buffer can be placed between the first and second level instruction caches. Any read requests that cannot be immediately passed to the second level instruction cache are queued in the buffer. To prevent buffer overflow, and to prevent the pipelines of the first level instruction cache from stalling, the read requests generated via the branch predictor are not issued to the first level instruction cache when the number of unused (or free) buffer entries runs low.

Branch Predictor

The base microarchitecture can predict up to 3 branches per cycle. The technique used to predict multiple branches per cycle is somewhat ideal: a conventional branch predictor, which, in a real microarchitecture, would be accessed once per cycle and would produce one prediction per access, is accessed multiple times per cycle—with one access per branch—in order to provide a prediction for each branch. Realistic techniques for predicting multiple branches per cycle have been proposed [31, 54, 87, 111, 133]. One promising technique is to dynamically convert, or *promote*, each multi-way branch (i. e., conditional or indirect [or computed] branch) that is strongly biased towards a particular target into a one-way branch (i. e., unconditional branch) to the dominant target [96]. A promoted branch is, in essence, statically predicted, and doesn't require a prediction from the branch predictor. According to Patel [97], about 60% of all dynamic branches can be promoted. Hence, the bandwidth, in branch predictions per cycle, that is required of the branch predictor for a particular machine implementation can potentially be reduced by 60% by using branch promotion.

The branch predictor used by the base microarchitecture is the default branch predictor that was used in Chapter 5 to study performance bottlenecks. It was also used in Chapter 6 to generate the preliminary results for out-of-order fetch/decode/issue. It uses a gshare [81] predictor to predict the directions (i. e., taken or not taken) of conditional branches; the pattern based predictor proposed by Chang, Hao, and Patt [19] to predict the targets of indirect (or computed) branches; and our Checkpointed Return Address Stack (RAS) [60, 62] to predict the targets of subroutine returns. See Section 5.1 for a more detailed description of this predictor, and Table 5.1 and the accompanying text for statistics (e. g., miss rate). (In Table 5.1 and the accompanying text, predictor number 2 is the predictor used by the base microarchitecture.)

Unlike the branch predictors used in Chapters 5 and 6, the branch predictor used by the base microarchitecture uses a real (rather than perfect) Branch Target Buffer (BTB). The BTB is a cache that is used by the branch predictor to identify branches prior to decode. Each entry in this cache contains the information pertaining to a single branch. Note that if branches cannot be identified prior to decode, the branch predictor cannot respond to changes in the control flow until after the branch that caused the control flow redirection is decoded. As a result, pipeline bubbles are introduced at every control flow change.

For processors that only fetch a single instruction from the instruction cache each cycle, a branch is identified (i. e., tagged) by its instruction address. During fetch, the BTB is accessed with the same address that is being used to access the instruction cache. In parallel with this BTB access, the branch predictor computes 3 predictions: the direction of a conditional branch, the target of an indirect branch, and the target of a subroutine return. A hit in the BTB indicates that the instruction that was fetched from the instruction cache is a branch. On a BTB hit, the branch predictor uses the information returned by the BTB to calculate the target address of this branch. This target address is used for the next fetch address; i. e., the address that will be used next to access the instruction cache. For example, if the BTB information indicates that the fetched branch is a subroutine return, the return target—which was computed by the branch predictor during the BTB access—is selected as the next fetch address. A miss in the BTB, in most cases, indicates that the instruction that was fetched from the instruction cache is not a branch. On a BTB miss, the branch predictor always assumes that the fetched instruction is not a branch. The address of the instruction that follows the fetched instruction is used for the next fetch address.

On a BTB hit, the BTB returns the type of the branch (unconditional branch, conditional branch, branch to subroutine, jump [computed branch], jump to subroutine, or subroutine return) and one of its possible target addresses. The branch predictor uses the branch type to select the next fetch address and to control the RAS. For an unconditional branch or a branch to subroutine, there is only one possible target address. This address, which is specified via the PC-relative addressing mode, is stored in the BTB. Ergo, for unconditional branches and branches to subroutines, the target address returned by the BTB is selected as the next fetch address. For a conditional branch, there are two possible target addresses: a taken address and a not-taken (fall-through) address. The taken address, which is specified via the PC-relative addressing mode, is stored in the BTB. The not-taken address is computed from the address used to access the BTB; i. e., it is computed from the address of the branch instruction. The conditional branch predictor provides the prediction for the direction (i. e., either taken or not-taken) of the conditional branch. Based on this predicted direction, either the taken address or the not-taken address is selected as the next fetch address. For a jump or a jump to subroutine, the target addresses are computed at run-time. The indirect branch predictor records and predicts these addresses. The address returned by the indirect branch predictor is selected as the next fetch address whenever a jump or a jump to subroutine is encountered. For a subroutine return, the address on the top of the RAS is selected as the next fetch address. For every subroutine call (i. e., either a branch to subroutine or a jump to subroutine), the return address is pushed onto the RAS. (The return address, like the not-taken address for a conditional branch, is computed from the address used to access the BTB.) And finally, for every subroutine return, an address is popped off the RAS.

Since a BTB is a cache, in some cases, a miss is simply the result of standard cache behavior. Like other caches, a BTB suffers from conflict, compulsory, and capacity misses. Hence, the information regarding a particular branch may not be resident in the BTB when that instruction is fetched from the instruction cache. This latter type of miss is highly undesirable: when the branch predictor cannot identify which instructions fetched from the instruction cache are branches, it cannot accurately predict the sequence of fetch blocks that comprises the dynamic instruction stream. When the latter type of miss occurs, the branch predictor is typically led astray, which results in one or more cycles of nonproductive fetching. The branch predictor is righted only after the branch whose information was absent from the BTB has been decoded. To reduce the number of occurrences of this latter type of miss, the size and/or associativity of the BTB can be increased. Unfortunately, this increases the access time of the BTB, and, as a result, may increase the processor cycle time. To reduce the number of occurrences of these misses without increasing the processor cycle time, it may be necessary to use a multi-level BTB hierarchy instead of a single BTB [105].

For processors that can fetch up to one fetch block from the instruction cache each cycle, selecting the address to access the BTB is not straightforward, since any of the instructions that are fetched simultaneously from the instruction cache may be branches. The base microarchitecture accesses the BTB using a variation of Yeh and Patt's fetch address based indexing [135]. (The base microarchitecture fetches up to 3 fetch blocks from the instruction cache each cycle. However, as mentioned earlier, it uses a conventional fetch mechanism, which, in a real microarchitecture, would be accessed once per cycle and would produce one fetch block per access. To fetch multiple fetch blocks in a cycle, the base microarchitecture accesses its conventional fetch mechanism multiple times per cycle, obtaining one fetch block per access.) Their scheme uses the address of an instruction that dominates [4] a branch to identify the branch. When an instruction fetch occurs for the sequence of instructions that starts with the dominating instruction, the BTB identifies which one of the instructions in the sequence is the branch. In cases where the branch is in a fetch block that can be entirely contained within a single first level instruction cache line, the address of the first instruction in the fetch block is used to identify the branch. In cases where the branch is in a fetch block that cannot be entirely contained within a single first level instruction cache line, the branch is identified by its first level instruction cache line address. The first level instruction cache line address is calculated by taking the address of the branch and setting all of the bits of this address that specify the location (or offset) of the branch within its first level instruction cache line to zero.

During fetch, the BTB and the first level instruction cache are accessed with the fetch address. In parallel with the BTB access, the branch predictor computes the 3 predictions: the direction of a conditional branch, the target of an indirect branch, and the target of a subroutine return. The cache returns the instruction at that fetch address and all subsequent instructions stored in the same cache line as the instruction at that fetch address. A hit in the BTB indicates that one of the instructions returned by the cache is a branch. The branch predictor uses the information returned by the BTB to identify which of these instructions is the branch. Only the instructions up to and including the branch are written into the fetch buffer entry allocated to the fetch; i. e., they are written into the fetch buffer entry that is allocated to the cache read request that generated the fetch. The instructions after the branch belong to a different fetch block than the branch and the instructions before it, and are discarded. If the instructions after the branch are needed, they are obtained via a separate cache read request. The branch predictor also uses the information returned by the BTB to calculate the target address of this branch. This target address is selected as the next fetch address. For example, if the BTB information indicates that the fetched branch is a subroutine return, the return target—which was computed by the branch predictor during the BTB access—is selected as the next fetch address. A miss in the BTB (at least in most cases, when the miss is *not* the result of standard cache behavior) indicates that none of the instructions returned by the cache are branches. All of the instructions returned by the cache belong to the same fetch block, and are written into the fetch buffer entry allocated to the fetch. On a BTB miss, the address of the first level instruction cache line that follows the fetched cache line is selected as the next fetch address. [1]

[1] A cache read request reads a single cache line from the first level instruction cache. The request also

On a BTB hit, the BTB returns the type of the branch and one of its possible target addresses, as it did for processors that only fetch a single instruction each cycle. In addition, it returns a few bits that indicate the location of the branch within the set of instructions that were fetched from the cache. That is, it returns an offset into the cache line that was fetched from the first level instruction cache. The offset specifies the location of the branch. This offset is used to calculate the not-taken (fall-through) addresses for conditional branches, and the return addresses for subroutine calls. The information returned by the BTB (i. e., the branch's type, offset, and one of its possible target addresses) is used by the branch predictor to select the next fetch address and to control the RAS. Except for using the branch's offset to calculate its not-taken address (for a conditional branch) and its return address (for a subroutine call), the next fetch address selection and the RAS control are identical to that of processors that can only fetch a single instruction from the instruction cache each cycle.

The major drawback of Yeh and Patt's fetch address based indexing scheme is that it creates "hot spots" in the BTB [135]. (A "hot spot" is a set of BTB entries that are read and/or written a disproportionate number of times.) Every branch that is in a fetch block that cannot be entirely contained within a single first level instruction cache line is identified by its first level instruction cache line address. As a result, the low-order bits of many of the identifying addresses are zero. These low-order bits are used as part of the set index into the BTB. Consequently, some of the BTB sets (i. e., those sets with an index whose low-order bits are zero) are used more heavily than others.

specifies which instructions from that cache line are needed. The fetch address together with the BTB information fully specify which instructions have been requested. The fetch address identifies the first level instruction cache line that needs to be read, as well as the first instruction from that cache line that is needed. On a BTB hit, the BTB identifies the last instruction that is needed. On a BTB miss, the last instruction that is needed is the last instruction in the cache line.

Figure 7.3 shows how the bits of the fetch address are used when accessing the BTB and first level instruction cache. The lower two bits of every fetch address are guaranteed to be 0, since memory is byte addressable and instructions are aligned at 4-byte boundaries. These two bits are ignored when accessing the BTB and the instruction cache. For the BTB, the set index is specified by bits 2–10 of the fetch address and the tag is specified by bits 11–63. For the instruction cache, the line offset is specified by bits 2–5, the set index is specified by bits 6–13, and the tag is specified by bits 14–63. A first level instruction cache line address is an address whose offset bits are all zero. Any branch that is identified by its first level instruction cache line address will use a BTB entry that resides in a set with an index whose low-order 4 bits are all zero. Hot spots are created at every 16th set; i. e., every set with an index whose low-order 4 bits are all zero.

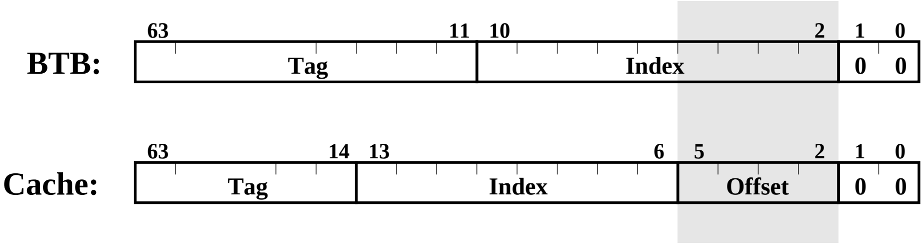


Figure 7.3: Figure showing how the bits of the fetch address are used when accessing the BTB and first level instruction cache

To eliminate some of these hot spots, the base microarchitecture uses a variation of Yeh and Patt's fetch address based indexing. In this variation, the bits of the fetch address that specify the BTB set index and the first level instruction cache line offset are XORed with the lowest bits of the fetch address that specify the BTB tag. The result, together with the BTB set index bits that do not specify the first level instruction cache line offset, is used as the BTB set index. For example, in Figure 7.3, bits 2–5 are used as BTB set index bits as well as first level instruction cache line offset bits. These bits are XORed with bits 11–14, which are the lowest 4 bits of the BTB tag. The 4 bit result, together with bits 6–10, which are BTB set index bits but not first level instruction cache line offset bits, form the 9-bit BTB set index. This variation eliminates some of the BTB hot spots because the XOR forces the low-order 4 bits of the BTB set index to take on values that are more uniformly distributed.

Figure 7.4 and Figure 7.5 plot the BTB miss rates for the SPEC and Non-SPEC benchmarks, respectively. The BTB miss rate is the percentage of branches in the dynamic instruction stream that experienced a BTB miss when they were fetched. [2] A BTB with a low miss rate is desirable, since pipeline bubbles are introduced whenever a branch is fetched and the information regarding that branch is not resident in the BTB. There are two bars for each benchmark. Each bar plots the miss rate of a 2048 entry, 4-way set-associative BTB. The bar labeled "Without XOR" plots the miss rate of a BTB that uses Yeh and Patt's fetch address based indexing scheme. Henceforth, their scheme will be referred to as fetch address based indexing without XOR. The bar labeled "With XOR" plots the miss rate of a BTB that uses my variation of Yeh and Patt's fetch address based indexing scheme. Henceforth, my scheme will be referred to as fetch address based indexing with XOR. In some cases, the BTB miss rate was very close to zero, so the bar may not be visible on the graph. The results show that the fetch address based indexing with XOR scheme performs much better than the fetch address based indexing without XOR scheme. For the fetch address based indexing without XOR scheme, the BTB miss rate averaged over all the benchmarks is 3.78%. For the fetch address based indexing with XOR scheme, the number of BTB misses (due to standard cache behavior) is reduced by 72%, and the average BTB miss rate falls to 1.04%.

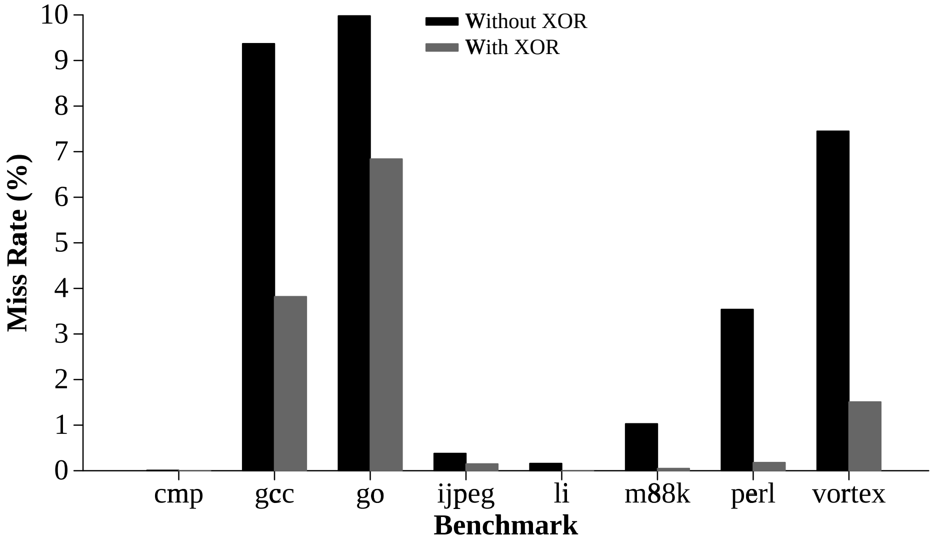


Figure 7.4: BTB Miss Rates—SPEC Benchmarks

[2] The BTB miss rate is equal to the number of BTB misses that occurred due to standard cache behavior (i. e., the number of undesirable BTB misses) divided by the number of executed branches. This metric ignores the BTB misses that do not impact performance; i. e., the BTB misses that are not the result of standard cache behavior.

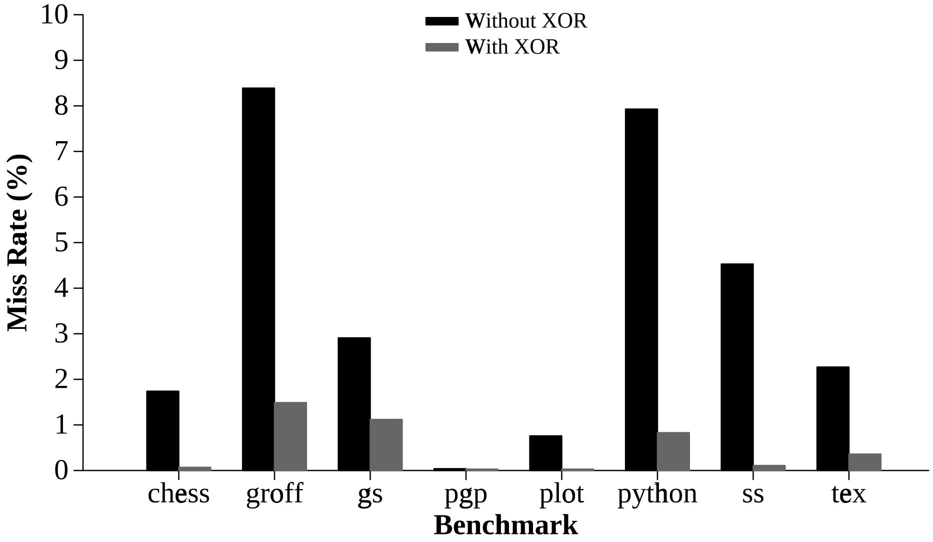


Figure 7.5: BTB Miss Rates—Non-SPEC Benchmarks

For the base microarchitecture and those derived from it, the BTB has 2048 entries and is 4-way set-associative. It uses the fetch address based indexing with XOR scheme. (Figure 7.3 and the accompanying text specify how the bits of the fetch address are used when accessing this BTB. The bars labeled "With XOR" in Figure 7.4 and Figure 7.5 plot the BTB miss rate for this BTB for each of the benchmarks.) Each BTB entry requires 61 bits of storage. Consequently, the entire BTB requires a little more than 15k bytes of storage.

Each BTB entry contains a valid bit, a 21 bit partial address tag, a 3 bit branch type field, a 30 bit partial target address, a 4 bit offset field, and a 2 bit field for implementing the BTB's least recently used (LRU) replacement policy. Although, instruction addresses are 64 bits in the Alpha AXP architecture, programs never address more than 4G bytes (i. e., 2^{32} bytes) of instruction data. The upper 32 bits of every instruction address are the same. Additionally, the lower 2 bits of every instruction address are guaranteed to be 0, since memory is byte addressable, and instructions are aligned at 4-byte boundaries. To reduce the size of each BTB entry, the upper 32 bits and the lower 2 bits of instruction addresses are not stored in the BTB entry. That is, only a partial address tag and a partial target address are stored. This reduces the size of the address tag from 53 bits to 21 bits, and the size of the target address from 62 bits to 30 bits. The offset field requires only 4 bits, since a first level instruction cache line contains 16 instructions.

7.1.2 Execution Core

Figure 7.6 is a block diagram of the execution core. The core contains 16 functional units. The functional units are not homogeneous. For example, only 8 of the 16 functional units can execute loads and stores. Each functional unit has its own Node Table (set of reservation stations), tag bus, and result bus. During decode, each instruction is assigned to a functional unit capable of executing that instruction. As the instructions are written (issued) into the Node Tables, each instruction is routed via the switch to the Node Table that belongs to its assigned functional unit. The Register Alias Table (RAT) is used by the decode/issue logic to rename registers. It also captures speculative register values, and keeps track of the architectural (non-speculative) register state.

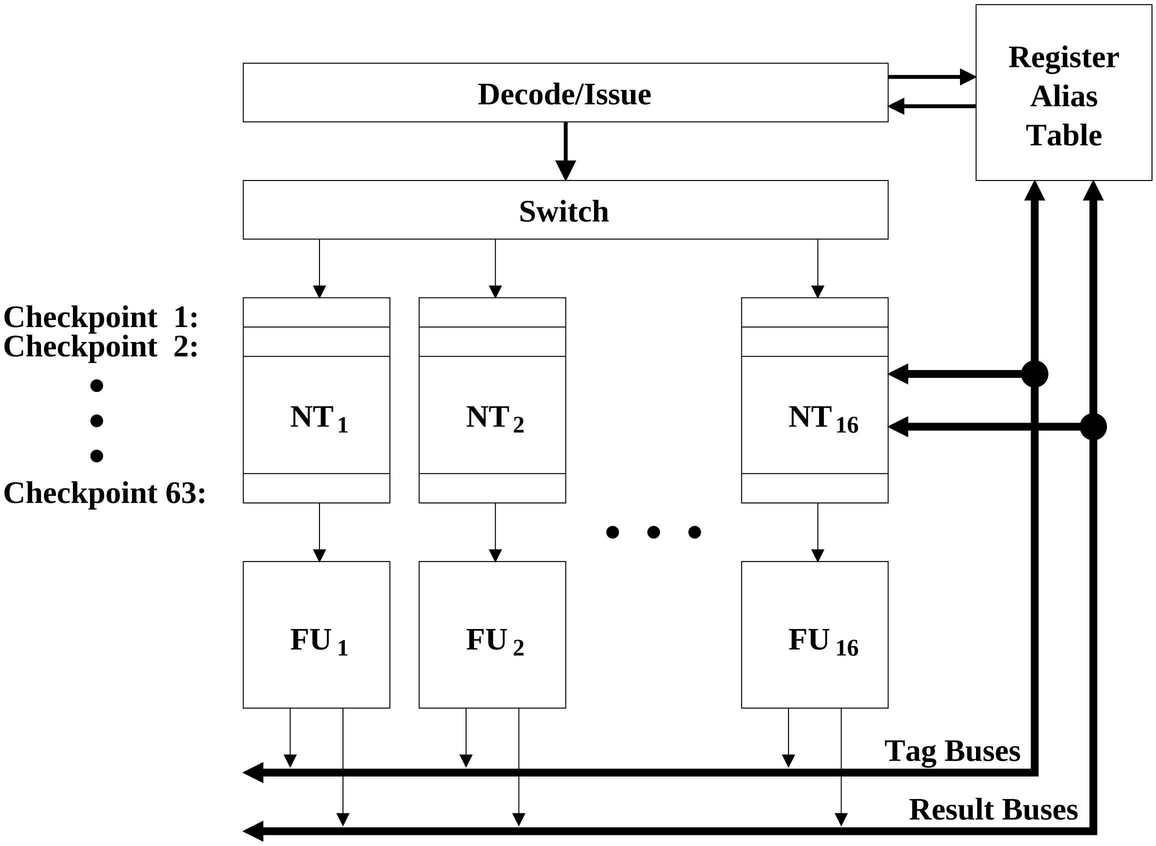


Figure 7.6: Block Diagram of the Execution Core

The state of the RAT and the Node Tables are protected via checkpointing [52, 53]. Checkpointing is used to quickly restore the machine to a known previous state in the event of a branch mispredict or exception. A checkpoint records the machine state at a particular point in the execution of the dynamic instruction stream. The machine supports 64 checkpoints. One of these checkpoints is used to store the machine's architectural (non-speculative) state. The remaining 63 checkpoints are used to store speculative state.

Instructions are decoded, issued, and retired in units of *issue packets*. An issue packet is a sequence of logically consecutive instructions that are decoded, issued, and retired together. An issue packet may contain any number of branches. Each issue packet is allocated a single checkpoint and a set of physical registers. One physical register is allocated for each instruction in the packet. The checkpoint records the machine state at a point just after the last instruction in the packet has executed. The set of physical registers—which are actually associated with the checkpoint rather than the issue packet—record the speculative register state associated with the checkpoint. At most one packet is issued each cycle, at most one packet is decoded each cycle, and at most one packet is retired each cycle. A packet can only be retired if all the instructions in the packet have completed without generating exceptions, and if the predicted targets of all the branches in the packet have been verified. Retiring a packet frees its checkpoint and associated physical registers.

The decode/issue logic performs four tasks: issue packet creation, functional unit assignment, dependency analysis, and operand fetch. These tasks are interrelated. However, much of their work occurs in parallel. Each of these four tasks is described in detail below.

The decode/issue logic examines the instructions in the fetch buffer and attempts to create a single issue packet each cycle. During functional unit assignment, each instruction in a packet will be assigned its own unique functional unit. To guarantee that this can occur, restrictions are placed on the size and composition of packets during packet creation. For the base microarchitecture and the microarchitectures derived from it, each packet could contain at most 16 instructions. At most 4 of those 16 instructions could be integer multiplies and floating point instructions, at most 8 could be loads and stores, and at most 4 could be branches and conditional moves. To handle serializing instructions, each serializing instruction was placed in its own packet with no other instructions. A packet containing a serializing instruction was not allowed to issue until all older instructions had retired. Once the packet containing the serializing instruction was issued, new packets were not allowed to issue until the serializing instruction had retired. The instructions that comprise a packet are removed from the fetch buffer when the issue packet is created.

Functional unit assignment is the task of assigning each instruction in an issue packet to a functional unit. Each instruction in the packet is assigned to a different functional unit. During the last stage of decode/issue, each instruction is routed via the switch to the Node Table that belongs to its assigned functional unit. Since each instruction is assigned to a different functional unit, at most one instruction needs to be written into each Node Table per cycle.

The decode/issue logic performs dependency analysis to determine which instructions consume values that are produced by instructions within the same packet. Source operands that are produced by instructions within the same packet must be tagged with the destination operand tag of the producing instruction. These tags override any source operand tags fetched from the RAT. (Dependency analysis occurs in parallel with operand fetch from the RAT.)

Operand fetch occurs as soon as the source and destination register identifiers for the instructions in the packet have been decoded. Essentially, the RAT behaves like a checkpointed register file. It contains a set of physical registers, and a checkpointed map that maps the architectural registers onto the physical registers. Each source register accesses the RAT to obtain its tag, and, if available, its data. The tag is just the address of the physical register that contains (or will contain) the data for the requested source register. Each destination register accesses the RAT to allocate a new physical register, to obtain the tag associated with that physical register, and to update the checkpointed map.

The execution core has 16 Node Tables. Each Node Table has 63 slots for instructions. Hence, the window size is 1008 (16 × 63) instructions. Each of the 63 slots within a Node Table belongs to one of the 63 checkpoints that are used to store speculative state. After the instructions have been decoded, each instruction in the issue packet is routed to the Node Table that belongs to its assigned functional unit. Each instruction is then written to the slot associated with the checkpoint allocated to the packet.

The tag of a value generated by a functional unit is broadcast on the functional unit's tag bus at least one cycle before the value is broadcast on the functional unit's result bus [130]. Each instruction stored in a Node Table monitors the tag buses to determine when its source operand values are generated. When a source operand value is generated, the value is read from the appropriate result bus and latched into the instruction's Node Table entry. When all of an instruction's source operands become available, the instruction wakes up; i. e., it becomes eligible for firing. Every cycle, each of the 16 Node Tables selects the oldest firable instruction within that Node Table and dispatches it to the functional unit associated with the Node Table. When an instruction begins execution, the functional unit immediately broadcasts the tag for the value that will be produced. While the instruction executes, dependent instructions examine the tag, potentially wakeup, and, if they wakeup, may be selected for dispatch. When the instruction finishes execution, its value is distributed over the functional unit's result bus to the RAT; to dependent instructions that woke up, were selected, and will need to use the result in the next cycle; and to other instructions in the Node Tables that await the result.

Figure 7.7 shows a pipeline timing diagram for the execution of two dependent instructions. The add instruction (R20 ← R1 + R2) executes during cycle number 1. At the beginning of the cycle, it broadcasts the tag for the value that it will produce (i. e., the value for register R20) on the tag bus. The subtract instruction (R1 ← R20 – 10), which is dependent on this value, notices that the tag for the value has been broadcast and wakes up. The scheduling logic selects the subtract instruction and dispatches it to its functional unit. In the middle of cycle number 1, the add instruction completes and bypasses its value over the result bus to all the functional units. At the beginning of cycle number 2, the subtract instruction receives the bypassed value and begins executing. Note that during cycle number 2, the add instruction continues to broadcast its value over the result bus until all the dependent instructions in the Node Tables have latched the value. (If the result bus is not pipelined, the add instruction must relinquish the result bus in the middle of cycle number 2 in order to allow any instruction that completed in that cycle to bypass its value over the bus.) Details of the instruction scheduling logic (i. e., instruction wakeup and select) are provided later in this section.

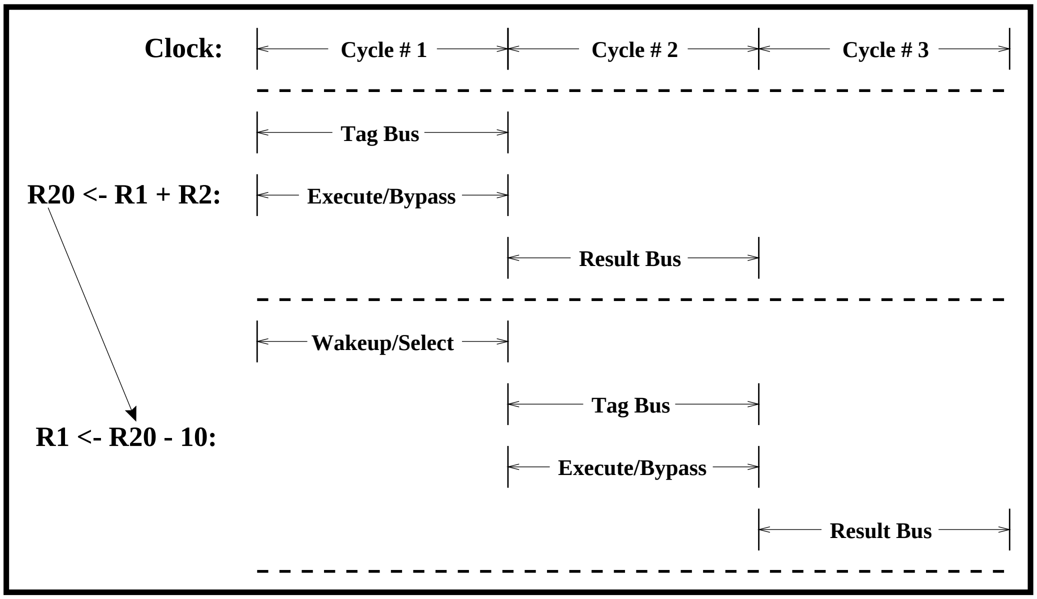


Figure 7.7: Pipeline timing diagram showing the execution of dependent instructions

The execution core contains 16 functional units. All functional units can execute simple integer instructions. The first 4 functional units can also execute integer multiplies and floating point instructions. The middle 8 functional units can execute loads and stores. The last 4 functional units can execute branches and conditional moves. The instruction class latencies are the same as those used in the previous chapters. They are provided in Table 4.1. All instructions are fully pipelined, except for floating point divide. A functional unit can only execute one floating point divide at a time; i. e., one floating point divide every 16 cycles. However, it can execute other instructions that aren't floating point divides while it is working on the floating point divide. A value produced by any functional unit is available to all 16 functional units in the same cycle that the value is produced. That is, there is no penalty for bypassing values between functional units.

Instruction Scheduling Logic

Each of the 16 Node Tables has logic for selecting the highest priority firable instruction within the Node Table and dispatching it to the associated functional unit. This logic—called the instruction scheduling logic—is identical for all 16 Node Tables. Figure 7.8 shows a block diagram of the instruction scheduling logic for one of the Node Tables.

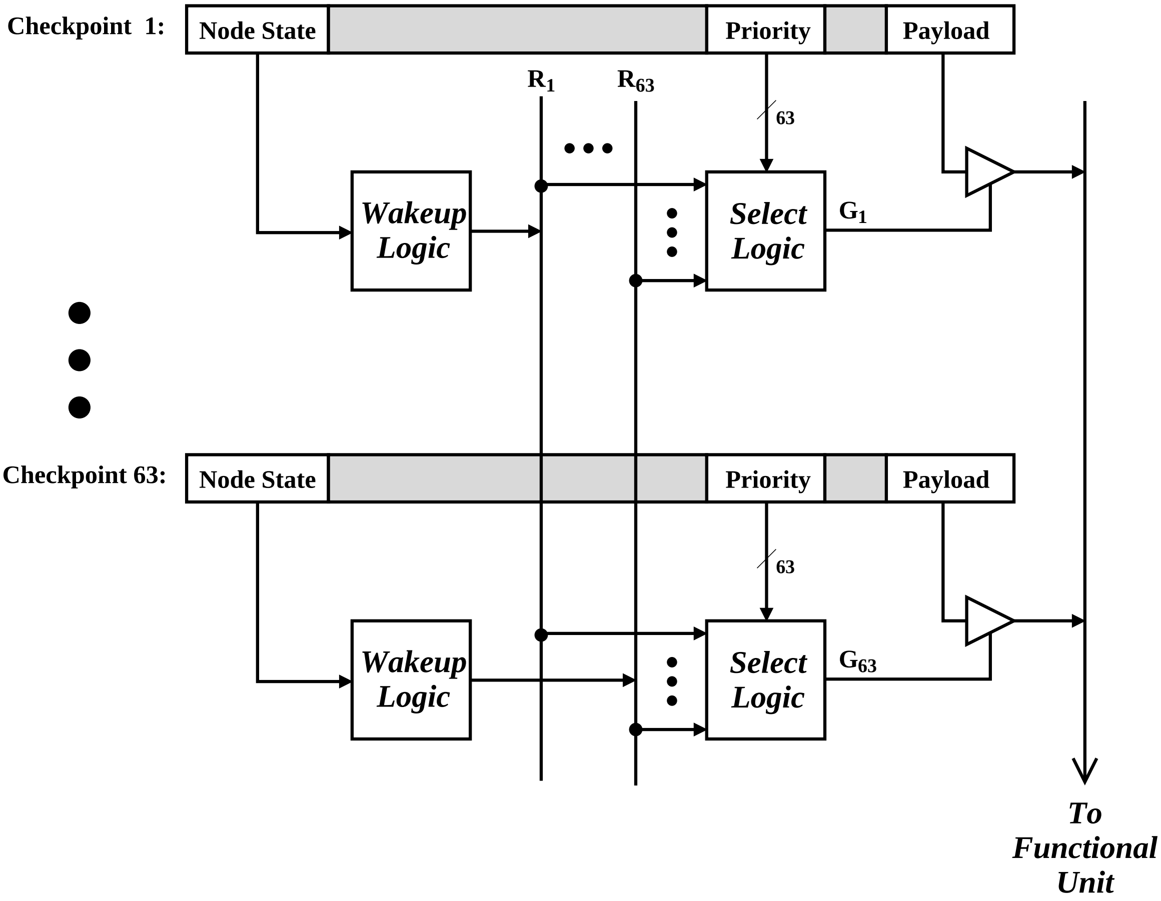


Figure 7.8: Block Diagram of the Scheduling Logic

The Node Table has 63 slots for instructions: one slot for each of the 63 checkpoints that are used to store speculative state. Each checkpoint owns a particular slot number. Specifically, checkpoint number X ($X \in \{1, 2, \ldots, 63\}$) owns slot number X.

Each slot stores the instruction's (node's) state, priority information, and payload. The node state contains the tags for the source operands, keeps track of which source operands are available, and keeps track of whether or not the instruction has been dispatched. The priority information identifies which slots contain instructions with a higher scheduling priority than the slot in question. This information is used to implement the dynamic scheduling heuristic [13], which, for the base microarchitecture and the microarchitectures derived from it, is the oldest first scheduling heuristic. The payload is the data (e. g., the decoded instruction opcode, the source operand values, and destination operand tag) that is delivered to the functional unit when the instruction is dispatched.

Each slot also contains wakeup and select logic. The wakeup logic uses the node state to determine when the instruction stored in the slot is firable. When the instruction is firable, the wakeup logic asserts the slot's request line. (The request lines are labeled R_1–R_{63}.) For example, when the instruction in slot number 1—which belongs to checkpoint number 1—is firable, the wakeup logic asserts R_1. The select logic examines the request line of the slot associated with the select logic, and the request lines of all other slots. Using the priority information, it determines whether a request from the associated slot can be granted. If it can, the select logic asserts the slot's grant line. (The grant lines are labeled G_1–G_{63}.) For example, if R_1 is asserted, and the scheduling priority of slot number 1 is higher than that of all other slots whose request lines are asserted, then the select logic associated with slot number 1 will assert G_1. The select logic is designed such that at most one of the 63 grant lines will be asserted in any given cycle. A slot whose request is granted gates its payload onto the bus connected to the functional unit associated with the Node Table. That is, a slot whose request is granted is dispatched.

Figure 7.9 shows the wakeup logic for one of the slots. The node state from Figure 7.8 is shown in more detail in this figure. Each instruction has two source operands: α and β. The node state contains three fields for each operand. The one bit ready field is set if the source operand is available, and reset if it is not. The CAM (Content Addressable Memory) field contains the tag for the source operand value. When the tag for the value is broadcast over one of the 16 tag buses (Tag_1–Tag_{16}), the CAM asserts its match line. Asserting the match line causes the ready bit to be set in the following cycle. The whence field specifies which of the 16 tag buses will be used to broadcast the tag for the source operand value. Each of the 16 functional units owns (and broadcasts tags over) a particular tag bus. Specifically, functional unit number X ($X \in \{1, 2, \ldots, 16\}$) owns tag bus number X. During instruction decode, the RAT records not only the destination operand tag for each instruction, but also the number of the functional unit assigned to that instruction.

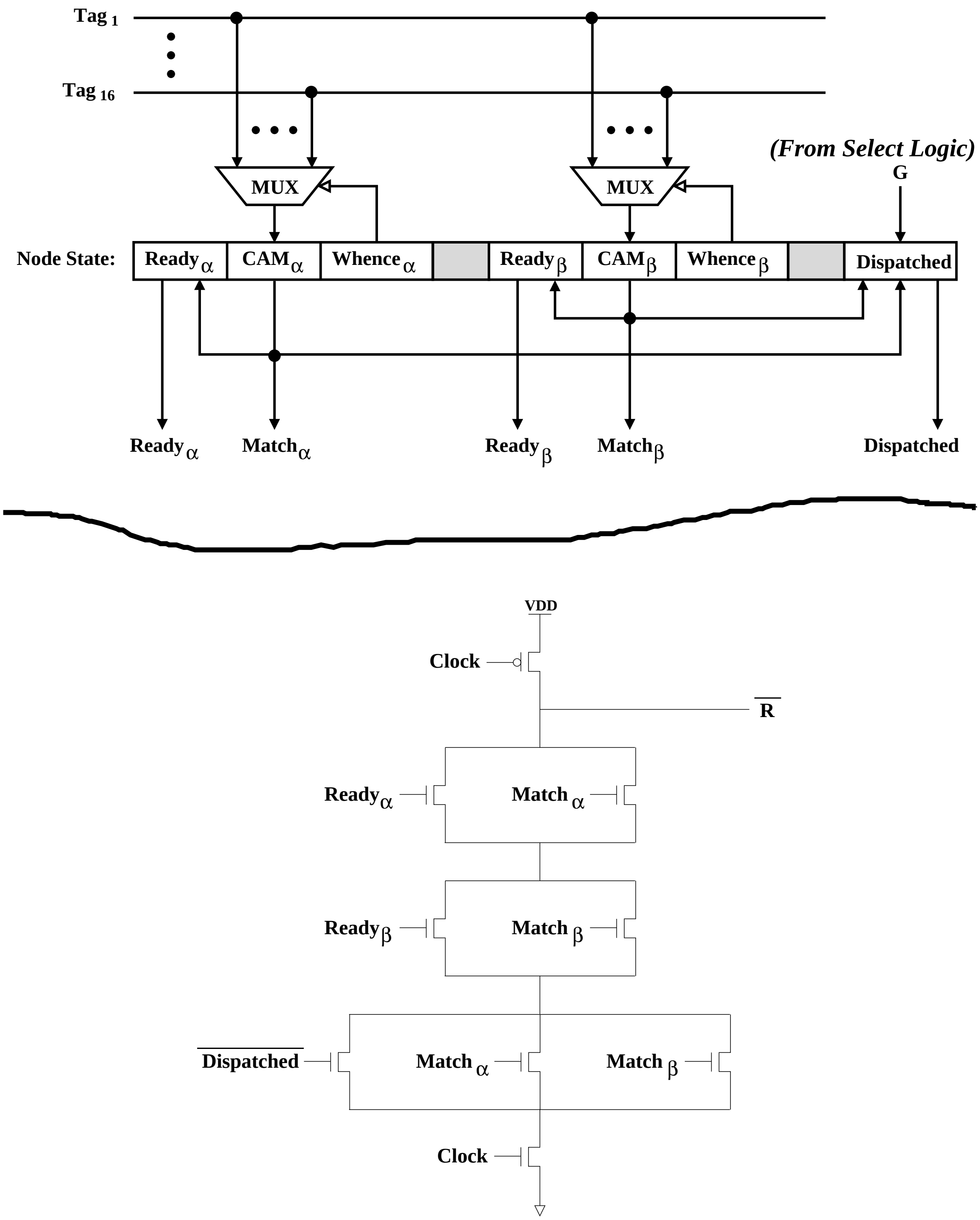


Figure 7.9: Wakeup Logic

When source operands are fetched from the RAT during decode, the tag is returned along with the number of the functional unit that will produce the value associated with that tag. This number is used to fill the whence field.

For the base microarchitecture and those derived from it, an instruction may initially execute using incorrect source operand values. This instruction will receive "updates" of its source operand values. Whenever it receives an update, it wakes up and generates a request for dispatch. Eventually, the instruction receives all the correct source operand values, is dispatched, and then re-executed correctly. The node state contains a one bit field, called the dispatched field, which prevents an instruction from being dispatched more than once with the same set of source operand values. When the instruction is initially written into the slot, this bit is reset. Whenever the instruction receives a new source operand value, one of the match lines is asserted, and, in the following cycle, the dispatched bit is reset. When the slot's grant line (i. e., G) is asserted, the instruction is dispatched, and, in the following cycle, the dispatched bit is set. If an instruction receives a new source operand value and is immediately dispatched (i. e., if one of the match lines and the grant line are asserted in the same cycle), the dispatched bit is set rather than reset in the following cycle.

The bottom of the figure shows the logic for generating the request signal. The slot generates a request if both source operands (α and β) are or will be available in the next cycle, and if the instruction has not yet been dispatched with the current set of source operand values. An operand is currently available if its ready bit is set. An operand will be available in the next cycle if its match line is asserted; that is, if its tag is being broadcast in the current cycle. (Recall that the tag for a value is distributed one cycle before the value is distributed.) An instruction has not yet been dispatched with the current set of source operand values if either the dispatched bit is reset, or if a tag for a source operand value is being broadcast in the current cycle. (Recall that the dispatched bit is not reset until the cycle after a tag match occurs.)

Figure 7.10 shows the select logic for slot number 1. The select logic for other slots is similar. The slot's request line is labeled R_1 and its grant line is labeled G_1. Request lines from the other 62 slots are labeled R_2–R_{63}. $\mathrm{P}_{x>1}$ ($X \in \{2, 3, \ldots, 63\}$) is the x^{th} bit of the priority information field associated with slot number 1. It is set if the instruction in slot number x has a higher scheduling priority than the instruction in slot number 1. It is reset otherwise. The slot's grant line is precharged high during the first clock phase. During the second clock phase, the line is discharged if the slot does not generate a request, or if any slot that has a higher scheduling priority than slot number 1 generates a request.

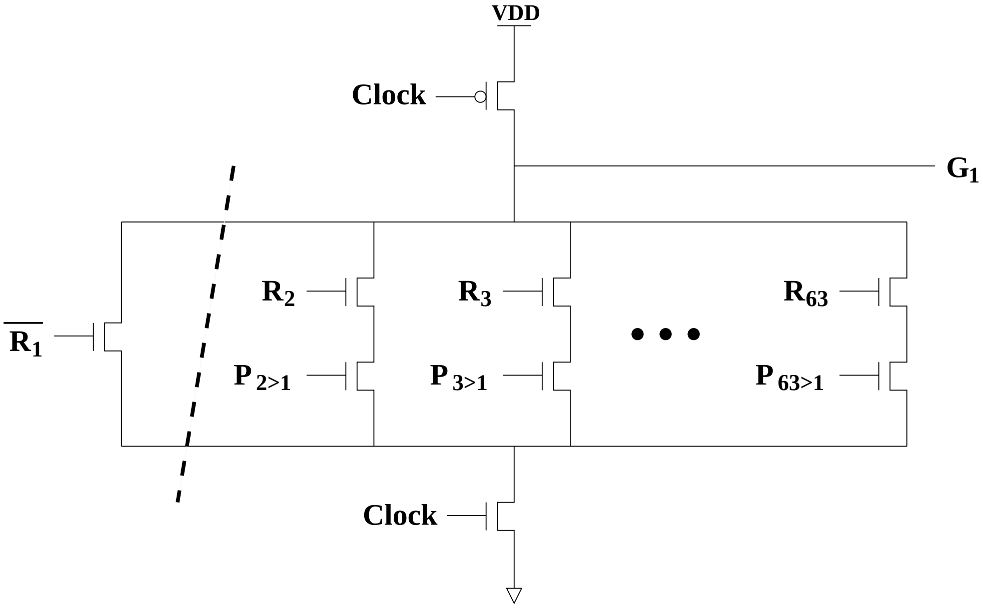


Figure 7.10: Select Logic for Slot Number 1

The priority information is actually maintained on a per slot basis rather than on a per Node Table entry basis. That is, the machine has 1008 Node Table entries (16 Node Tables × 63 slots per Node Table), but only 63 logical (as opposed to physically implemented) priority information registers. Each of the priority information registers is shared by the 16 Node Table entries (one Node Table entry for each of the 16 Node Tables) that are assigned to the same slot number.

As was mentioned earlier, the priority information is used to implement the oldest first scheduling heuristic. The age of an issue packet is the age of the first instruction in the packet. Issue packets are given priority based on their age. A given packet will have a higher priority than all other packets that are younger than it. Each packet is allocated a checkpoint, and each checkpoint owns a particular slot number. The priority of an instruction stored in a slot is the priority of that instruction's packet. Oldest first scheduling is performed by selecting the slot with the highest priority firable instruction.

For conventional machines and machines that support out-of-order fetch, instructions, and hence issue packets, are issued in strict program order. When a packet is issued, all packets stored in the Node Tables are older than it. As the packet is issued, the priority information register associated with the packet's slot number is written with a value that indicates that all packets stored in the Node Tables are older than it. Once a packet is installed in the Node Tables, any new packet issued will be younger than it. Consequently, the packet's priority information register doesn't need to be updated when a new packet is issued. On the other hand, when a packet is retired, the retiring packet is older than the installed packet. When the retiring packet is removed from the Node Tables, the installed packet's priority information register is updated to indicate that the retiring packet's slot no longer contains older instructions.

For machines that support out-of-order fetch/decode/issue, packets may be issued out-of-order. Special consideration is required to accommodate for this. Every packet that is created by the machine knows its age. When a new packet is issued, it may be older than some of the packets already stored in the Node Tables. As the new packet is issued, every packet in the Node Tables compares its age to that of the new packet. The results of these comparisons are used to create the value that is written into the new packet's priority information register. The value written indicates which of the packets in the Node Tables are older than the new packet. Once a packet is installed in the Node Tables, packets subsequently issued may be older than it. The packet's priority information register is updated when a new packet is issued based on the result of the comparison between its age and the age of the new packet. If the new packet is older than it, its priority information register is updated to reflect that fact.

7.1.3 Load/Store System

Figure 7.11 is a block diagram of the load/store system. Only 8 of the 16 functional units execute loads and stores. The functional units calculate the addresses of load and store memory accesses and then pass them on to the first level data cache and memory disambiguator. The machine's architectural (non-speculative) memory state is contained in the cache/memory hierarchy, and its speculative memory state is contained in the memory disambiguator. For a load, the load data is obtained from either the cache/memory hierarchy or the memory disambiguator, and then returned to the functional unit that calculated the load's address. The functional unit broadcasts the tag and data value for the load on its tag bus and result bus, respectively. For a store, the memory disambiguator obtains the store data from the execution core, and then buffers the data until the store retires. When the store retires, the store data is committed to the architectural memory state; i. e., it is written to the cache/memory hierarchy.

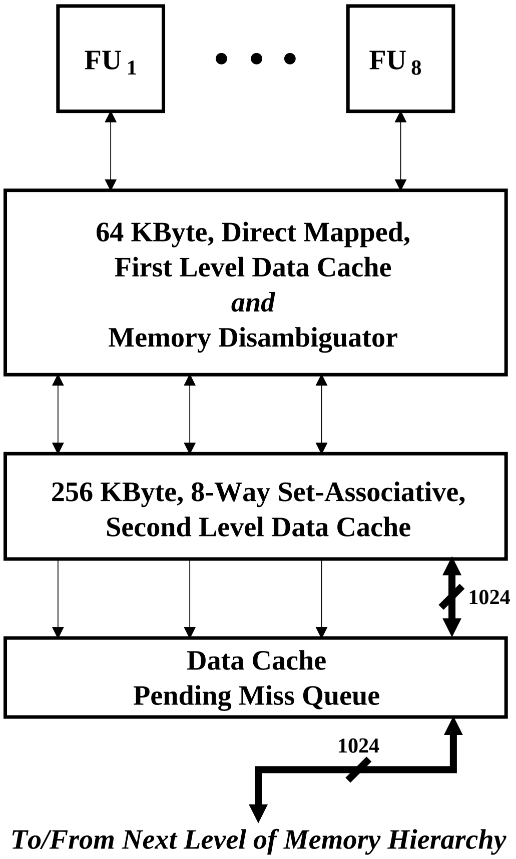


Figure 7.11: Block Diagram of the Load/Store System

The first level data cache is a fully pipelined, non-blocking, direct mapped, 64k byte cache, with an 8 byte line size and a two cycle latency. The cache implements a write-through store policy, as do the first level data caches of the Compaq Alpha 21164 [29] and the Cyrix M3 [26]. Write-through caches generate more write traffic to the next level cache than write-back caches. However, they are easier to implement. The cache can service up to 8 requests per cycle. A request may either be a cache read, a cache write, or a cache fill. (A cache fill—which occurs in response to a cache miss—writes a new line, including both tag and data, into the cache.) To provide the necessary bandwidth, the cache is 8-way interleaved. The cache lines are divided into 8 independent banks. The low order 3 bits of the cache index specify the bank number. Each bank can service at most one request per cycle; i. e., each bank is single ported. Hence, although the peak number of requests serviced per cycle is 8, the actual number of requests serviced per cycle may be lower due to bank conflicts.

The memory disambiguator implements naive memory dependence speculation [22]. An address calculation for a load or a store begins once the source register operands required for the address calculation are available. After an address has been calculated, it is inserted and stored in the disambiguator. As each load address is inserted into the disambiguator, the disambiguator checks whether it contains the address of an aliasing store. If it does, the disambiguator identifies the aliasing store, and, when the store data becomes available, forwards the store data to the load. The load then distributes its result. If it does not, the load obtains its data from the cache/memory hierarchy and immediately distributes its result. As each store address is inserted into the disambiguator, the disambiguator checks whether it contains the addresses of any dependent loads. The disambiguator identifies the dependent loads, and, when the store data becomes available, forwards the store data to them. These loads then distribute their results.

With this memory disambiguator, a load may distribute its result before its dependencies have been correctly identified. As a result, the load may initially distribute incorrect data. Instructions that are dependent on the load may execute using this incorrect data, producing and distributing still more incorrect data. Eventually, however, the load's dependencies are correctly identified, and the load's correct data is located and distributed. Dependent instructions that initially executed using incorrect data eventually receive their correct data, re-execute, and then distribute their correct results. All instructions that initially executed incorrectly are eventually re-executed as the correct values propagate down the dependence graph.

The second level data cache is a fully pipelined, non-blocking, 8-way set-associative, 256k byte, write-back cache, with a 128 byte line size and an 8 cycle latency. It has 3 ports which are used to service the cache read and cache write requests that originate from the first level data cache. Up to 3 such requests can be serviced per cycle. It has a single port dedicated for the cache fill and cache copyback requests that originate from the Data Cache Pending Miss Queue. (A cache copyback occurs whenever a dirty cache line is replaced. The copyback reads the dirty cache line, both tag and data, from the cache and then places the cache line in a read-only state. The cache line read by the copyback is eventually copied back to the next level of the memory hierarchy.) At most one cache fill or cache copyback can be serviced per cycle.

Cache read and cache write requests that miss in the second level data cache are placed in the Data Cache Pending Miss Queue (Data Cache PMQ). The PMQ can store up to 16 miss requests. For cache read requests that missed in the second level data cache, the PMQ obtains the sought after data from the lower levels of the memory hierarchy. The PMQ also marshals the cache fills and cache copybacks that are brought about by these cache misses. For cache write requests that missed in the second level data cache, the PMQ buffers the write until it is sent to the next level of the memory hierarchy.

Figure 7.12 is a block diagram of the first level data cache. Each of the 8 functional units that can execute loads and stores can submit one request per cycle. For each of the 8 banks, arbitration logic determines which request may access that bank. Requests that win arbitration are forwarded to their respective banks via the upper switch. Requests that lose arbitration re-try in the next cycle. Each bank contains a piece of the cache; i. e., an 8th of the lines in the cache. For each load, the data is obtained. The tag and data are then routed, via the lower switch, to the tag bus and result bus of the functional unit that calculated the load's address. The tag and data are then broadcast over these buses.

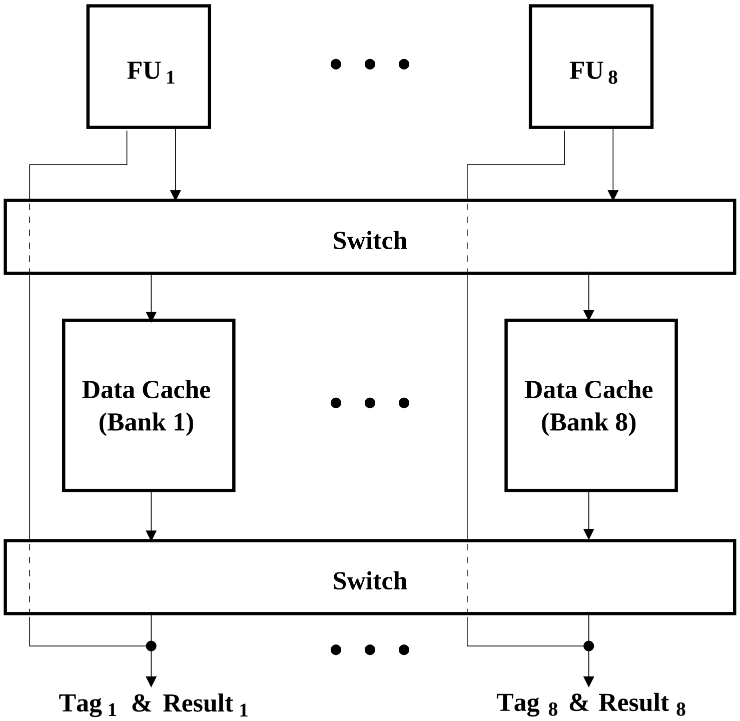


Figure 7.12: Block Diagram of the Dual Switch First Level Data Cache

The lower switch is needed for the following reason. Each load (and indeed, each instruction) is assigned a functional unit during instruction decode. The number of the functional unit assigned to the load is recorded in the RAT alongside the load's destination operand tag. When an instruction that is dependent on the load fetches the dependent source operand from the RAT, the tag for the load's destination operand is returned along with the number of the functional unit that the load was assigned to. This number is used to fill the source operand's whence field (see the wakeup logic in Figure 7.9). Recall that the whence field identifies which functional unit's tag bus and result bus will be used to broadcast the tag and data value for the source operand.

Figure 7.13 shows one possible way of eliminating the lower switch. Each bank is hardwired to a particular tag bus and result bus. A load forwarded to bank number X ($X \in \{1, 2, \ldots, 8\}$) will always broadcast its tag and data value over the tag bus and result bus that belong to functional unit number X. For the wakeup logic to work properly, whence fields are re-written as load requests are forwarded to their banks. When a functional unit forwards a load request to a bank, it broadcasts the tag for the load on its tag bus and asserts its change whence field line (ΔWhence). It also broadcasts the number of the functional unit whose tag bus and result bus will be used to distribute the load's tag and data value on the new whence bus (NewWhence). This number is equal to (or can easily be obtained from) the load's destination bank number. The instruction scheduling logic ignores tags broadcast over tag buses whose associated change whence field lines are asserted. Broadcasting the tag when the change whence field line is asserted causes all instructions that are dependent on the load to re-write the whence fields of their dependent source operands. The value written into the whence field is the value broadcast over the new whence bus.

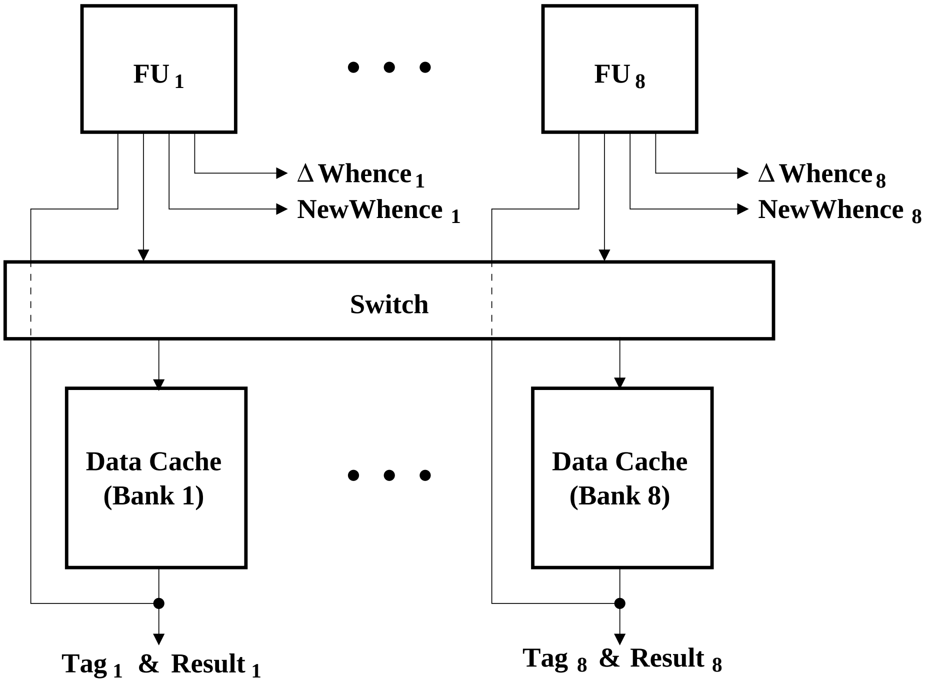


Figure 7.13: Block Diagram of the Single Switch First Level Data Cache

7.1.4 Main Memory

Processor issue widths are increasing and processor cycle times are decreasing. As a result, memory bandwidth requirements are growing. In addition, processor cycle times are decreasing faster than off-chip memory access times. Consequently, the memory latency, in terms of processor cycles, is also growing.

Currently, a different solution is used for each of these two problems. The solution to the problem of the growing memory bandwidth requirements has been the adoption of high bandwidth DRAMs such as Synchronous DRAMs and Rambus DRAMs. Unfortunately, these devices don't provide low latency. The solution to the problem of the growing memory latency has been to increase the size of the processor's on-chip—as well as off-chip—SRAM caches. For example, the Hewlett-Packard PA-8500, which is built in a 0.25 μm process technology, has 1.5M bytes of on-chip cache [8]. The Compaq Alpha 21364, which will be built in a 0.18 μm process technology, will also have 1.5M bytes of on-chip cache [45]. With each new generation of process technology, the amount of cache that can be implemented on-chip will increase. Within 5 to 10 years—the time frame when I expect that the base microarchitecture can be built—it will be possible to implement 16M bytes to 64M bytes of on-chip cache. [3] For programs with good spatial or temporal locality, large caches solve the problem of growing memory latency by shielding the processor from the memory latency. Unfortunately, not all programs have good locality.

In the future, the solution to both these problems may be to implement on-chip DRAM. Figure 7.14 shows the proposed solution. Main memory caches pages of the virtual address space(s). Essentially, main memory is a "page" cache: each line of the page cache (i. e., each page frame) contains a virtual page of a running process. Since DRAM density (bits/cm^2) is about 5 times higher than that of SRAM, within 5 to 10 years, it will be possible to implement a 64M–256M byte on-chip DRAM primary page cache; that is, a 64M–256M byte on-chip DRAM primary main memory. Placing main memory on the same chip as the processor allows the width of the memory bus to be increased. The width of the on-chip memory bus can be scaled to provide the required memory bandwidth. This solves one of the two problems. On-chip DRAM can be optimized to provide low latency.

[3] The Compaq Alpha 21364, which has 1.5M bytes of on-chip cache, is expected to ship in the fourth quarter of 2000 [45]. Assuming that the transistor budget—and hence the amount of on-chip cache—doubles every 18 months, the amount of on-chip cache will be about 16M bytes in the fourth quarter of 2005, and about 64M bytes in the fourth quarter of 2008.

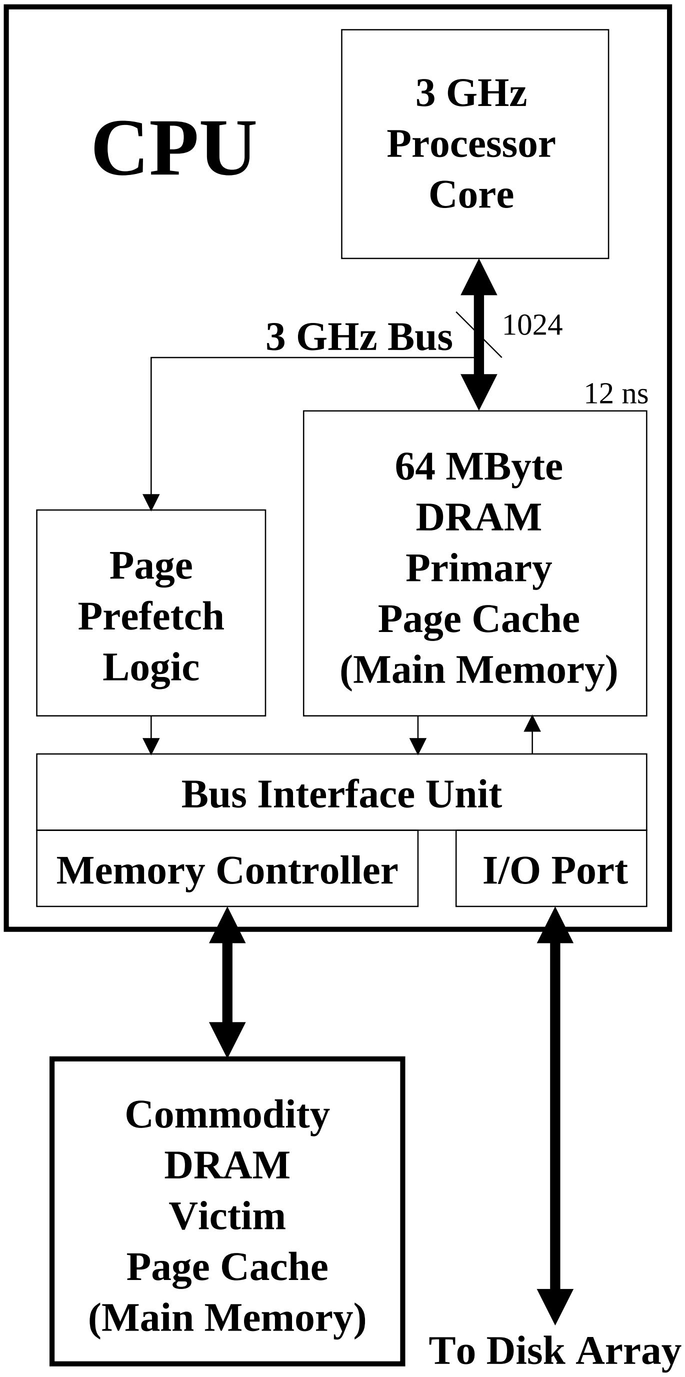


Figure 7.14: Main Memory Architecture

This solves the other problem. An example of an on-chip DRAM is the 8M byte DRAM macro for ASICs that was recently developed by NEC Corporation [68]. This DRAM has a 6.8ns random access time and a 9.1ns complete random access cycle. It can be configured to read or write up to 1024 bits of data. Naritake et al. [93] used this DRAM to create a 12ns, 8M byte, on-chip, DRAM secondary cache for a MIPS R10000 microprocessor [89]. In the figure, I assumed that the size of the on-chip DRAM primary page cache will be 64M bytes, that its random access time will be 12 nanoseconds, and that the width of its bus to the processor core will be 1024 bits.

In addition to the on-chip DRAM primary page cache, I expect that there will be a large off-chip, high latency, victim page cache built out of commodity DRAMs. When a program references a page that does not reside in either the primary page cache or the victim page cache, a page fault occurs. The missing page is retrieved from disk and written into the primary page cache. Before the new page can be written into its cache line (i. e., page frame), however, the processor and the operating system must deal with any (old) page that currently resides in the cache line. If the old page is unlikely to be used again, it is either discarded (if the page is clean) or written back to disk (if the page is dirty). If the old page is likely to be used again, it is transferred to the victim page cache. The old page may replace a page stored in the victim page cache, in which case the page to be replaced is either discarded or written back to disk.

Prefetching will be used to insure that the pages required by the processor core are resident in the on-chip, low latency, primary page cache. Prefetch logic will monitor the transactions on the bus between the processor core and the primary page cache. It will use this information to anticipate which pages are about to be accessed that are not in the primary page cache. It will then try to fetch these pages from either the off-chip, high latency, victim page cache or the disk array, and place them in the primary page cache—all before the accesses occur. Prefetching a page from the victim page cache usually entails swapping the contents of two cache lines: the page stored in the victim page cache is swapped with the page stored in its destination cache line in the primary page cache. To allow pages to be swapped without operating system intervention, it may be necessary to add an extra level of indirection to the main memory access. The virtual page number of a memory access is translated into a physical frame number using the standard translation process. The physical frame number is then translated into a cache identifier (either primary or victim) and a cache line number using a table that is maintained by the processor. [4] As in multiprocessors that use the cache-only memory architecture (COMA) model, the physical address space is dynamically mapped onto the caches. The processor can swap two pages without altering the physical address space by (1) swapping the contents of the two cache lines that contain those pages, and (2) swapping the contents of the two table entries associated (via physical frame number) with those pages. Note that the cache that contains a requested page is known after the table access has completed. Hence, for each requested page, either the primary page cache is accessed, or the victim page cache is accessed, but it is never the case that both caches are accessed.

The microarchitecture used for the remaining experiments in this dissertation uses the main memory architecture shown in Figure 7.14. All benchmarks use less than 64M bytes of memory, so I only model the processor core, the primary page cache, and the bus between them. Hereafter, the primary page cache will just be called main memory. Main memory is connected to the processor via a 1024 bit split-transaction bus. Note that the line sizes for both the second level instruction cache and the second level data cache are also 1024 bits. Ergo, an entire cache line can be transferred over the bus in a single bus cycle. The bus is assumed to operate at the frequency of the processor; i. e., 3 GHz. Main memory is fully pipelined: it can accept a new request (either read or write) every cycle. Requests originate from the Pending Miss Queue (PMQ) for the second level instruction cache, and the PMQ for the second level data cache. Once a read request gains access to the bus, 37 cycles (12.33 ns) are required to access the on-chip DRAM and return the data to the appropriate PMQ. The bus is only used in the last of these 37 cycles to transfer the data to the processor. Once a write request gains access to the bus, it occupies the bus for a single cycle while it transfers its data to main memory.

[4] To avoid two sequential translations, each TLB entry contains the cache identifier and the cache line number for its associated virtual page number in addition to the physical frame number.

7.2 Apparatus for Out-of-Order Fetch

A processor with out-of-order fetch initiates fetch requests in program order, but allows these requests to complete out-of-order. The branch predictor produces, in program order, a sequence of fetch requests for the instructions that comprise the dynamic instruction stream. Each of these requests initiates an instruction cache fetch. The branch predictor is completely decoupled from the instruction cache. That is, the branch predictor continues producing fetch requests regardless of whether or not the requests hit in the instruction cache. Due to the occurrence of instruction cache misses, the fetch requests may complete (i. e., obtain their data) out-of-order. As they complete, they write their data into the fetch buffer entries that were allocated to them. Processors with out-of-order fetch still decode and issue instructions in program order. Consequently, the instructions, even though they may be inserted into the fetch buffer out-of-order, are always removed from the fetch buffer in program order.

There is one advantage—other than performance—for implementing out-of-order fetch. For a processor without out-of-order fetch, the branch predictor and the instruction cache stall while the cache line associated with the miss is fetched from the lower levels of the memory hierarchy. Fetch requests logically following the request for the cache line that missed are flushed from the machine. When the fetch completes, the cache line is installed in the instruction cache and the requested instructions are written into the fetch buffer. Also, since the fetch requests logically following the request for the cache line that missed were flushed, the branch predictor is forced to re-generate those requests; i. e., the branch predictor is reset so that when it is restarted, it will restart an earlier point in the sequence of fetch requests. After this has been accomplished, both the branch predictor and the instruction cache are restarted. By stalling, flushing, and then restarting, the instructions are guaranteed to be written into the fetch buffer in program order. For a processor with out-of-order fetch, instructions do not have to be written into the fetch buffer in program order, so the processor does not stall, flush, and restart on a miss. On a miss, the branch predictor does not need to be restarted at an earlier point in the sequence of fetch requests. This somewhat simplifies the design and implementation of the branch predictor.

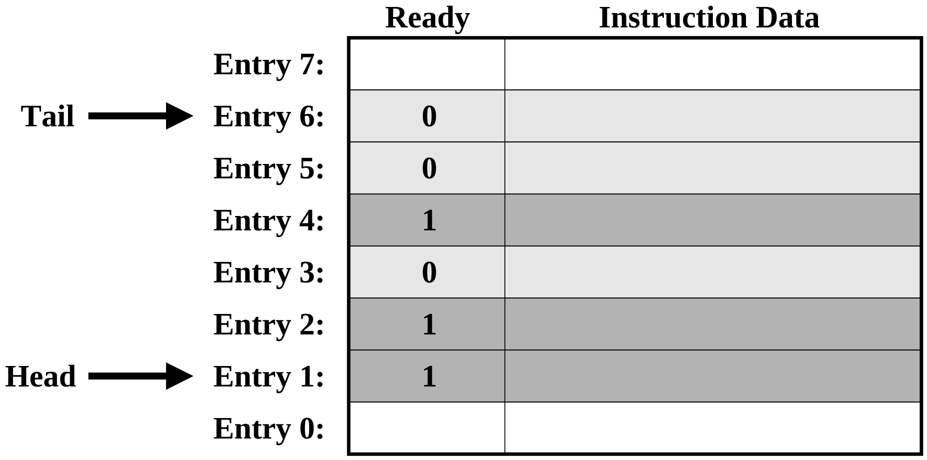
The base microarchitecture requires two changes in order to support out-of-order fetch. The first change is to the fetch buffer and the second is to the decode/issue logic.

For the base microarchitecture, the entries of the fetch buffer form a FIFO. The FIFO is implemented using a circular array. The fetch buffer contains an array of entries, a head pointer, and a tail pointer. The head pointer points to the first entry of the FIFO. The tail pointer points to the last entry of the FIFO. Fetch requests complete in the order that they were initiated; i. e., in program order. When a fetch request completes, its data is inserted into the FIFO. This is accomplished by advancing the tail pointer, and then writing the data into the entry specified by the tail pointer. The decode/issue logic removes the instructions stored in the fetch buffer in program order. If the fetch buffer contains any instructions, the entry specified by the head pointer contains the instructions that need to be removed next. When all the instructions stored in that entry have been removed, the entry is removed from the FIFO. This is accomplished by advancing the head pointer.

For the microarchitecture that supports out-of-order fetch, the fetch buffer does not function as a FIFO. However, the fetch buffer has a structure similar to that of a FIFO. It contains an array of entries, a head pointer, and a tail pointer. These three items are used to implement a circular array. Entries are deallocated in the same order that they are allocated. The head pointer points to the entry that will be deallocated next. The tail pointer points to the entry that was allocated last. Unlike the fetch buffer for the base microarchitecture, the entries from the entry specified by the head pointer to the entry specified by the tail pointer are not guaranteed to contain ready instruction data. Each entry contains an additional bit, called the ready bit, that indicates whether or not that entry has been written; i. e., the bit is set if the entry contains ready instruction data, and reset if it does not.

Fetch requests are initiated in program order. To initiate a request, the request is allocated a fetch buffer entry. This is accomplished by advancing the tail pointer, and then tagging the request with the value of the tail pointer. The ready bit of the entry specified by the tail pointer is also reset to indicate that the entry does not contain ready instruction data. Note that if the above technique is used for allocating fetch buffer entries, when a request completes, it is guaranteed an entry in which to write its data. If the microarchitecture implements non-stalling instruction cache pipelines, as the base microarchitecture and those derived from it do, such a guarantee is required anyway. Unlike the base microarchitecture, requests do not necessarily complete in the same order that they were initiated. When a request completes, it writes its data into the entry indicated by its tag. The ready bit of that entry is set to indicate that the entry contains ready instruction data. The decode/issue logic removes the instructions stored in the fetch buffer in program order. If the fetch buffer contains any instructions, the entry specified by the head pointer contains the instructions that need to be removed next. However, the instructions can only be removed if the entry contains ready instruction data; i. e., if the entry's ready bit is set. If the entry does not contain ready instruction data, the decode/issue logic stalls until the data is ready. When all the instructions stored in the entry specified by the head pointer have been removed, that entry is deallocated. This is accomplished by advancing the head pointer.

Figure 7.15 is a picture of the fetch buffer. The fetch buffer has 8 entries. Each entry has a ready bit and storage for the instruction data. The head pointer points to the entry that will be deallocated next; i. e., entry 1. The tail pointer points to the entry that was allocated last; i. e., entry 6. Entries 1–6 have been allocated. Entries 0 and 7 have not been allocated. Entry 1 was allocated first. The request that was allocated this entry hit in the first level instruction cache and then wrote its data into the entry. Hence, the entry contains ready instruction data, so its ready bit is set. Entry 2 was allocated next. The request that was allocated this entry also hit in the first level instruction cache, wrote its data in the entry, and then set the ready bit of the entry. The request that was allocated entry 3 missed in the first level instruction cache, and has not yet written its data into the entry. Thus, entry 3 does not contain ready instruction data, so its ready bit is reset. When the request that was allocated entry 3 completes, it will write its data into the entry and then set the entry's ready bit to indicate that the entry contains ready instruction data. The request that was allocated entry 4 hit in the first level instruction cache, wrote its data in the entry, and then set the ready bit of the entry. Entries 5 and 6 are the most recently allocated entries. The requests that were allocated these entries have not yet accessed the first level instruction cache. Consequently, the ready bits of these entries are still reset. In this picture, the decode/issue logic can only remove instructions from entries 1 and 2. When the head pointer advances to entry 3, the decode/issue logic will stall until that entry's instruction data becomes ready. After the data becomes ready, the decode/issue logic will remove the instructions from entry 3, and then proceed on with entry 4, entry 5, and finally, entry 6.



		Ready	Instruction Data
	Entry 7:		
Tail	Entry 6:	0	
	Entry 5:	0	
	Entry 4:	1	
	Entry 3:	0	
	Entry 2:	1	
Head	Entry 1:	1	
	Entry 0:		

Figure 7.15: Fetch Buffer for a Machine that Supports Out-of-Order Fetch

The fetch buffer supplies the dynamic instruction stream to the decode/issue logic. The entry specified by the head pointer contains (or, if the instruction data has not yet been written into the entry, will contain) the instructions that appear earliest in the dynamic instruction stream; that is, it contains the instructions that would execute first on a processor that executes instructions in program order. The entry after that one contains the instructions that appear next earliest in the dynamic instruction stream. The farther an entry is from the entry specified by the head pointer, the later the instructions contained in that entry appear in the dynamic instruction stream. The entry specified by the tail pointer contains the instructions that appear latest in the dynamic instruction stream.

The decode/issue logic decodes and issues the instructions of the program in order. To accomplish this, it examines the instructions in the fetch buffer and attempts to create a single issue packet each cycle. An issue packet is a piece of the dynamic instruction stream; i. e., a sequence of logically consecutive instructions. An issue packet can only be created if the next instruction in the dynamic instruction stream that has not been decoded and issued is contained in the fetch buffer. Of all the instructions contained in the fetch buffer, this instruction appears earliest in the dynamic instruction stream. If an issue packet is created, its first instruction will be this instruction. That is, its first instruction will be the first instruction that is contained in the fetch buffer entry specified by the head pointer. Once an issue packet has been created, the instructions that comprise that issue packet are removed from the fetch buffer. The issue packet is then decoded and issued.

For the base microarchitecture, every fetch buffer entry from the entry specified by the head pointer to the entry specified by the tail pointer is guaranteed to contain ready instruction data. Therefore, the instructions from all entries between the entry specified by the head pointer and the entry specified by the tail pointer, inclusive, can be used to create an issue packet.

For the microarchitecture that supports out-of-order fetch, the entries from the entry specified by the head pointer to the entry specified by the tail pointer are not guaranteed to contain ready instruction data. Consequently, the decode/issue logic must be modified to prevent it from creating issue packets that contain instructions from fetch buffer entries that do not contain ready instruction data. The decode/issue logic scans the fetch buffer entries, starting with the entry specified by the head pointer and proceeding towards the entry specified by the tail pointer, to determine which entry is the last entry that contains ready instruction data. That is, it scans the entries to determine which entry is the last entry with a ready bit that is set. The instructions from all entries between the entry specified by the head pointer and this last entry, inclusive, can be used to create an issue packet. As an example, in Figure 7.15, only the instructions from fetch buffer entries 1 and 2 can be used to create an issue packet.

7.3 Apparatus for Out-of-Order Fetch/Decode/Issue

The apparatus for out-of-order fetch/decode/issue builds upon the apparatus for out-of-order fetch. This section describes the remaining apparatus that a processor needs to implement out-of-order fetch/decode/issue. Subsection 7.3.1 describes *sequence numbers*, which the processor uses to keep track of the logical order of the instructions that are resident in the machine. Subsection 7.3.2 describes modifications to the fetch buffer. Subsection 7.3.3 describes how the processor handles register dependencies. Register dependencies require special consideration for a processor with out-of-order fetch/decode/issue. Memory dependencies do not. Subsection 7.3.4 describes the modifications to the instruction scheduling logic required by a processor that implements out-of-order fetch/decode/issue using the assume dependence dependency handling technique. Subsection 7.3.5 describes physical register withholding and checkpoint withholding, which are needed to prevent deadlock. Finally, subsection 7.3.6 describes how the processor handles serializing instructions.

7.3.1 Sequence Numbers

A processor with out-of-order fetch/decode/issue assigns a *sequence number* to every instruction it fetches. The sequence number specifies the instruction's position within the dynamic instruction stream; that is, it specifies the instruction's age. The first instruction in the dynamic instruction stream, which is the oldest instruction in the dynamic instruction stream, is assigned the first sequence number: sequence number 1. The second instruction in the dynamic instruction stream, which is the second oldest instruction in the dynamic instruction stream, is assigned the second sequence number: sequence number 2. And so on. The processor uses these sequence numbers to keep track of the logical order of the instructions that are resident in the machine.

The processor actually supports only a finite number of sequence numbers, rather than an infinite number of sequence numbers. The sequence numbers are recycled in order to make it appear as though there are an infinite number of them, just as the ticket numbers of the take-a-number systems used by butcher shops are recycled in order to make it appear as though there are an infinite number of ticket numbers. To guarantee that a butcher shop does not run out of ticket numbers, the number of ticket numbers should be greater than or equal to the maximum number of customers that can potentially be waiting for service. To guarantee that the processor cannot run out of sequence numbers, the number of supported sequence numbers should be greater than or equal to the maximum number of instructions that can be resident in the machine at one time.

Two counters, the *head* and the *tail*, define the set of sequence numbers that are assigned to the instructions that are resident in the machine. The head specifies the sequence number of the oldest instruction that is resident in the machine. Whenever an instruction is retired, the head is incremented. The tail specifies the sequence number of the youngest instruction that is resident in the machine. Whenever an instruction is fetched, the tail is incremented. The resulting value of the tail counter is the sequence number that is assigned to the fetched instruction. When either counter is incremented past its maximum value, which is N for a processor that supports N sequence numbers, the value of the counter wraps around to 1.

The sequence numbers are used to implement the oldest first scheduling heuristic, to flush instructions from the microengine, and to keep track of the information required for out-of-order fetch/decode/issue.

To implement the oldest first scheduling heuristic, the sequence number that is assigned to the first instruction in an issue packet specifies the age of that packet. The age of a packet (i. e., the sequence number assigned to the first instruction in the packet) is stored in an Issue Packet Age Register (IPAR), as shown in Figure 7.16. There is one IPAR for each of the 63 checkpoints that are used to store speculative state. When a packet is issued, its age is written into the IPAR associated with the checkpoint allocated to the packet. As it is written, it is compared to the ages—which are stored in the other IPARs—of all other packets stored in the Node Tables. The results of these comparisons classify packets into two groups: those older than the newly issued packet, and those younger than the newly issued packet. As described at the very end of Section 7.1.2, these results are used to initialize the newly issued packet's priority information register, and to update the priority information registers of the other packets already stored in the Node Tables. The priority information registers are used to implement the oldest first scheduling heuristic.

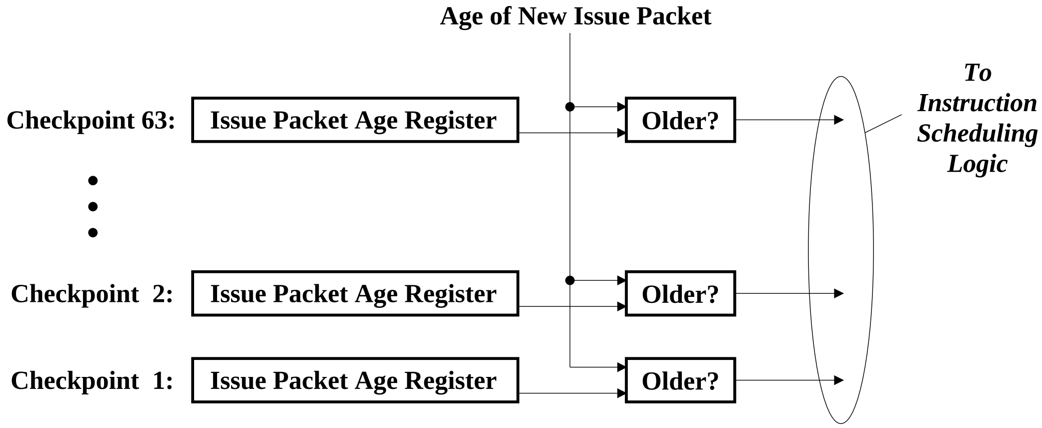

Age of New Issue Packet
Checkpoint 63:
Issue Packet Age Register
Older?
To Instruction Scheduling Logic
Checkpoint 2:
Issue Packet Age Register
Older?
Checkpoint 1:
Issue Packet Age Register
Older?

Figure 7.16: Issue Packet Age Registers

Whenever a mispredicted branch is resolved, instructions may need to be flushed from the machine. Instructions are always flushed from the back-end of the machine (i. e., the execution core and the load/store system) in units of issue packets. If a mispredicted branch is the last (i. e., youngest) instruction in its packet, only the packets that are younger than the packet containing the branch must be flushed. If a mispredicted branch is not the last instruction, the packet that contains the branch must also be flushed. To flush these packets, the age of the branch (i. e., the sequence number assigned to the branch) is compared to the ages—which are stored in the IPARs—of all the packets stored in the Node Tables. All checkpoints allocated to packets that are younger than the branch, as well as the physical registers associated with those checkpoints, are deallocated. If the branch is not the last instruction in its packet, the checkpoint allocated to the packet containing the branch and the checkpoint's associated physical registers are also deallocated. The machine must also reclaim all sequence numbers assigned to the flushed instructions. To accomplish this, the tail counter is simply rolled back. If the mispredicted branch is the last instruction in its packet, the tail counter is set to the sequence number assigned to the branch. If the mispredicted branch is not the last instruction in its packet, the tail counter is set to one less than the sequence number assigned to the first instruction in the packet that contains the branch.

When a mispredicted branch resolves, all instructions that are younger than the branch must be flushed from the front-end of the machine (i. e., the fetch unit).

For the base microarchitecture and the microarchitecture that supports out-of-order fetch, instructions are always issued in program order. When a mispredicted branch resolves, all instructions that are older than the branch have been issued, and are no longer in the front-end of the machine. Any instructions in the front-end are younger than the branch, and must be flushed. These microarchitectures simply flush all instructions from the front-end whenever a mispredicted branch resolves.

For a microarchitecture that implements out-of-order fetch/decode/issue using the assume dependence dependency handling technique, a post-hole branch instruction is not allowed to execute until the hole disappears. When a mispredicted branch resolves, it can never be a post-hole instruction. All instructions that are older than the branch have been issued, and are no longer in the front-end of the machine. Any instructions in the front-end are younger than the branch, and must be flushed. Hence, this microarchitecture also flushes all instructions from the front-end whenever a mispredicted branch resolves.

For a microarchitecture that implements out-of-order fetch/decode/issue using the assume independence dependency handling technique, if a mispredicted branch resolves, and that branch is a post-hole instruction, the front-end contains instructions that are older than the branch. The fetch unit must only flush the instructions that are younger than the branch from the front-end of the machine. The instructions that are older than the branch must not be flushed. Designing a front-end that can perform such selective flushing is nontrivial.

To simplify the design of such a microarchitecture, the following policy may be used: branch instructions use the assume dependence dependency handling technique, and all other instructions use the assume independence dependency handling technique. Since branch instructions assume dependence, when a mispredicted branch is resolved, it can never be a post-hole instruction. All instructions that are older than the branch will have been issued, and will no longer be in the front-end of the machine. Any instructions in the front-end will be younger than the branch, and need to be flushed. With this policy, the microarchitecture does not need a front-end that can be selectively flushed, so it is simpler to design. In Chapter 8, I will show that a microarchitecture that uses this policy outperforms the like microarchitecture that does not.

Finally, the processor uses the sequence numbers to keep track of information required for out-of-order fetch/decode/issue. This information is stored in the Sequence Number Information Table (SNIT), as shown in Figure 7.17. There is one entry in the table for each sequence number. Each entry contains a decode bit and a 6 bit checkpoint field. The decode bit is set if the instruction that was assigned the given sequence number has been removed from the fetch buffer; i. e., if the instruction has already passed through the first stage of the instruction decode pipeline. When the decode bit is set, the checkpoint field specifies the checkpoint that was allocated for the issue packet that contains that instruction. Note that the base microarchitecture and the microarchitectures derived from it support 64 checkpoints, so a 6 (i. e., $\lceil \log_2 64 \rceil$) bit field is required to specify the checkpoint number. The decode bit is reset if the instruction that was assigned the given sequence number has not yet been inserted into the fetch buffer, or if the instruction has been inserted into the fetch buffer but it has not yet been removed from the fetch buffer. When the decode bit is reset, the checkpoint field is not used.

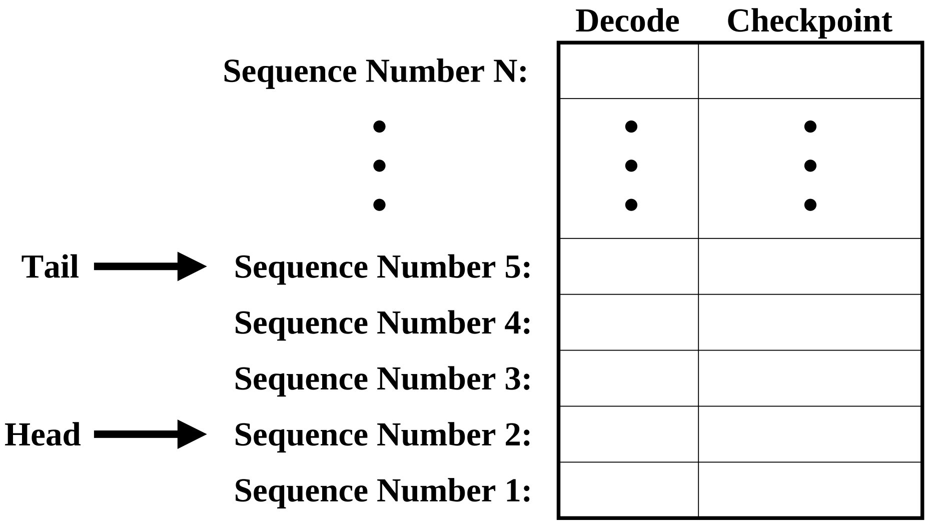


Figure 7.17: Sequence Number Information Table

When an instruction is fetched, the tail is incremented. The resulting value of the tail counter is the sequence number assigned to the fetched instruction. The sequence number is used as an index to access the SNIT. The decode bit of the selected SNIT entry is reset to indicate that the instruction assigned that sequence number has not yet passed through the first stage of the instruction decode pipeline.

Later, when the instruction is removed from the fetch buffer and has passed through the first stage of the instruction decode pipeline, the sequence number assigned to the instruction is used again as an index to access the SNIT. The decode bit of the selected SNIT entry is set to indicate that the instruction has passed through the first stage of the instruction decode pipeline. Also, the number of the checkpoint that was allocated for the issue packet containing the instruction is written into the checkpoint field of the selected SNIT entry.

The head counter specifies the sequence number of the oldest instruction that is resident in the machine. Since instructions are always retired in program order, the oldest instruction is the next instruction to be retired. The instruction retirement logic uses the sequence number specified by the head counter to access the SNIT. The selected SNIT entry contains the information regarding the oldest instruction. If the decode bit of that entry is set, the instruction has passed through the first stage of the instruction decode pipeline, and the checkpoint field of the entry specifies the number of the checkpoint that was allocated for the issue packet containing the instruction. The retirement logic examines that checkpoint, and, if all the instructions in the packet assigned to the checkpoint have completed without generating exceptions, and if the predicted targets of all the branches in the packet have been verified, the packet is retired. For each instruction that is retired, the retirement logic increments the head counter by one. If the decode bit of the selected SNIT entry is reset, either the oldest instruction has not yet been inserted into the fetch buffer, or, if the oldest instruction has been inserted into the fetch buffer, it has not yet been removed from the fetch buffer. The retirement logic waits until the instruction has passed through the first stage of the instruction decode pipeline (i. e., it waits until the decode bit is set) before trying to retire the instruction.

7.3.2 Modified Fetch Buffer

Like a processor with out-of-order fetch, a processor with out-of-order fetch/decode/issue initiates fetch requests in program order but allows them to complete out-of-order. The branch predictor produces, in program order, a sequence of requests for the instructions that comprise the dynamic instruction stream. Each request is allocated a fetch buffer entry, and then issued to the instruction cache, where it initiates an instruction cache fetch. As a result of cache misses, the requests may complete (i. e., obtain their data) out-of-order. As they complete, they write their data into the fetch buffer entries allocated to them.

Unlike a processor with out-of-order fetch, a processor with out-of-order fetch/decode/issue can decode and issue instructions out of program order. To enable out-of-order fetch/decode/issue, the processor allows issue packets to be created out of program order. Each cycle, the decode/issue logic examines the instructions in the fetch buffer and attempts to create a single packet. A packet is created so long as there are instructions in fetch buffer entries whose associated fetch requests have completed. Any packet created will contain the oldest such instruction in the fetch buffer. A packet is a sequence of logically consecutive instructions. The sequence numbers assigned to the instructions in the fetch buffer are used to determine which of those instructions are in sequence and can therefore be placed in the same packet. Once a packet is created, the instructions that comprise the packet are removed from the fetch buffer. When all the instructions in a fetch buffer entry have been removed, that entry is deallocated. Since packets may be created out-of-order, instructions may be removed from the fetch buffer out-of-order. Consequently, fetch buffer entries, which are allocated to fetch requests in the order that those requests are produced, may not be deallocated in that same order.

Figure 7.18 is a picture of the fetch buffer for the microarchitecture that supports out-of-order fetch/decode/issue. The fetch buffer contains two structures: a list and a storage array. An entry in the fetch buffer consists of an entry in the list and an entry in the storage array. A pointer links each list entry with the storage array entry that belongs to the same fetch buffer entry as the list entry. The list specifies the order of the fetch buffer entries. The first entry in the list belongs to the fetch buffer entry allocated for the oldest set of instructions that are stored (or will be stored) in the fetch buffer. The second entry in the list belongs to the fetch buffer entry allocated for the second oldest set of instructions in the fetch buffer. And so on. The storage array is used to store the instruction data associated with each fetch buffer entry.

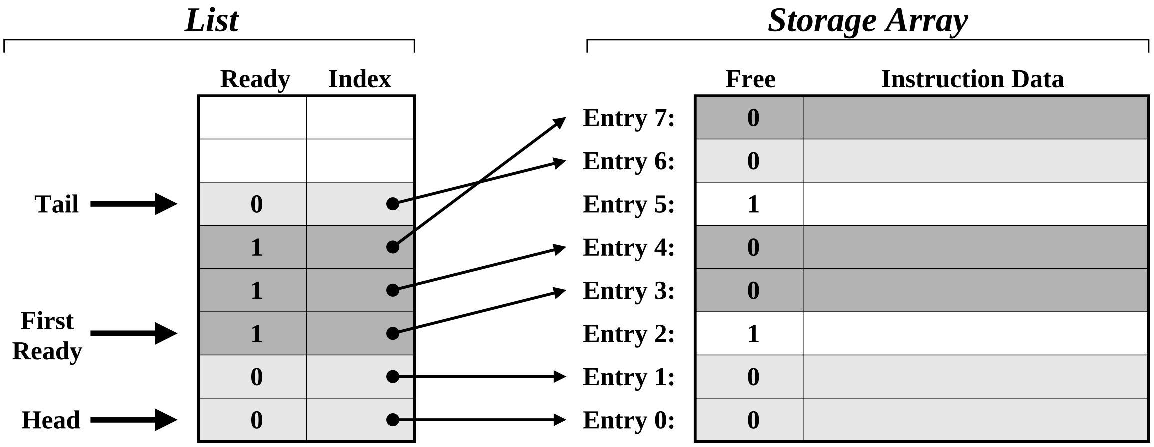


Figure 7.18: Fetch Buffer for a Machine that supports Out-of-Order Fetch/Decode/Issue

Each list entry contains a ready bit and a 3 bit index field. The index field specifies a pointer to a storage array entry. The storage array entry contains the instruction data for the fetch buffer entry associated with the given list entry. The ready bit indicates whether or not the associated fetch buffer entry has been written. The bit is set if the fetch buffer entry contains ready instruction data. To be more specific, the bit is set if the storage array entry that belongs to the fetch buffer entry contains ready instruction data. The bit is reset if it does not contain ready instruction data.

Each storage array entry contains a free bit in addition to the field(s) required to store the instruction data. This bit is set if the storage array entry belongs to a free fetch buffer entry; i. e., a fetch buffer entry that is not allocated to a fetch request. It is reset if the storage array entry belongs to a fetch buffer entry that is allocated to a fetch request.

The list is implemented with an array. The first entry in the array, which is obtained by accessing the array with index 0, is the first item in the list. The second entry in the array, which is obtained by accessing the array with index 1, is the second item in the list. And so on. A tail pointer provides the index of the array entry that contains the last item in the list. Although the figure shows a head pointer, the head pointer is not actually needed, since the index of the array entry that contains the first item in the list is always 0. To add an item to the list, the tail pointer is incremented by 1. The data for the new item is then written into the entry pointed to by the tail pointer. To remove the item from the list that is stored at an array entry whose index is *N*, each item stored at an array entry whose index is *M*, where *M* is greater than *N*, is shifted (copied) into the entry whose index is *M*-1. The tail pointer is then decremented by 1.

To initiate a fetch request, the request is allocated a fetch buffer entry. This is accomplished by scanning the storage array to find a storage array entry that is free; i. e., to find a storage array entry whose free bit is set. The free bit of this entry is reset to indicate that the entry has been allocated. An entry for the fetch buffer entry is then added to the list. The ready bit of the list entry is reset to indicate that the fetch buffer entry does not contain ready instruction data. The index field of the list entry is set equal to the index of the storage array entry. The fetch request is then tagged with the index of the storage array entry. When a request completes, it writes its data into the storage array entry indicated by its tag. The list is scanned (via CAMs) to find the list entry whose index field matches the tag. The ready bit of that list entry is set to indicate that the associated fetch buffer entry contains ready instruction data.

The *first ready* pointer points to the first entry in the list whose ready bit is set. The fetch buffer entry associated with this list entry contains the oldest set of instructions in the fetch buffer whose associated fetch request has completed. The decode/issue logic examines the instructions in the fetch buffer; starting with the fetch buffer entry indicated by the first ready pointer, proceeding towards the fetch buffer entry indicated by the tail pointer, and ending with the first fetch buffer entry that does not contain ready instruction data; and attempts to create a single issue packet each cycle. A packet is created if there is at least one fetch buffer entry that contains ready instruction data. Any packet created will contain the oldest instruction from the fetch buffer entry indicated by the first ready pointer. The sequence numbers assigned to the examined instructions are used to determine which of them are in sequence and can therefore be placed in the same packet. Once a packet is created, the instructions that comprise the packet are removed from the fetch buffer. When all the instructions in a fetch buffer entry have been removed, that entry is deallocated. To deallocate a fetch buffer entry, its associated list entry is removed from the list, and the free bit of its associated storage array entry is set to indicate that the entry is available.

The decode/issue logic uses the value of the first ready pointer to determine whether or not it creates an issue packet that consists of post-hole instructions. If the value of the pointer is 0, it points to the first entry in the list. The fetch buffer entry associated with this list entry contains the oldest set of instructions in the fetch buffer that have not been decoded and issued. Any packet created will contain the oldest of these instructions, which is not a post-hole instruction. Hence, when the value of the pointer is 0, any packet created will not consist of post-hole instructions. On the other hand, if the value of the pointer is not 0, it does not point to the first entry in the list. The oldest instruction in the fetch buffer that has not been decoded and issued will not be included in the issue packet if one is created. Thus, when the value of the pointer is not 0, any packet created will consist of post-hole instructions.

If the decode/issue logic creates an issue packet that consists of post-hole instructions, it can use the value of the first ready pointer to estimate the total number of hole instructions. The value of the first ready pointer is equal to the number of fetch buffer entries that contain the hole instructions. To get an estimate of the total number of hole instructions, the decode/issue logic can multiply the value of the first ready pointer by an estimate of the average number of instructions that are stored in a fetch buffer entry. The decode/issue logic can also obtain an exact count of the total number of hole instructions. Starting with the fetch buffer entry associated with the first entry in the list, and ending with the fetch buffer entry associated with the list entry whose index is one less than the value of the first ready pointer, the decode/issue logic can add up the number of instructions that will be written into each of those fetch buffer entries. The sum is an exact count of the total number of hole instructions.

7.3.3 Handling of Register Dependencies

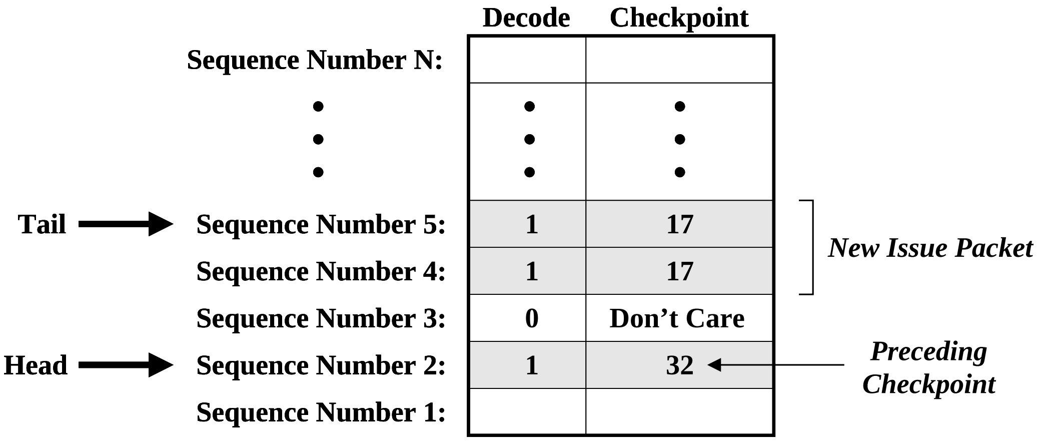
The Register Alias Table (RAT) behaves like a checkpointed register file. It contains a set of physical registers, and, for each of the checkpoints that the machine supports, an array—or map—that maps the architectural registers onto the physical registers. Indexing the array with the number of an architectural register yields the number of the physical register mapped to the architectural register for the given checkpoint. The map for a given checkpoint records the architectural-to-physical register mapping for the point just after the last instruction in the issue packet assigned to that checkpoint has been decoded.

Before an issue packet is issued into the machine, it is allocated a checkpoint. The RAT map for that checkpoint is then initialized. To initialize the map, the decode/issue logic determines which of the packets stored in the Node Tables is the youngest packet that still logically precedes (i. e., is still older than) the packet being issued, and which checkpoint is assigned to that packet. The map for that checkpoint is then copied to the map for the newly allocated checkpoint.

To determine which checkpoint contains the youngest issue packet that is still older than the packet being issued, the entries of the Sequence Number Information Table (SNIT) are scanned. The first entry scanned contains the information regarding the instruction that is one instruction older than the first (youngest) instruction in the packet being issued. The index of this entry is equal to one less than the sequence number assigned to the first instruction in the packet. The second entry scanned contains the information regarding the instruction that is two instruction older than the first instruction in the packet. The index of this entry is equal to two less than the sequence number assigned to the first instruction in the packet. Scanning proceeds in this way towards the entry that contains the information regarding the oldest instruction in the machine; i. e., towards the entry whose index is equal to the head counter for the sequence numbers. Scanning stops when an entry is found whose decode bit is set. This entry contains the information regarding the youngest instruction that has passed through the first stage of the instruction decode pipeline and that is still older than the instructions in the packet being issued. The checkpoint field of this entry specifies the number of the checkpoint that was allocated for the issue packet containing this instruction; i. e., the number of the checkpoint that contains the youngest issue packet that is still older than the packet being issued.

If the packet being issued does not consist of post-hole instructions, determining which checkpoint contains the youngest packet that is still older than the packet being issued is trivial, since a scan of the SNIT entries is not actually needed. Because the packet does not consist of post-hole instructions, the instruction that is one instruction older than the first instruction in that packet has already been issued; i. e., it has already passed through the first stage of the instruction decode pipeline. As a result, the decode bit of its SNIT entry will be set. This SNIT entry is the first entry scanned, and scanning stops when an entry is found whose decode bit is set. If there were a scan of the SNIT entries, the scan would always stop at this entry. Instead of a scan, the decode/issue logic can simply read out this first entry.

On the other hand, if the packet being issued does consist of post-hole instructions, determining which checkpoint contains the youngest packet that is still older than the packet being issued is not trivial, since a scan of the SNIT entries is needed. Figure 7.19 shows an example of this situation. A new issue packet, consisting of two post-hole instructions whose sequence numbers are 4 and 5, is being issued into the machine. The issue packet has passed through the first stage of the instruction decode pipeline and has been allocated checkpoint number 17. The SNIT entries for the instructions in this packet have already been updated to reflect these facts. The instruction whose sequence number is 3 is a hole instruction. It has not been issued, nor has it passed through the first stage of the instruction decode pipeline. The instruction whose sequence number is 2 is a pre-hole instruction. It has passed through the first stage of the instruction decode pipeline, it has been issued, and it has been assigned checkpoint number 32.



	Decode	Checkpoint
Sequence Number N:		
• • •	• • •	• • •
Sequence Number 5:	1	17
Sequence Number 4:	1	17
Sequence Number 3:	0	Don't Care
Sequence Number 2:	1	32
Sequence Number 1:		

Figure 7.19: Figure showing how the Sequence Number Information Table is used to determine which checkpoint contains the issue packet that logically precedes a new issue packet

The scan starts with the SNIT entry associated with sequence number 3. This entry contains the information regarding the hole instruction, which is one instruction older than the first instruction in the new issue packet; i. e., the instruction whose sequence number is 4. Note that the hole instruction's sequence number is one less than the sequence number of the first instruction in the new issue packet. The scan stops at the SNIT entry associated with sequence number 2, which is the first entry scanned whose decode bit is set, and which contains the information regarding the pre-hole instruction. The checkpoint field of this entry provides the number of the checkpoint that contains the youngest issue packet that is still older than the packet being issued; i. e., checkpoint number 32.

After the RAT map for the checkpoint allocated to an issue packet is initialized, the decode/issue logic performs operand fetch for each instruction in that packet. Each source register accesses the RAT to obtain its tag, and, if available, its data. The tag is obtained by using the architectural register number of the source register as an index to access the map associated with the checkpoint. The returned value is the address of the physical register that contains (or will contain) the data for the requested source register. This value serves as the tag. Each destination register accesses the RAT to allocate a new physical register. The address of this physical register is then used to update the map associated with the checkpoint. This is accomplished by writing the address of the physical register into the map entry whose index is equal to the architectural register number of the destination register.

For machines that support out-of-order fetch/decode/issue, issue packets may be decoded out of program order. When a packet consisting of post-hole instructions is decoded, the decode/issue logic may not know for certain which instructions the post-hole instructions are dependent on. The decode/issue logic assumes that the post-hole instructions are not dependent on hole instructions. All register dependencies are assumed to be between post-hole instructions and other post-hole instructions, or between post-hole instructions and pre-hole instructions. Register renaming is performed as though the hole instructions do not exist.

Later, when the hole instructions are decoded, the register dependencies of the post-hole instructions can be correctly computed, and, if necessary repaired. Some of the post-hole instructions may be dependent on the hole instructions. Suppose that a post-hole instruction is only dependent on a single hole instruction. When the post-hole instruction was decoded, it picked up the wrong tag for the source register operand that depends on the hole instruction. Rather than picking up the tag for the value produced by the hole instruction, it picked up the tag for a value produced by a pre-hole instruction.

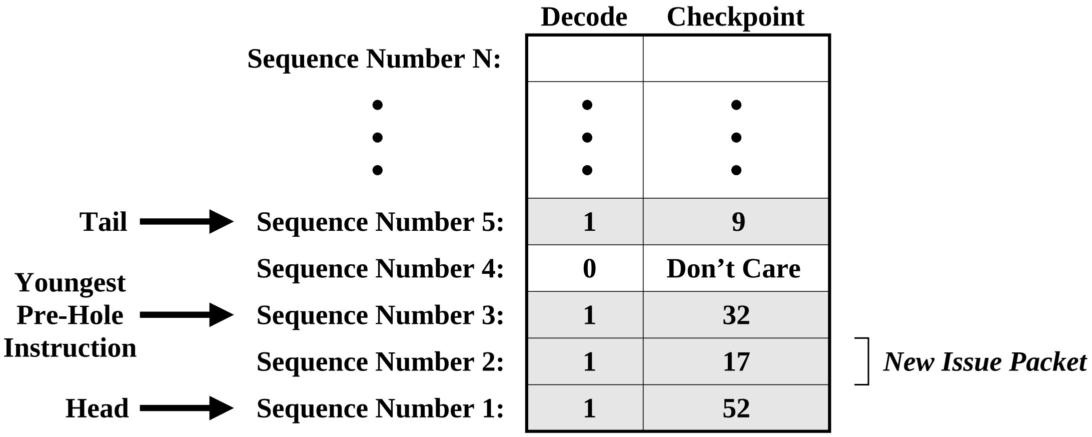
To fix this, during operand fetch, for each instruction in an issue packet, the instruction's destination register accesses the RAT to obtain the tag for the *previous instance* of that architectural register. As before, the destination register also accesses the RAT to allocate a new physical register, to obtain the tag associated with that physical register—that is, to obtain the tag for the *current instance* of the architectural register—and to update the checkpointed map. All instructions that are younger than the given instruction can only be dependent on the current instance. They cannot be dependent on the previous instance. If they have source register operands whose tags are equal to the tag for the previous instance, those operands have the wrong tags. The tags for those operands must be re-written with the correct tag value; i. e., the value of the tag for the current instance.

The tags for the operands are re-written using a technique that is similar to the technique used to re-write the whence fields. (See the very end of Section 7.1.3 for a description of the technique used to re-write the whence fields.) When a new packet is issued, each instruction in the packet broadcasts the tag for the previous instance to the instructions stored in the Node Tables. Each instruction also broadcasts the tag for the current instance, as well as the number of the functional unit whose tag bus and result bus will be used to distribute the instruction's tag and data value. As the packet is issued, its age is compared to the ages—which are stored in the Issue Packet Age Registers (IPARs)—of the other packets stored in the Node Tables. For all instructions in packets that are younger than the packet being issued, and that have source register operands whose tags match one of the tags for a previous instance, the matching tags are overwritten with the correct rename tags; i. e., the tags for the current instances. In addition, the whence fields for the operands with matching tags are overwritten with the correct functional unit numbers; i. e., the functional unit numbers that were broadcast along with the tags for the current instances.

7.3.4 Modified Instruction Scheduling Logic

For processors that implement out-of-order fetch/decode/issue using the assume dependence dependency handling technique, post-hole instructions are not allowed to execute until the hole disappears. Only pre-hole instructions are allowed to execute. The instruction scheduling logic must be modified so that only pre-hole instructions are allowed to schedule for execution. To be more precise, the instruction scheduling logic must be modified so that only the instructions that are older than (or as old as) the youngest pre-hole instruction are allowed to schedule for execution.

The Sequence Number Information Table (SNIT) is used to keep track of the sequence number of the youngest pre-hole instruction. Figure 7.19 shows an example SNIT. To determine the sequence number of the youngest pre-hole instruction, the entries of the SNIT are scanned, starting with the entry indicated by the head counter, proceeding towards the entry indicated by the tail counter, and ending with the first entry whose decode bit is reset. Note that the entries are scanned in the order of oldest to youngest, where the oldest entry is defined as the entry that contains the information regarding the oldest instruction in the machine, and the youngest entry is defined as the entry that contains the information regarding the youngest instruction in the machine. The sequence number of the youngest pre-hole instruction is equal to one less than the sequence number associated with the entry that ended the scan. This sequence number is 3 in the figure.



	Decode	Checkpoint
Sequence Number N:		
• • •	• • •	• • •
Sequence Number 5:	1	9
Sequence Number 4:	0	Don’t Care
Sequence Number 3:	1	32
Sequence Number 2:	1	17
Sequence Number 1:	1	52

Figure 7.20: Figure showing how the Sequence Number Information Table is used to determine the sequence number of the youngest pre-hole instruction

Before a new issue packet is issued into the machine, the decode/issue logic calculates what the sequence number of the youngest pre-hole instruction will be after that packet has been issued. In the figure, the new issue packet consists of a single instruction whose sequence number is 2. This instruction is a hole instruction, since an instruction that is younger than it (i. e., the instruction whose sequence number is 3) has already passed through the first stage of the instruction decode pipeline. After this new packet has issued, the sequence number of the youngest pre-hole instruction will be 3.

When the new issue packet is issued into the machine, the age of the youngest pre-hole instruction (i. e., the sequence number of the youngest pre-hole instruction calculated as described in the preceding paragraph) is compared to the ages—which are stored in the Issue Packet Age Registers (IPARs)—of all the packets stored in the Node Tables. Each checkpoint allocated to a packet that is older than (or as old as) the youngest pre-hole instruction allows its instructions to be scheduled. Each checkpoint allocated to a packet that is younger than the youngest pre-hole instruction inhibits its instructions from being scheduled. As a result of this, only pre-hole instructions are allowed to schedule and execute, and post-hole instructions are not allowed to schedule and execute until the hole disappears.

7.3.5 Physical Register and Checkpoint Withholding

Physical register withholding and checkpoint withholding are needed to prevent deadlock in machines equipped with out-of-order fetch/decode/issue. When a post-hole instruction is issued, the machine must guarantee that all hole instructions that logically precede that post-hole instruction will have the physical registers and checkpoints they need to be issued into the machine. If this guarantee is not met, the machine will deadlock: the post-hole instruction will not retire—and hence, release the physical register and checkpoint allocated to it—until the hole instructions have retired, yet the hole instructions cannot be issued because they don't have the required physical registers and checkpoints.

To guarantee that all hole instructions that logically precede the post-hole instruction will have the physical registers they need to be issued into the machine, the machine withholds one physical register per hole instruction whenever it issues a post-hole instruction. Every instruction requires exactly one physical register when it is issued, so the machine can compute the exact number of physical registers that will be needed for the hole instructions. As described in Section 7.3.2, the fetch buffer's first ready pointer can be used to either estimate or compute the total number of hole instructions.

To guarantee that all hole instructions that logically precede the post-hole instruction will have the checkpoints they need to be issued into the machine, the machine withholds some of its checkpoints for the hole instructions whenever it issues a post-hole instruction. The machine knows how many hole instructions there are. Unfortunately, it does not know how many checkpoints will be needed for these hole instructions, since the number of instructions that can fit on a checkpoint varies.

Fortunately, as long as the machine withholds at least one checkpoint for the hole instructions, deadlock is prevented. Whenever there are two or more free (unallocated) checkpoints, the machine may use those checkpoints to issue whichever instructions it chooses. Whenever there is only one free checkpoint, that checkpoint is withheld for the hole instructions, so the machine may not use that checkpoint to issue post-hole instructions. The machine uses that remaining free checkpoint to try to issue, in program order, all the hole instructions, starting with the hole instruction that appears the earliest in the dynamic instruction stream, and working towards the hole instruction that appears the latest. If the number of hole instructions is greater than the number of instructions that can fit on a checkpoint, the machine will need to use the remaining free checkpoint multiple times. Each time, it will assign an issue packet's worth of hole instructions to the checkpoint, issue those instructions, and then wait for them to retire. After those instructions have retired, it will use the checkpoint to issue a new group of instructions. Essentially, whenever the machine has only one free checkpoint, it operates as an in-order issue machine that supports only a single speculative checkpoint. Hence, even though a machine can prevent deadlock by withholding only one checkpoint for all of the hole instructions, this technique for preventing deadlock may result in poor performance.

Another technique for preventing deadlock is to withhold a set fraction of a checkpoint for each hole instruction. To calculate the number of checkpoints that need to be withheld, the number of hole instructions is multiplied by this fraction, and then the result is rounded up to the nearest integer. For example, if on average 8 instructions fit on a checkpoint, a machine might want to withhold 1/8 of a checkpoint per hole instruction. If there are 17 hole instructions, the machine would withhold 3 (i. e., $\lceil 17 \times 1/8 \rceil$) checkpoints. The fraction is selected such that the multiplication is easily performed in hardware; i. e., the fraction is $\frac{1}{2^n}$ for n $\in \{0, 1, \dots\}$. Rounding the result up to the nearest integer ensures that at least one checkpoint is withheld whenever there are any hole instructions. If this technique underestimates the number of checkpoints that need to be withheld, deadlock is prevented via the technique described in the preceding paragraph.

For the base microarchitecture and those derived from it, 16 instructions can fit on a checkpoint. One of these machines might optimistically assume that the hole instructions will be packed 16 per checkpoint. This machine would withhold 1/16 of a checkpoint for each hole instruction. A machine might also assume the average, as was done in the example. Or, a machine might pessimistically assume that the hole instructions will only be packed 1 per checkpoint, and therefore withhold a whole checkpoint for each hole instruction.

If the fraction of a checkpoint withheld per hole instruction is too low, the machine underestimates the number of checkpoints that need to be withheld, and must rely on the first technique to prevent deadlock. The first technique may result in poor performance. Consequently, if the fraction is too low, the performance may suffer. On the other hand, if the fraction is too high, the machine overestimates the number of checkpoints that need to be withheld. The machine may not be able to issue a packet of post-hole instructions because all of the free (unallocated) checkpoints have been withheld. Thus, if the fraction is too high, the performance may suffer.

For the experiments in the next chapter, all machines that support out-of-order fetch/decode/issue will use the first checkpoint withholding technique for preventing deadlock unless otherwise indicated. That is, if there are hole instructions, they withhold 1 checkpoint—and only 1 checkpoint—regardless of the number of hole instructions. Experimental results presented in the next chapter will show that the performance of a machine does not depend on which checkpoint withholding technique is used.

7.3.6 Handling of Serializing Instructions

A serializing instruction is an instruction that, from the standpoint of the instruction set architecture, must be executed in order with respect to the other instructions in the program's dynamic instruction stream. A microarchitecture can speculatively execute serializing instructions. Indeed, all of today's high performance microarchitectures do. When a microarchitecture retires a serializing instruction, however, that serializing instruction must have executed in order with respect to the other instructions.

For microarchitectures that decode and issue instructions in program order (i. e., the base microarchitecture and the microarchitecture that supports out-of-order fetch), each serializing instruction is placed in its own issue packet with no other instructions. A packet containing a serializing instruction is not allowed to issue until all logically preceding (older) instructions have retired. Once the packet containing the serializing instruction is issued, new packets are not allowed to issue until the serializing instruction has retired.

For the most part, microarchitectures that decode and issue instructions out-of-order (i. e, the microarchitecture that supports out-of-order fetch/decode/issue) handle serializing instructions in the same way as microarchitectures that do not decode and issue instructions out-of-order. However, for microarchitectures that decode and issue instructions out-of-order, there are a couple of special considerations regarding serializing instructions.

First, a serializing instruction is not allowed to issue as a post-hole instruction, since the hole instructions—which are older than it—must execute first. The decode/issue logic inhibits the creation of an issue packet if that packet would contain a serializing instruction that is a post-hole instruction. As described in Section 7.3.2, it uses the fetch buffer's first ready pointer to determine whether or not an instruction is a post-hole instruction. A serializing instruction is only allowed to issue after all older instructions (i. e., the hole instructions) have issued.

Second, the machine may issue packets containing post-hole instructions, and then, when creating the packets of hole instructions, discover that one of the hole instructions is a serializing instruction. From the standpoint of the instruction set architecture, the post-hole instructions are not allowed to execute until after the serializing instruction has executed. To handle this problem, all instructions younger than the serializing instruction—that is, the post-hole instructions—are flushed from the machine and then re-fetched. When all instructions older than the serializing instruction have retired, the packet containing the serializing instruction is issued. After the serializing instruction retires, the instructions younger than it, including the recently flushed post-hole instructions, are issued (or re-issued) into the machine.

CHAPTER 8

Out-of-Order Fetch, Decode, and Issue: Final Results

This chapter uses the more realistic machine model presented in Chapter 7, which is modeled with the full simulator, to evaluate the actual performance benefit of out-of-order fetch, decode, and issue. It also examines various implementation tradeoffs that could not be evaluated with the abstract machine model used in Chapter 6, which was modeled with the RDF simulator.

In particular, with the abstract machine model, only the performance benefit of out-of-order fetch/decode/issue could be calculated. The performance benefit of out-of-order fetch could not be calculated. With the more realistic machine model, the performance benefits of both out-of-order fetch and out-of-order fetch/decode/issue can be—and are—calculated. The results show that out-of-order fetch is almost as effective as out-of-order fetch/decode/issue in eliminating the performance penalty that results from instruction cache misses.

Additionally, with the abstract machine model, a machine implementing out-of-order fetch/decode/issue using the assume independence dependency handling technique could not be modeled. With the more realistic machine model, a machine implementing out-of-order fetch/decode/issue using the assume independence dependency handling technique can be modeled. The results show that a machine that implements out-of-order fetch/decode/issue using the assume independence dependency handling technique does not perform as well as a machine that implements out-of-order fetch/decode/issue using the assume dependence dependency handling technique.

This chapter also examines two minor implementation tradeoffs: performance versus fetch buffer size, and performance versus the fraction of a checkpoint withheld per hole instruction. (Checkpoint withholding is needed to prevent deadlock in machines equipped with out-of-order fetch/decode/issue.) Understanding these two tradeoffs is not crucial to the understanding out-of-order fetch, decode, and issue. This chapter examines these two tradeoffs solely for the purpose of completeness.

This chapter is organized into five sections. Section 8.1 compares the efficacy of various solutions (prefetching, out-of-order fetch, and out-of-order fetch/decode/issue) to the instruction cache bottleneck for machines that use the default 16k byte direct mapped first level instruction cache. Section 8.2 compares the efficacy of these solutions as the size of the first level instruction cache varies. Section 8.3 evaluates the performance of machines that employ these solutions and that use the default first level instruction cache as their fetch buffer sizes (in number of fetch buffer entries) vary. Section 8.4 evaluates the performance of machines equipped with out-of-order fetch/decode/issue and the default first level instruction cache as the fraction of a checkpoint withheld per hole instruction is varied. Section 8.5 provides a summary of the chapter.

8.1 Comparison of Instruction Cache Bottleneck Solutions

This section compares the effectiveness of three different solutions to the instruction cache bottleneck: prefetching, out-of-order fetch, and out-of-order fetch/decode/issue. To perform this comparison, I simulated five machines.

The first machine is the base microarchitecture presented in Chapter 7. It has a real 16k byte first level instruction cache. It does nothing to deal with the problem of the instruction cache bottleneck. It serves as the baseline.

The second machine is identical to the first machine, except that on a first level instruction cache miss, the machine prefetches the lines located sequentially after the missing line. Prefetch requests probe the first level instruction cache, and are only passed on to the second level instruction cache if the probe results in a miss. Sequential prefetching continues until either a prefetch request hits in the first level instruction cache, or the servicing of the cache miss completes. Note that for both the first machine and the second machine, on every first level instruction cache miss, the fetch requests logically following the request for the cache line that missed are flushed. Consequently, on every cache miss, the branch predictor must be backed up in order to re-generate those requests. (See Chapter 7 for more details.)

The third machine is identical to the first machine, except that it has been equipped with out-of-order fetch.

The fourth machine is identical to the first machine, except that it has been equipped with out-of-order fetch/decode/issue. It implements out-of-order fetch/decode/issue using the assume dependence dependency handling technique. This machine has higher performance than a machine that implements out-of-order fetch/decode/issue using the assume independence dependency handling technique. Experimental results for this latter machine are presented later in this section.

The last machine is also identical to the first machine, except that it has a perfect (100 percent hit rate) first level instruction cache. Hence, it cannot be implemented. This machine provides an upper bound on the performance of any technique (prefetching, out-of-order fetch, out-of-order fetch/decode/issue, . . .) that tries to deal with the problem of the instruction cache bottleneck.

Figure 8.1 shows the performance, in Instructions Per Cycle (IPC), averaged over all the benchmarks (both SPEC and Non-SPEC) for each of the five machines. Results for the individual benchmarks are provided in Figure 8.2 (SPEC benchmarks) and Figure 8.3 (Non-SPEC benchmarks). In the figures, OOO Fetch stands for Out-of-Order Fetch, and OOO FDI stands for Out-of-Order Fetch/Decode/Issue. The average performance of the baseline machine is 2.70 IPC. The average performance of the machine with the perfect instruction cache is 3.53 IPC, which is 31% higher than that of the baseline machine. The machine with prefetching achieves an average speedup of 11% over the baseline machine, the machine with out-of-order fetch achieves a speedup of 22%, and the machine with out-of-order fetch/decode/issue achieves a speedup of 24%. The performance of these three machines comes within 15%, 7%, and 5%, respectively, of the machine with the perfect instruction cache.

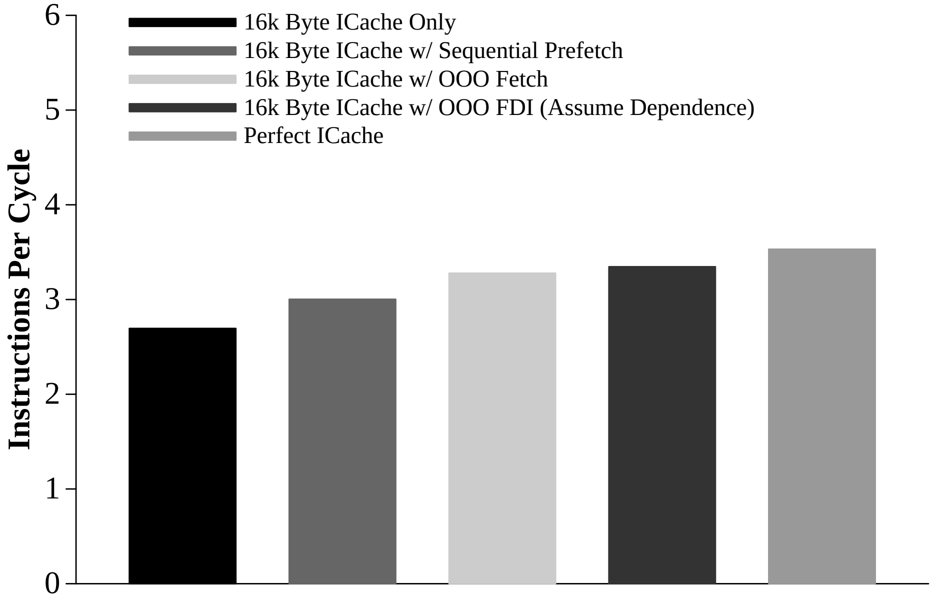


Figure 8.1: Instruction Cache Bottleneck Solutions—Harmonic Average

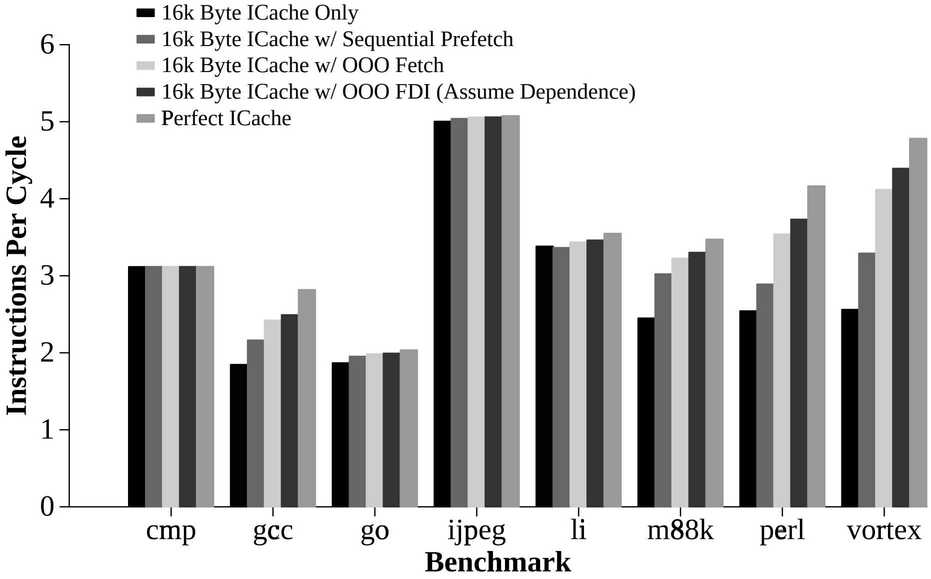


Figure 8.2: Instruction Cache Bottleneck Solutions—SPEC Benchmarks

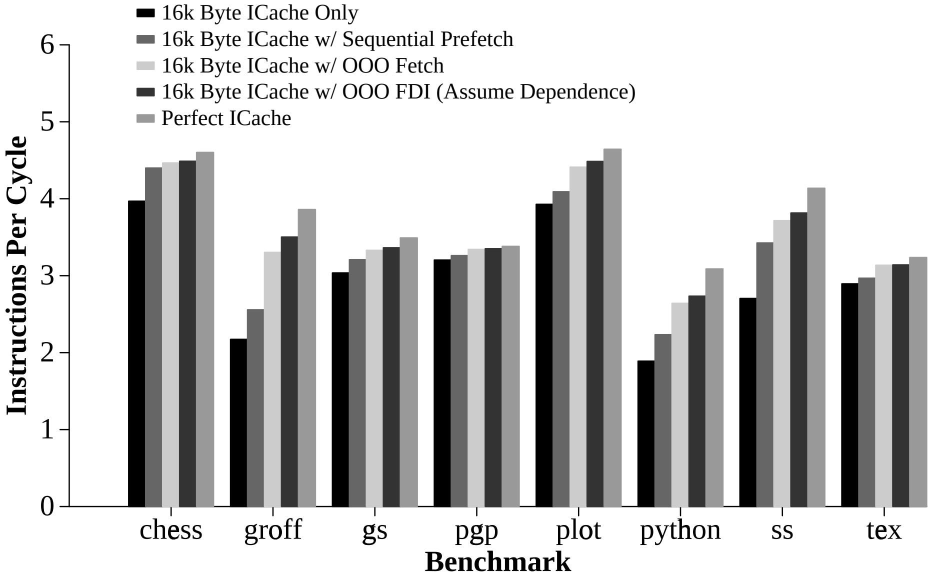


Figure 8.3: Instruction Cache Bottleneck Solutions—Non-SPEC Benchmarks

These results indicate that all three solutions to the instruction cache bottleneck are effective. Out-of-order fetch is much more effective than prefetching. Given that its implementation is simple and inexpensive, perhaps even more so than the implementation of prefetching, it has a clear advantage over prefetching. Out-of-order fetch/decode/issue is not significantly more effective than out-of-order fetch. Given that its implementation is much more complex and expensive than the implementation for out-of-order fetch, it does not have a clear advantage over out-of-order fetch.

Figure 8.4 plots the performance averaged over all the benchmarks for three machines. Each machine is identical to the base microarchitecture presented in Chapter 7, except that each has been equipped with a different variant of out-of-order fetch/decode/issue.

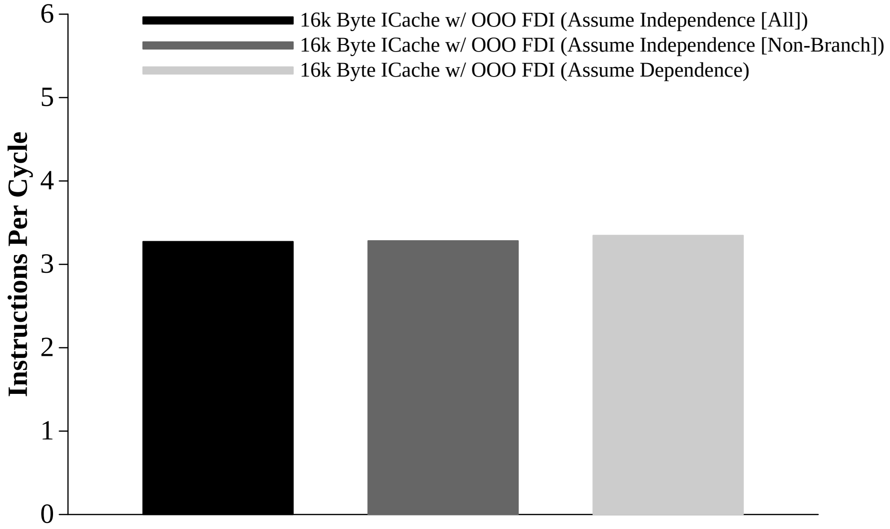


Figure 8.4: Out-of-Order Fetch/Decode/Issue Variants—Harmonic Average

The first machine, labeled "16k Byte ICache w/ OOO FDI (Assume Independence [All])", implements out-of-order fetch/decode/issue using the assume independence dependency handling technique. All instructions—including branches—assume independence. If a mispredicted branch is resolved, and that branch is a post-hole instruction, the fetch unit only flushes the instructions that logically follow the branch from the front-end of the machine. Instructions that logically precede the branch (i. e., the yet-to-be-issued hole instructions) must not be flushed from the front-end. Designing a front-end that can perform such selective flushing is nontrivial.

The second machine, labeled "16k Byte ICache w/ OOO FDI (Assume Independence [Non-Branch])", also implements out-of-order fetch/decode/issue using assume independence. However, for this machine, only the non-branch instructions assume independence. Branch instructions assume dependence; i. e., they are not allowed to execute until the hole disappears. When a mispredicted branch is resolved, it can never be a post-hole instruction. All instructions that logically precede the branch have been issued, and are no longer in the front-end of the machine. Any instructions in the front-end logically follow the branch, and must be flushed. Since this machine does not need a front-end that can be selectively flushed, it is simpler to design than the first machine.

The third machine, labeled “16k Byte ICache w/ OOO FDI (Assume Dependence)”, implements out-of-order fetch/decode/issue using the assume dependence dependency handling technique. It is the same as the identically labeled machine in Figures 8.1–8.3.

The machine that implements out-of-order fetch/decode/issue using assume dependence outperforms the other two machines. Its average performance is 3.35 IPC. The machine that implements out-of-order fetch/decode/issue using assume independence for only non-branch instructions has the second highest performance, at 3.28 IPC. The machine that uses assume independence for all instructions has the lowest performance, at 3.27 IPC. Since implementing out-of-order fetch/decode/issue using assume dependence is as easy or easier than implementing the other two variants of out-of-order fetch/decode/issue, and since it is more effective in eliminating the instruction cache bottleneck than the other two variants, there is no reason to even consider building a machine equipped with one of the other two variants. For the remaining experiments in this chapter, I won't simulate any machines equipped with either of the other two variants of out-of-order fetch/decode/issue.

The two machines that implement out-of-order fetch/decode/issue using assume independence have lower performance because they assume that all post-hole instructions are independent of hole instructions. Many post-hole instructions are dependent on hole instructions, and, consequently, on these two machines, some of the dependent instructions execute using source operands that contain incorrect data. These dependent instructions generate incorrect results, and, as a result, branches are sometimes incorrectly resolved. When a branch is resolved incorrectly, the machine may believe that the branch has been mispredicted, when, in fact, it has been correctly predicted. The machine responds to a perceived mispredict—real or otherwise—by flushing the instructions that logically follow the branch from the machine, and then redirecting instruction fetch to what it believes is the correct path of execution. Compared to the machine that implements out-of-order fetch/decode/issue using assume dependence, the number of machine flushes due to branch mispredicts is 13% greater for the machine that implements out-of-order fetch/decode/issue using assume independence for only non-branch instructions. For the machine that implements out-of-order fetch/decode/issue using assume independence for all instructions, this number jumps to 18%. Most of these additional machine flushes are probably due to incorrectly resolved branches. The results for the individual benchmarks are provided in Figure 8.5 (SPEC benchmarks) and Figure 8.6 (Non-SPEC benchmarks).

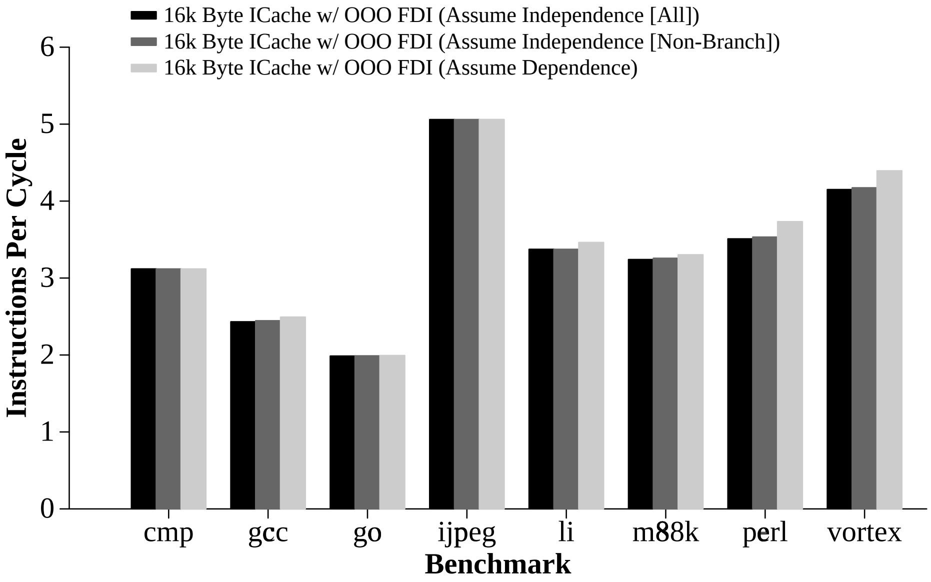


Figure 8.5: Out-of-Order Fetch/Decode/Issue Variants—SPEC Benchmarks

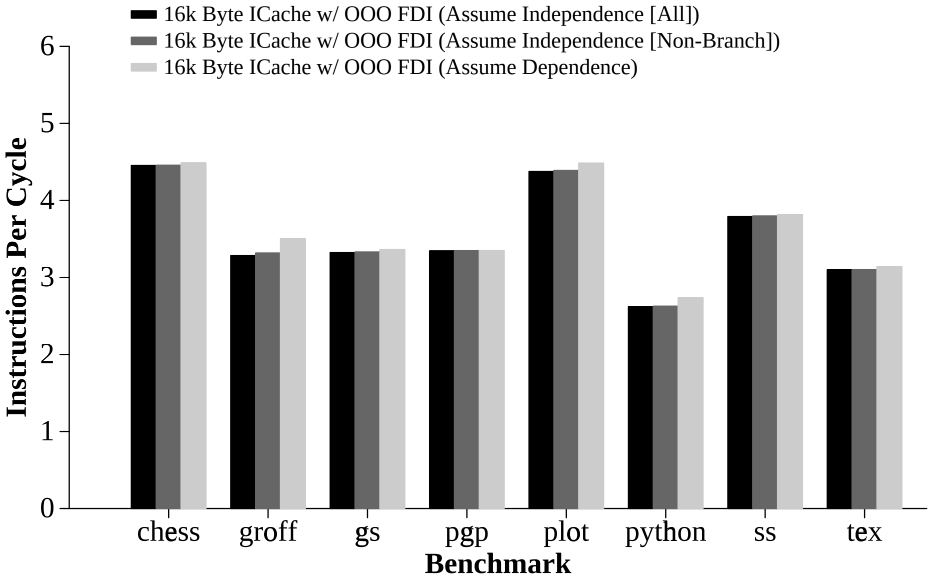


Figure 8.6: Out-of-Order Fetch/Decode/Issue Variants—Non-SPEC Benchmarks

8.2 Varying the Instruction Cache Size

Figure 8.7 shows the performance, in Instructions Per Cycle (IPC), averaged over all the benchmarks (both SPEC and Non-SPEC) for five machines. These five machines are the same as those that were used to generate Figures 8.1–8.3 in Section 8.1. The first level instruction cache size for each machine (with the exception of the machine with the perfect first level instruction cache) was varied from 4k bytes to 64k bytes. The access time of the first level instruction cache required one cycle regardless of its size; i. e., the access time was not scaled with the size of the first level instruction cache.

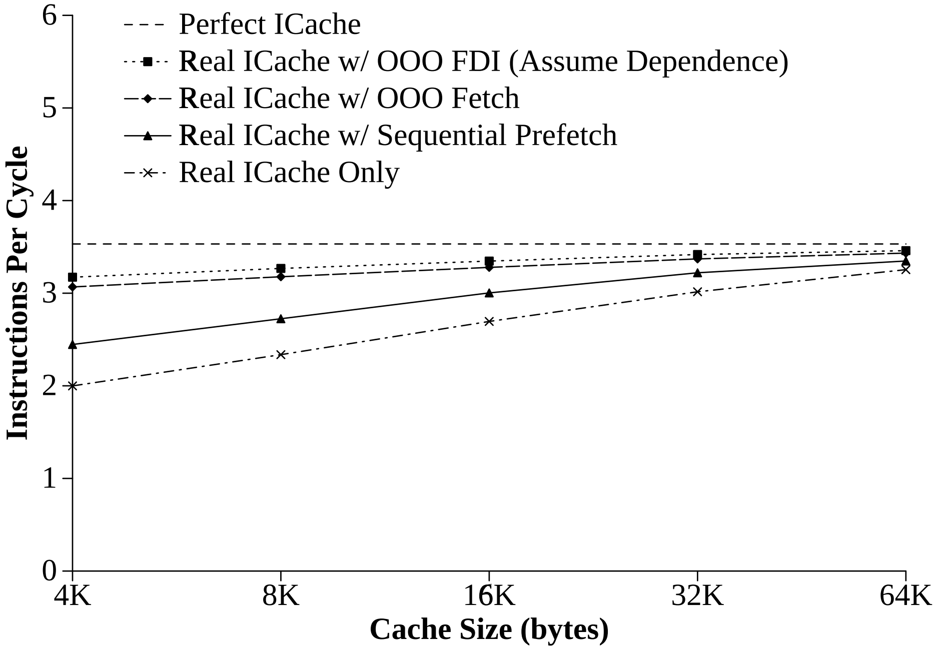


Figure 8.7: Varied Instruction Cache Size—Harmonic Average

As the cache size increases, fewer misses occur, and thus, there is less of a need to do something about the instruction cache bottleneck. As a result, the performance benefit of the three solutions to the instruction cache bottleneck (out-of-order fetch/decode/issue, out-of-order fetch, and prefetching) decreases as the cache size increases. The machine equipped with out-of-order fetch/decode/issue achieves a 59% gain in performance over the baseline machine (labeled “Real ICache Only”) at a cache size of 4k bytes. This gain falls to 6% when the cache size is increased to 64k bytes. The machine equipped with out-of-order fetch achieves a 53% gain at a cache size of 4k bytes and only a 5% gain at a cache size of 64k bytes. For the machine equipped with prefetching, these two percentages are 22% (4k byte cache) and 3% (64k byte cache).

The performance of a machine that uses an effective solution to the instruction cache bottleneck is less sensitive to cache size than the performance of a machine that uses a less effective solution. The performance of the machine equipped with out-of-order fetch/decode/issue drops by 8% as the cache size is reduced from 64k bytes to 4k bytes. This percentage is 11% for the machine equipped with out-of-order fetch, and 27% for the machine equipped with prefetching. For comparison, the performance of the baseline machine drops by 39% as the cache size is reduced from 64k bytes to 4k bytes.

A consequence of this fact is that it is possible to achieve the same performance at a lower cost by equipping a machine with an effective solution to the instruction cache bottleneck. For example, the baseline machine with a 64k byte cache has a performance of 3.25 IPC. If the baseline machine is equipped with out-of-order fetch, the size of the cache can be reduced to 16k bytes without sacrificing any performance. (The performance of this resulting machine would be 3.28 IPC.) Results for individual benchmarks are provided in Figure A.47 (SPEC) and Figure A.48 (Non-SPEC) of Appendix A.

8.3 Varying the Fetch Buffer Size

Figure 8.8 shows the performance, in Instructions Per Cycle (IPC), averaged over all the benchmarks (both SPEC and Non-SPEC) for five machines as their fetch buffer sizes, in number of fetch buffer entries, are varied from 4 entries to 32 entries. These five machines are the same as those that were used to generate Figures 8.1–8.3 in Section 8.1. (They all use the default 16k byte direct mapped first level instruction cache, with the exception of the machine with the perfect first level instruction cache.) The purpose of the fetch buffer is to compensate for cycle-to-cycle variances between the fetch rate and the issue rate. Larger fetch buffers are more effective at this than smaller fetch buffers, because they can tolerate larger variances. As a result, as the fetch buffer size increases, the average issue rate increases, and performance increases.

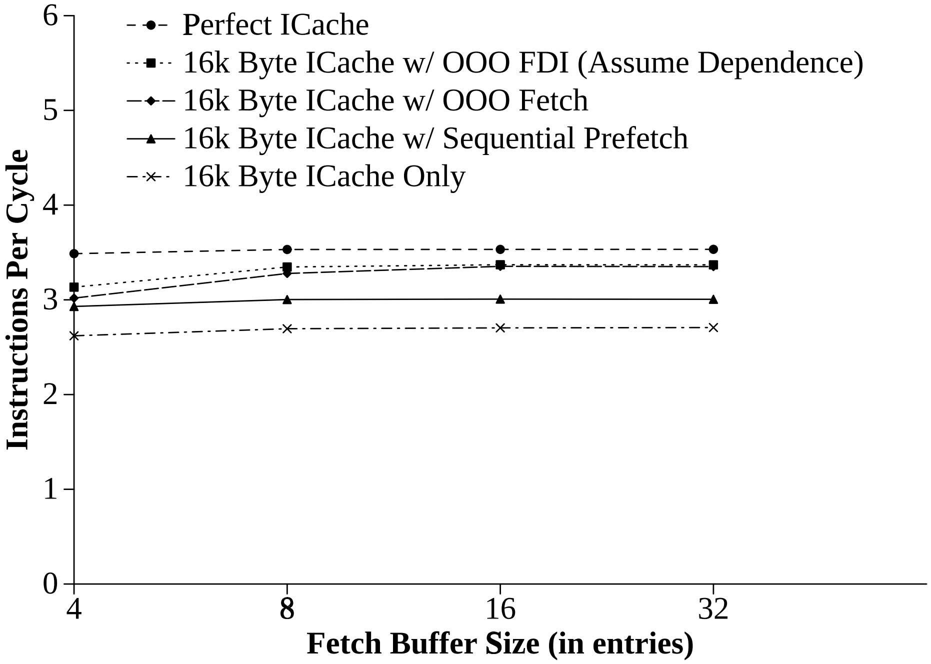


Figure 8.8: Varied Fetch Buffer Size—Harmonic Average

The performance of the machine with out-of-order fetch is strongly dependent on fetch buffer size, as is the performance of the machine with out-of-order fetch/decode/issue. The performance of the other three machines is only weakly dependent on fetch buffer size.

The fetch buffer size sets an upper bound on the number of fetch requests with outstanding first level instruction cache misses that the machine can support. Each fetch request is allocated a fetch buffer entry before it is issued to the instruction cache. If all entries are allocated, instruction fetch stalls until new entries become available. An entry is deallocated only after its assigned request has completed, the request has written all its instruction data into the entry, and the decode/issue logic has emptied the entry of all its instructions. If a request misses in the instruction cache, its entry is unavailable until after the miss has been serviced. Hence, for a machine with a fetch buffer that has X ($X \in \{1, 2, \ldots\}$) entries, instruction fetch stalls whenever there are X fetch requests with outstanding misses.

The machine with a perfect instruction cache does not experience instruction cache misses, so its performance is only weakly dependent on fetch buffer size. For the baseline machine ("16k Byte ICache Only") and the machine with prefetching, instruction fetch stalls whenever a fetch request misses in the instruction cache. These two machines never have more than one fetch request with an outstanding miss. Thus, their performance is also only weakly dependent on fetch buffer size. The only machines that support multiple fetch requests with outstanding misses are the machine with out-of-order fetch and the machine with out-of-order fetch/decode/issue. Hence, of the five machines, these two machines are the only machines whose performance is strongly dependent on fetch buffer size.

At small fetch buffer sizes, the machine with out-of-order fetch/decode/issue has a larger performance advantage over the machine with out-of-order fetch than it does at large sizes. At small sizes, each fetch buffer entry is a critical resource. The machine with out-of-order fetch/decode/issue uses these critical resources more efficiently than the machine with out-of-order fetch, and thus, it performs better. The machine with out-of-order fetch/decode/issue deallocates fetch buffer entries more quickly than the machine with out-of-order fetch, because it decodes and issues instructions out-of-order. Quickly deallocating entries is important, because it reduces the number of times instruction fetch stalls due to a lack of available entries. The machine with out-of-order fetch, on the other hand, decodes and issues instructions in program order. Its fetch buffer entries are deallocated in the same order that they were allocated, so its entries are not deallocated as quickly.

The results for the individual benchmarks are provided in Figure 8.9 (SPEC benchmarks) and Figure 8.10 (Non-SPEC benchmarks).

For some of the benchmarks (most notably, cmp, ijpeg, and vortex), increasing the size of a machine's fetch buffer results in lower performance. As the fetch buffer size increases, the average issue rate increases. The issue rate increases because the decode/issue logic packs more instructions into each issue packet. Recall that each issue packet is assigned a checkpoint. With more instructions in each issue packet, more instructions are assigned to each checkpoint. If free (unallocated) checkpoints are plentiful, increasing the number of instructions assigned to a checkpoint is detrimental. A machine that has a large number of free checkpoints is not bottlenecked by a lack of storage capacity in the Node Tables. Increasing the number of instructions assigned to each checkpoint will not increase the total number of instructions stored in the Node Tables. It will only reduce the number of checkpoints that cover the instructions stored there. When the machine needs to repair to a known previous state—for example, when a branch mispredict is detected—the checkpoints are further apart, so more useful work is discarded. For most of the benchmarks, free checkpoints are plentiful, so increasing the number of instructions assigned to each checkpoint can degrade performance.

For a few benchmarks (i. e., chess, m88k, and ss), at a fetch buffer size of 4 entries, the machine with prefetching outperforms the machine with out-of-order fetch and the machine with out-of-order fetch/decode/issue. The machine with prefetching prefetches sequential cache lines whenever one of its fetch requests misses in the instruction cache. The prefetch requests probe the instruction cache, and are only passed on to the next level of the memory hierarchy if the probe results in a miss. Sequential prefetching continues until either a prefetch request hits in the instruction cache, or the servicing of the cache miss completes. Many lines may be prefetched into the instruction cache during an instruction cache miss, preventing many future misses, particularly if the machine is executing a piece of code for the first time. For the machine with out-of-order fetch and the machine with out-of-order fetch/decode/issue, on the other hand, instruction fetch stalls whenever there are 4 fetch requests with outstanding misses. This may occur fairly frequently, especially if the machine is executing a piece of code for the first time. Thus, there are certain situations in which prefetching is better than either out-of-order fetch or out-of-order fetch/decode/issue.

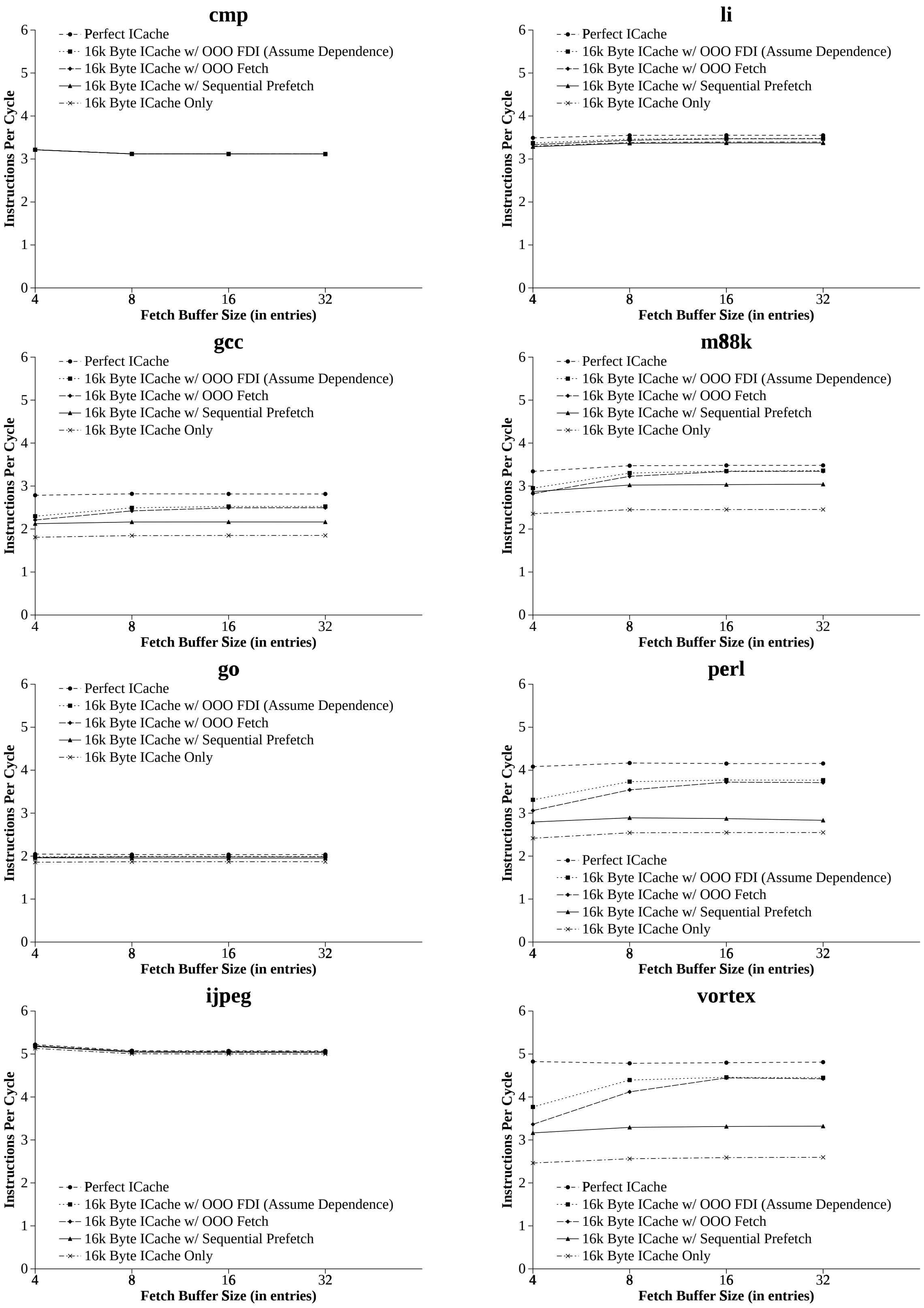


Figure 8.9: Varied Fetch Buffer Size—SPEC Benchmarks

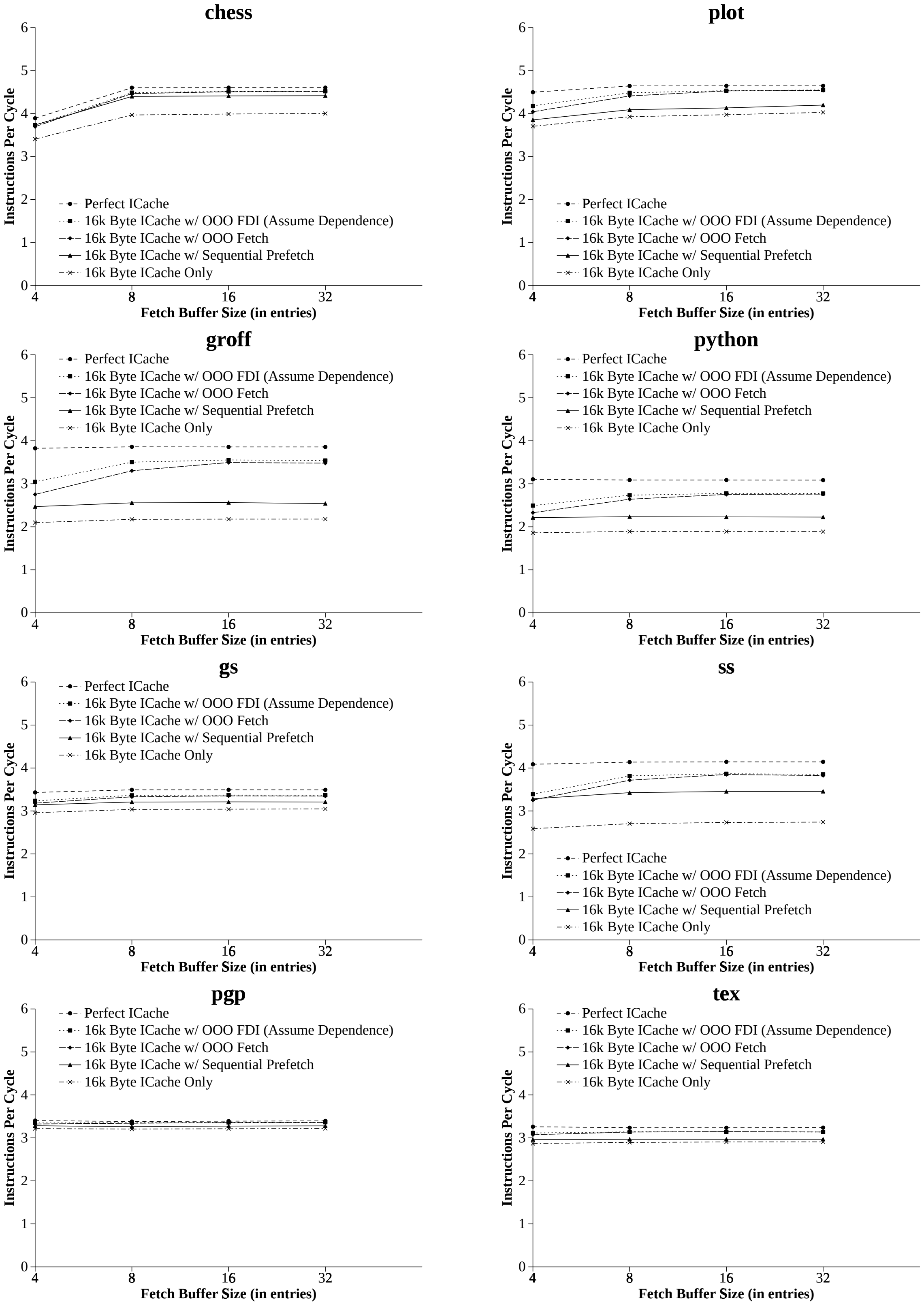


Figure 8.10: Varied Fetch Buffer Size—Non-SPEC Benchmarks

8.4 Varying the Checkpoint Withholding

Checkpoint withholding is needed to prevent deadlock in machines equipped with out-of-order fetch/decode/issue. When a post-hole instruction is issued, the machine must guarantee that all hole instructions that logically precede that post-hole instruction will have the checkpoints they need to be issued into the machine. If this guarantee is not met, the machine will deadlock: the post-hole instruction will not retire—and hence, release the checkpoint allocated to it—until the hole instructions have retired, yet the hole instructions cannot be issued because they don't have the required checkpoints.

To meet this guarantee, the machine withholds some of its checkpoints for the hole instructions whenever it issues a post-hole instruction. The machine knows how many hole instructions there are. Unfortunately, it does not know how many checkpoints will be needed for these hole instructions, since the number of instructions that can fit on a checkpoint varies.

Fortunately, as long as the machine withholds at least one checkpoint for the hole instructions, deadlock is prevented. Whenever there are two or more free (unallocated) checkpoints, the machine may use those checkpoints to issue whichever instructions it chooses. Whenever there is only one free checkpoint, that checkpoint is withheld for the hole instructions, so the machine may not use that checkpoint to issue post-hole instructions. The machine uses that remaining free checkpoint to try to issue, in program order, all the hole instructions, starting with the hole instruction that appears the earliest in the dynamic instruction stream, and working towards the hole instruction that appears the latest. If the number of hole instructions is greater than the number of instructions that can fit on a checkpoint, the machine will need to use the remaining free checkpoint multiple times. Each time, it will assign an issue packet's worth of hole instructions to the checkpoint, issue those instructions, and then wait for them to retire. After those instructions have retired, it will use the checkpoint to issue a new group of instructions. Essentially, whenever the machine has only one free checkpoint, it operates as an in-order issue machine that supports only a single speculative checkpoint. Hence, even though a machine can prevent deadlock by withholding only one checkpoint for all of the hole instructions, this technique for preventing deadlock may result in poor performance.

Another technique for preventing deadlock is to withhold a set fraction of a checkpoint for each hole instruction. To calculate the number of checkpoints that need to be withheld, the number of hole instructions is multiplied by this fraction, and then the result is rounded up to the nearest integer. For example, if on average 8 instructions fit on a checkpoint, a machine might want to withhold 1/8 of a checkpoint per hole instruction. If there are 17 hole instructions, the machine would withhold 3 (i. e., $\lceil 17 \times 1/8 \rceil$) checkpoints. The fraction is selected such that the multiplication is easily performed in hardware; i. e., the fraction is $\frac{1}{2^n}$ for n $\in \{0, 1, \ldots\}$. Rounding the result up to the nearest integer ensures that at least one checkpoint is withheld whenever there are any hole instructions. If this technique underestimates the number of checkpoints that need to be withheld, deadlock is prevented via the technique described in the preceding paragraph.

For the machines modeled in this chapter, 16 instructions can fit on a checkpoint. One of these machines might optimistically assume that the hole instructions will be packed 16 per checkpoint. This machine would withhold 1/16 of a checkpoint for each hole instruction. A machine might also assume the average, as was done in the example. Or, a machine might pessimistically assume that the hole instructions will only be packed 1 per checkpoint, and therefore withhold a whole checkpoint for each hole instruction.

If the fraction of a checkpoint withheld per hole instruction is too low, the machine underestimates the number of checkpoints that need to be withheld, and must rely on the first technique to prevent deadlock. The first technique may result in poor performance. Consequently, if the fraction is too low, the performance may suffer. On the other hand, if the fraction is too high, the machine overestimates the number of checkpoints that need to be withheld. The machine may not be able to issue a packet of post-hole instructions because all of the free (unallocated) checkpoints have been withheld. Thus, if the fraction is too high, the performance may suffer.

Figure 8.11 shows the performance, in Instructions Per Cycle (IPC), averaged over all the benchmarks (both SPEC and Non-SPEC) for a machine with the default first level instruction cache that implements out-of-order fetch/decode/issue using the assume dependence dependency handling technique as the fraction of a checkpoint that is withheld per hole instruction is varied from 1 to ϵ. A fraction of ϵ indicates that if there are hole instructions, 1 checkpoint—and only 1 checkpoint—is withheld regardless of the number of hole instructions. That is, a fraction of ϵ indicates that the machine uses the first technique for preventing deadlock. The results show that the fraction of a checkpoint that is withheld per hole instruction does not affect performance. The likely reason for this is that the machine, which supports 63 checkpoints for speculative state, rarely uses all of its checkpoints. The machine almost always has an abundance of free (unallocated) checkpoints. Withholding too many or too few of these free checkpoints (i. e., ϵ too high or too low) does not matter, as long as there is at least one free checkpoint that can be used to issue a packet. The results for the individual benchmarks, which are similarly uninteresting, are provided in Figure A.49 (SPEC benchmarks) and Figure A.50 (Non-SPEC benchmarks) of Appendix A.

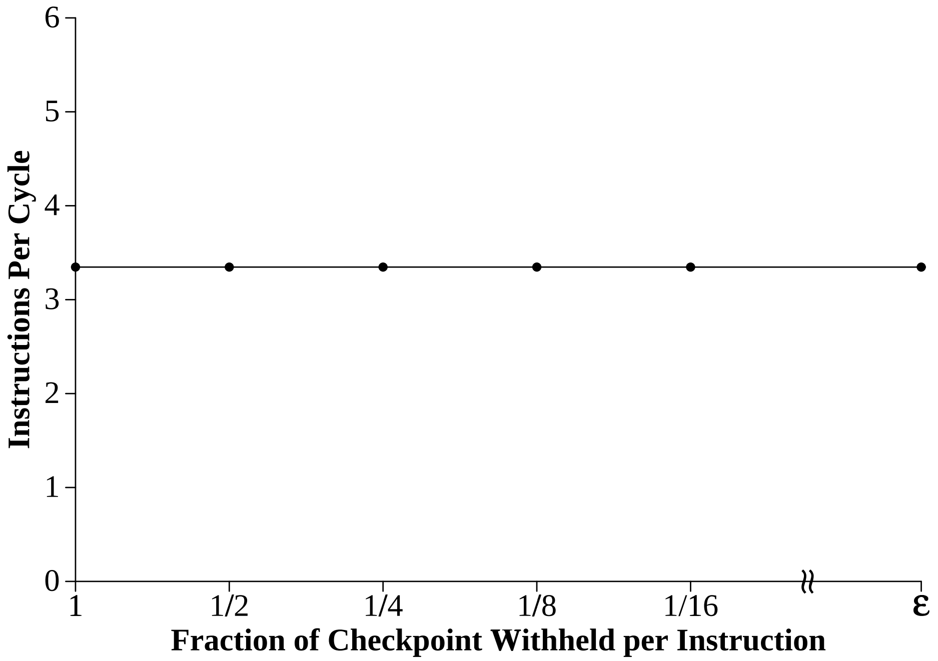


Figure 8.11: Varied Checkpoint Withholding—Harmonic Average

8.5 Summary

Out-of-order fetch, decode, and issue is an effective way to eliminate the performance penalty that results from instruction cache misses. The base microarchitecture loses 24% of its performance as a result of instruction cache misses, whereas the comparable microarchitecture with out-of-order fetch loses only 7%, and the comparable microarchitecture with out-of-order fetch/decode/issue loses only 5%. Put another way, equipping the base microarchitecture with out-of-order fetch eliminates 70% of the performance penalty that results from instruction cache misses, and equipping the base microarchitecture with out-of-order fetch/decode/issue eliminates 78% of that penalty.

Out-of-order fetch is almost as effective as out-of-order fetch/decode/issue in eliminating the performance penalty that results from instruction cache misses. The performance of the base microarchitecture increased by 22%, from 2.70 IPC to 3.28 IPC, when it was equipped with out-of-order fetch. When it was equipped with out-of-order fetch/decode/issue, performance increased by 24%, to 3.35 IPC. For a reference point, when it was equipped with a perfect first level instruction cache, performance increased by 31%, to 3.53 IPC.

The variant of out-of-order fetch/decode/issue that uses the assume dependence dependency handling technique outperforms the variant that uses the assume independence dependency handling technique. When the base microarchitecture was equipped with the variant that uses assume dependence, performance increased by 24%, to 3.35 IPC. When it was equipped with the variant that uses assume independence, performance only increased by 21%, to 3.28 IPC.

CHAPTER 9

Conclusion

Significant parallelism exists within a single instruction stream [14, 59]. To achieve high performance, today's processors are built to take advantage of some of this instruction level parallelism. To exploit even larger amounts of instruction level parallelism, tomorrow's processors will be built with wider issue widths. It has been projected that by the year 2005, it will be possible to place a billion transistors on a chip. With a billion transistors on a chip, a processor that can issue sixteen or more instructions per cycle is not infeasible.

Enough parallelism exists to justify building a processor that can issue 16 instructions per cycle. Using an abstract machine model, the performance of an ideal machine with an issue rate of 16 instructions per cycle and a window size of 1024 instructions was calculated. This machine was ideal in that it had four perfect components: a perfect (100 percent hit rate) instruction cache, a perfect (omniscient) branch predictor, a perfect execution core (i. e., an execution core with an unbounded number of functional units, each of which can perform every desired operation), and a perfect single cycle data cache. The performance of this machine averaged over sixteen integer benchmarks was calculated to be 12.4 instructions per cycle (IPC). This performance is far greater than that of any existing processors. Unfortunately, because the four perfect components cannot be built, this machine cannot be built.

To make this machine more realistic (i. e., buildable), the four perfect components were replaced by real components. That is, the machine was given a real instruction cache, a real branch predictor, a real execution core, and a real data cache. Both the instruction cache and data cache were 16k byte direct mapped caches. The branch predictor used a 16-bit gshare [81] predictor to predict the direction of conditional branches; the "tagless" variety of the pattern based predictor proposed by Chang, Hao, and Patt [19] to predict the targets of indirect (or computed) branches (a 9-bit global history was used); and a 64 entry Return Address Stack to predict the targets of subroutine returns. The execution core consisted of sixteen functional units.

Whenever a real (i. e., non-ideal) component is used instead of a perfect component, a performance bottleneck is created. Hence, this more realistic machine had four bottlenecks. The first bottleneck, created by using the real instruction cache, is due to instruction cache misses. The second bottleneck, created by using the real branch predictor, is due to branch mispredicts. The third bottleneck, created by using the real execution core, is due to a lack of execution bandwidth. And the fourth bottleneck, created by using the real data cache, is due to data cache misses.

Of these four bottlenecks, the bottleneck due to instruction cache misses was found to be the most severe bottleneck; i. e., of the four bottlenecks, the bottleneck due to instruction cache misses imposes the most severe performance penalty on the machine. The bottlenecks, in decreasing order of severity, were that due to instruction cache misses, that due to branch mispredicts, that due to data cache misses, and that due to a lack of execution bandwidth. The bottleneck due to instruction cache misses and the bottleneck due to branch mispredicts were found to impose significant performance penalties, whereas the bottleneck due to data cache misses and the bottleneck due to a lack of execution bandwidth were found to impose only minor performance penalties. All together, the four bottlenecks reduce the average performance (over the sixteen integer benchmarks) of the machine from 12.4 IPC to 3.0 IPC.

Fortunately, the most severe bottleneck—the bottleneck due to instruction cache misses—can be nearly eliminated with out-of-order fetch, decode, and issue. Instruction cache misses prevent the machine from fetching, decoding, and issuing new useful instructions until the instruction cache miss has been serviced. Out-of-order fetch, decode, and issue reduce the performance penalty due to instruction cache misses by enabling the machine to continue fetching, decoding, and issuing new useful instructions in the event of an instruction cache miss. Using the abstract machine model, the performance of the more realistic machine (i. e., the machine with a real instruction cache, a real branch predictor, a real execution core, and a real data cache) increased by an average of 63%, from 3.0 IPC to 4.9 IPC, when the machine was given a perfect instruction cache rather than a real instruction cache. When the machine was given a realizable form of out-of-order fetch, decode, and issue, performance increased by an average of 47%, from 3.0 IPC to 4.4 IPC. Thus, out-of-order fetch, decode, and issue can be used to eliminate most of the performance penalty that results from instruction cache misses.

Most of the performance benefit due to out-of-order fetch, decode, and issue comes from allowing the machine to fetch instructions out of program order. A machine with out-of-order fetch initiates fetch requests in program order, but allows these requests to complete out-of-order. As in conventional processors, the instructions are still decoded and issued in program order. A machine with out-of-order fetch/decode/issue not only fetches instructions out-of-order, it also decodes and issues them out-of-order. (Both out-of-order fetch and out-of-order fetch/decode/issue are variants of the concept of out-of-order fetch, decode, and issue.)

Using a sophisticated machine model, the average performance of a baseline machine was found to be 2.70 IPC. This machine could issue up to 16 instructions per cycle and had a window size of about 1024 instructions. It used a real instruction cache, a real branch predictor, a real execution core, and a real data cache. These four real components were configured as described at the beginning of this chapter, except that the data cache was 64k bytes rather than 16k bytes. This machine did not employ either out-of-order fetch or out-of-order fetch/decode/issue. When the baseline machine was equipped with out-of-order fetch, performance increased by an average of 22%, from 2.70 IPC to 3.28 IPC. When the baseline machine was equipped with a realizable form of out-of-order fetch/decode/issue, performance increased by an average of 24%, from 2.70 IPC to 3.35 IPC. For a reference point, when the baseline machine was given a perfect instruction cache rather than a real instruction cache, performance increased by an average of 31%, from 2.70 IPC to 3.53 IPC.

The first figure in this dissertation, Figure 1.1, demonstrated the problem investigated by the dissertation; i. e., the instruction cache bottleneck. It plotted the performance, in Instructions Per Cycle (IPC), of the gcc benchmark for two different processors as the fetch width of these processors varied between 1 and 64. Both processors had perfect branch prediction, large instruction windows, large pools of functional units, and perfect data caches. To illustrate the performance degradation that results from instruction cache misses, one processor was given a perfect instruction cache, and the other was given a real 16k byte direct mapped instruction cache.

Figure 9.1 is that same figure with an additional line that demonstrates the solution: out-of-order fetch, decode, and issue. This line, labeled “Real ICache w/ OOO FDI”, plots the performance of the processor with the real instruction cache when that processor is equipped with out-of-order fetch/decode/issue. [1] By comparing the performance of a processor with a real instruction cache to the performance of a processor with a perfect instruction cache, the amount of performance lost due to instruction cache misses can be assessed. For the processor that is not equipped with out-of-order fetch/decode/issue, 22% of the performance is lost due to instruction cache misses at a fetch width of 1, 81% is lost at a width of 16, and 90% is lost at a width of 64. For the processor that is equipped with out-of-order fetch/decode/issue, 13% of the performance is lost at a width of 1, 19% is lost at a width of 16, and 2% is lost at a width of 64. Thus, out-of-order fetch, decode, and issue eliminates most of the performance penalty that results from instruction cache misses.

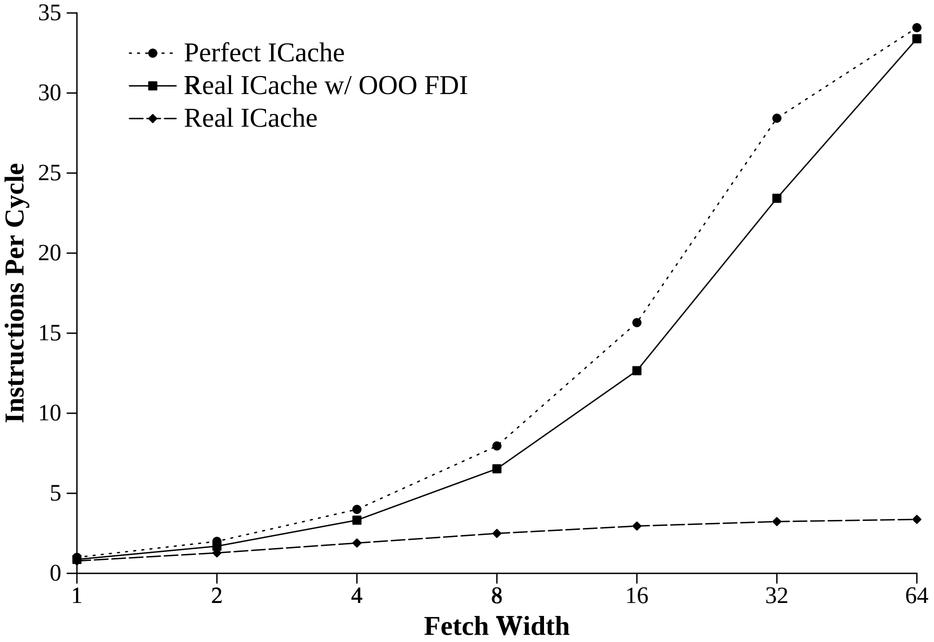


Figure 9.1: Demonstration of the problem (the instruction cache bottleneck) and the solution (out-of-order fetch, decode, and issue) investigated by this dissertation

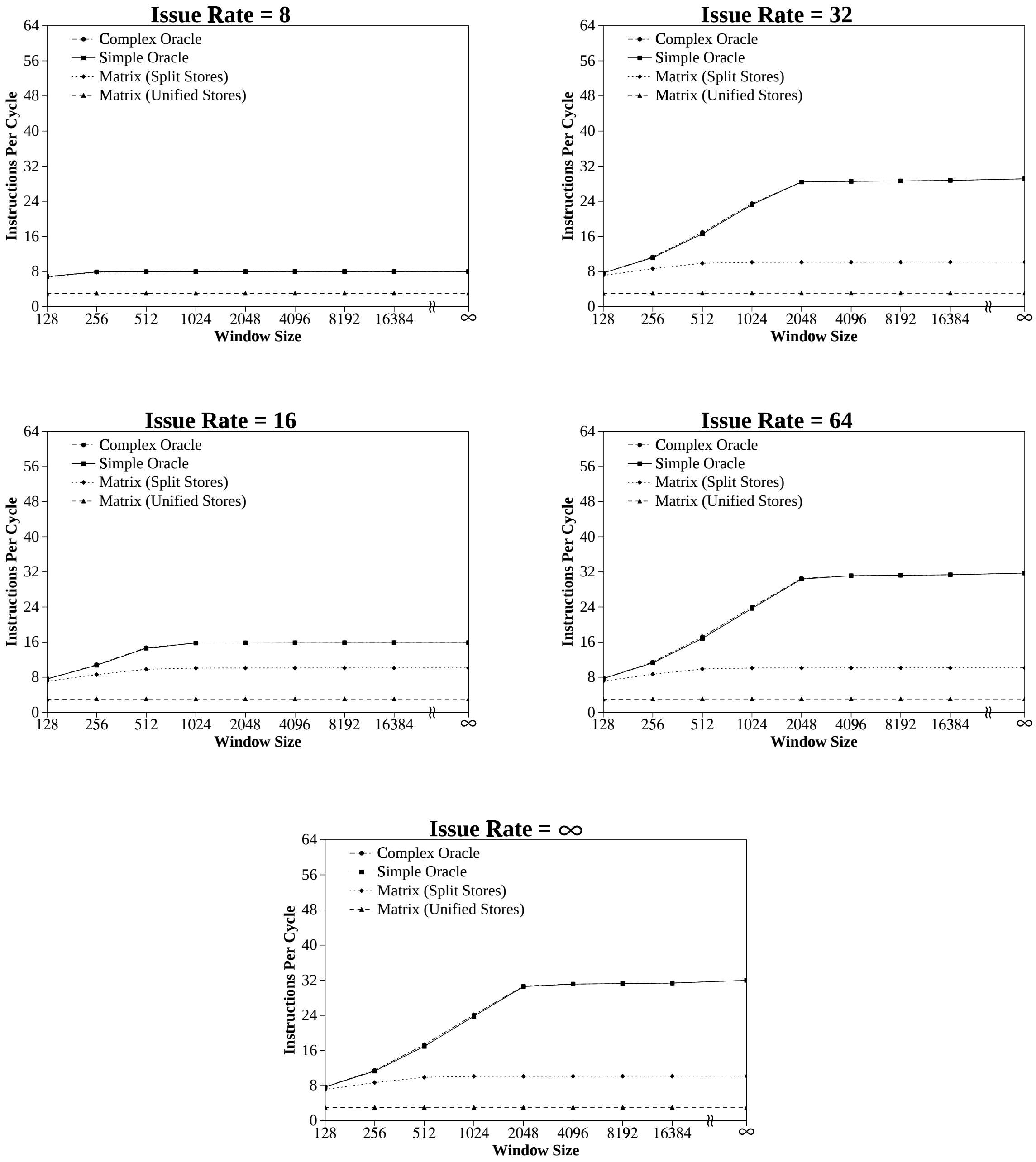
[1] The processor implemented out-of-order fetch/decode/issue using the assume dependence dependency handling technique, which is a realistic (i. e., implementable) dependency handling technique.

After the bottleneck due to instruction cache misses, the bottleneck due to branch mispredicts was the next most severe bottleneck. Indeed, branch mispredicts imposed a performance penalty that was almost as great as the performance penalty imposed by instruction cache misses. Branch Target Buffer (BTB) misses hinder the ability of the branch predictor to quickly and correctly predict branches. This dissertation proposed a new BTB indexing scheme that reduces the number of BTB misses. For a 2048 entry, 4-way set-associative BTB, the average miss rate was reduced from 3.78% to 1.04%. However, even with a perfect (100 percent hit rate) BTB, branch mispredicts are far too frequent. Unless substantial advances are made in branch prediction research, the performance of a machine that can issue 16 or more instructions per cycle may not justify the machine's cost, and, consequently, such machines will not be built.

APPENDIX

APPENDIX A

Additional Figures



Issue Rate = 8

Issue Rate = 32

Issue Rate = 16

Issue Rate = 64

Issue Rate = ∞

Figure A.1: Memory Disambiguation Techniques—Cmp

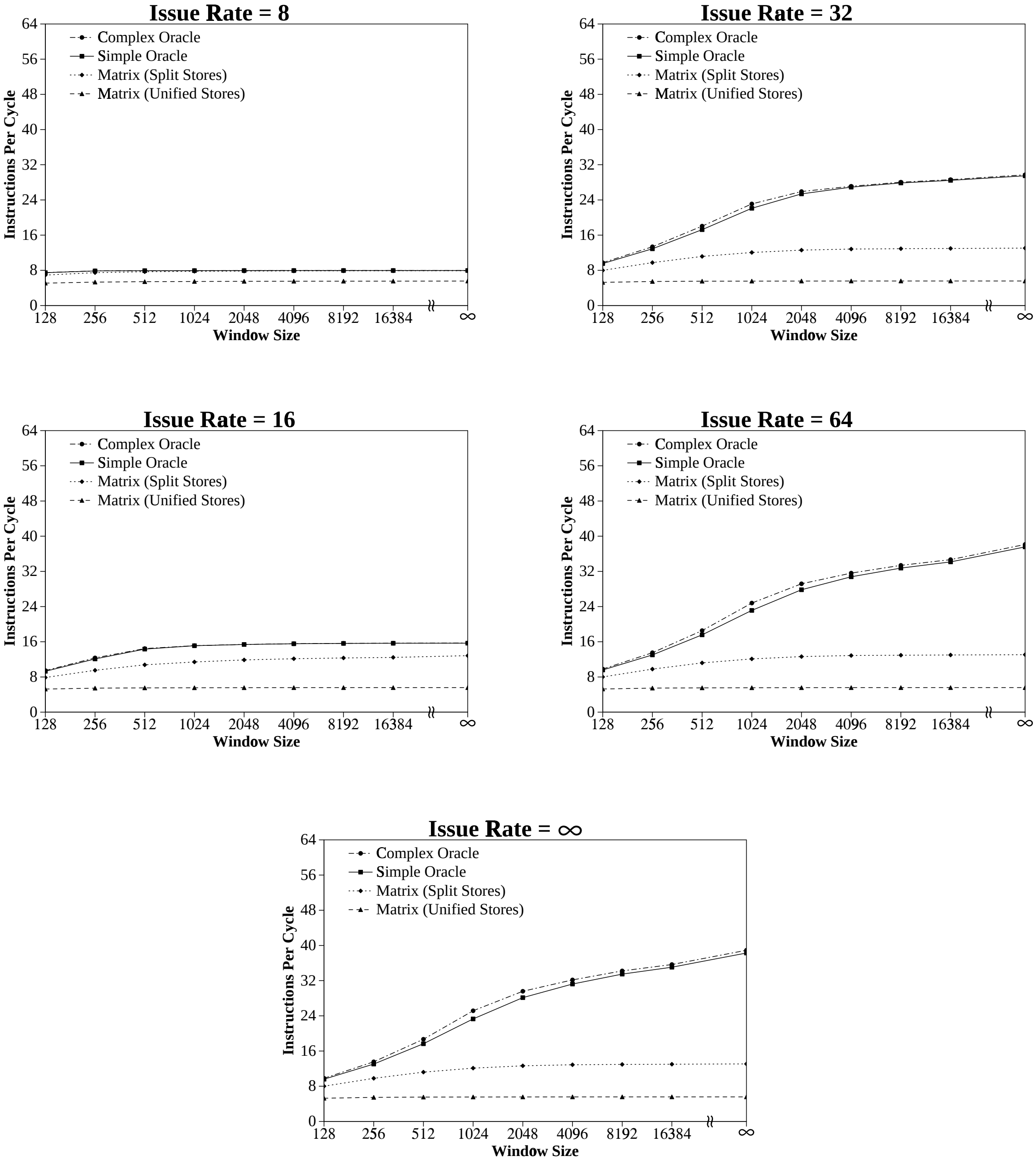


Figure A.2: Memory Disambiguation Techniques—Gcc

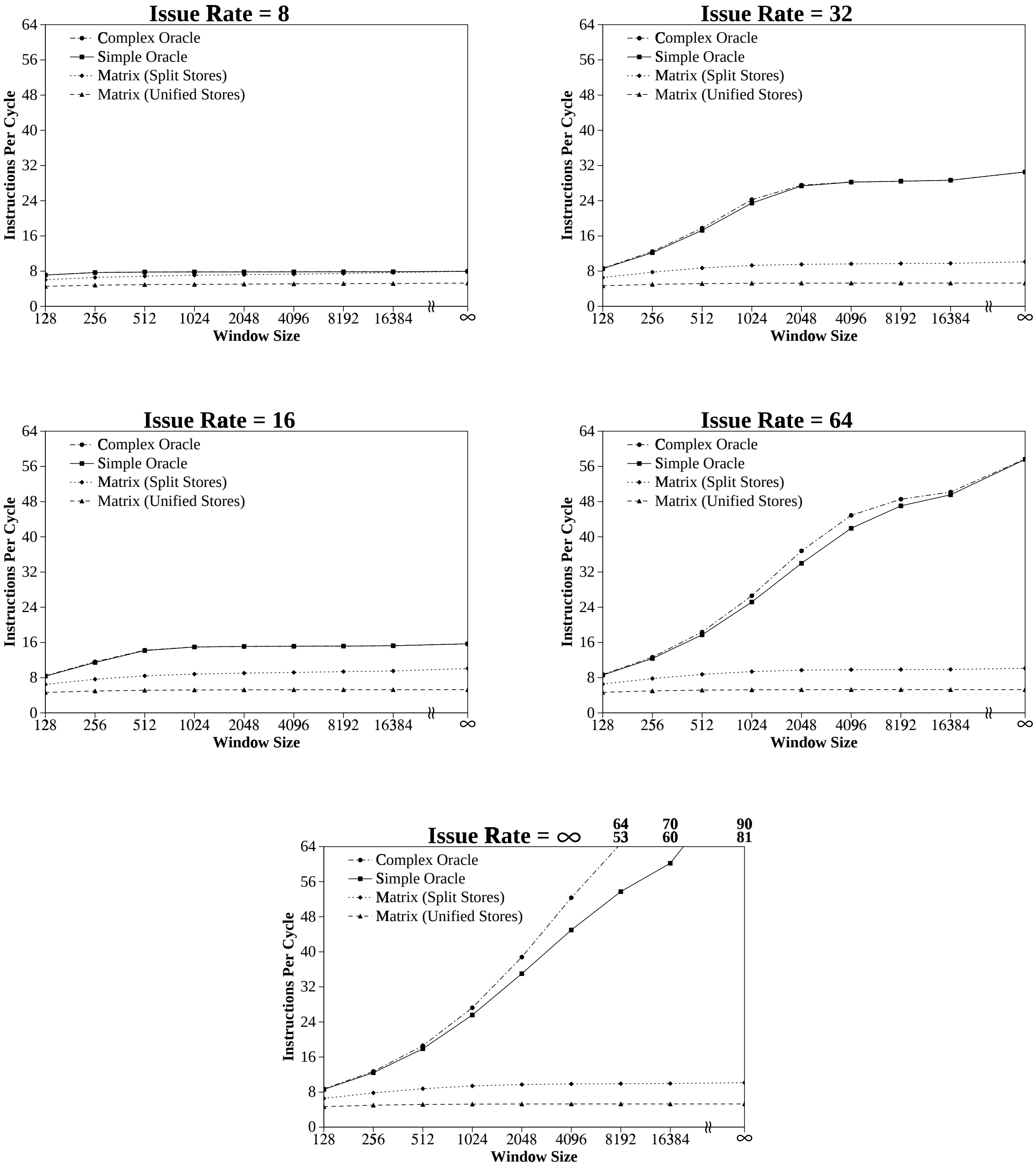


Figure A.3: Memory Disambiguation Techniques—Go

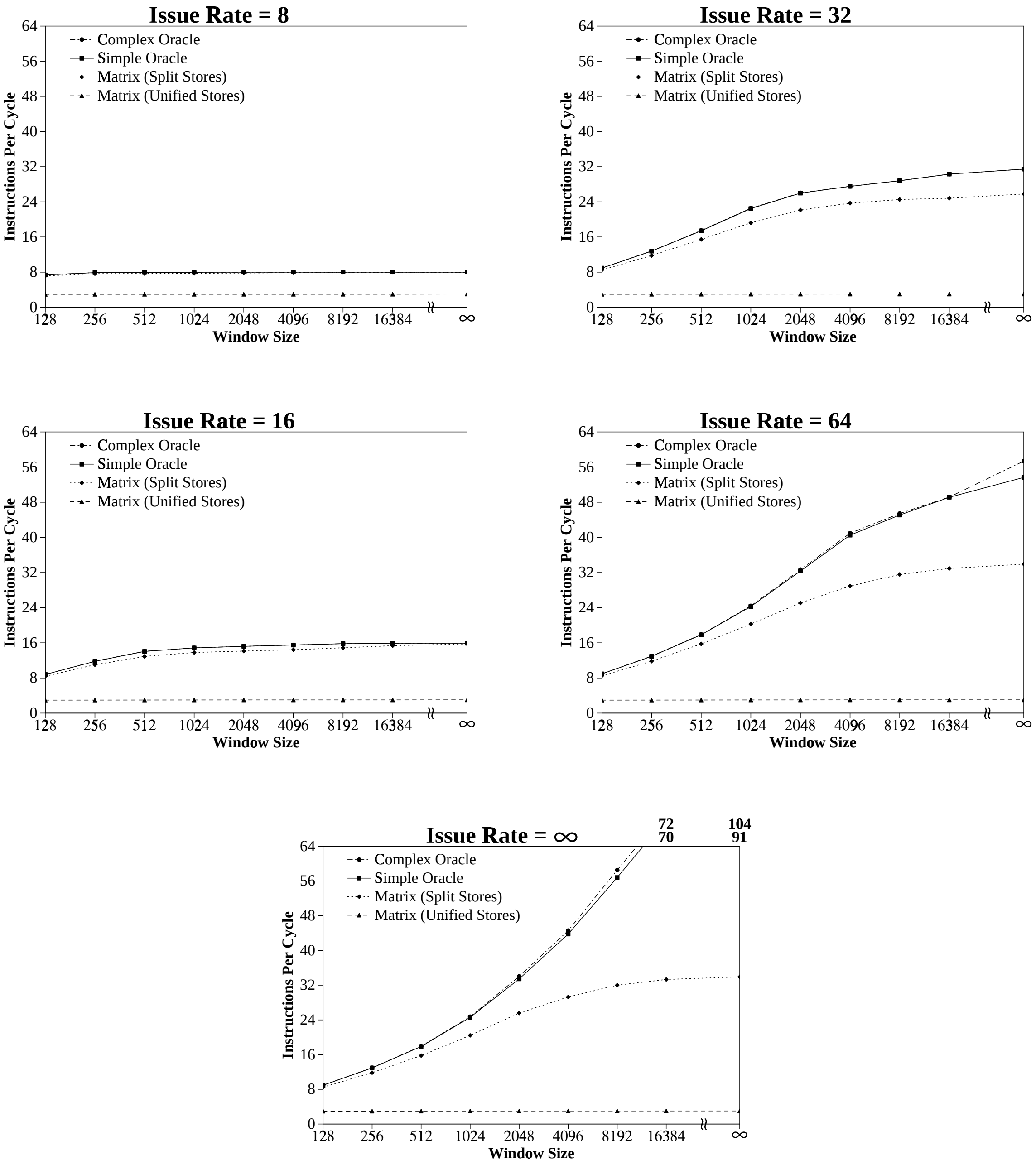
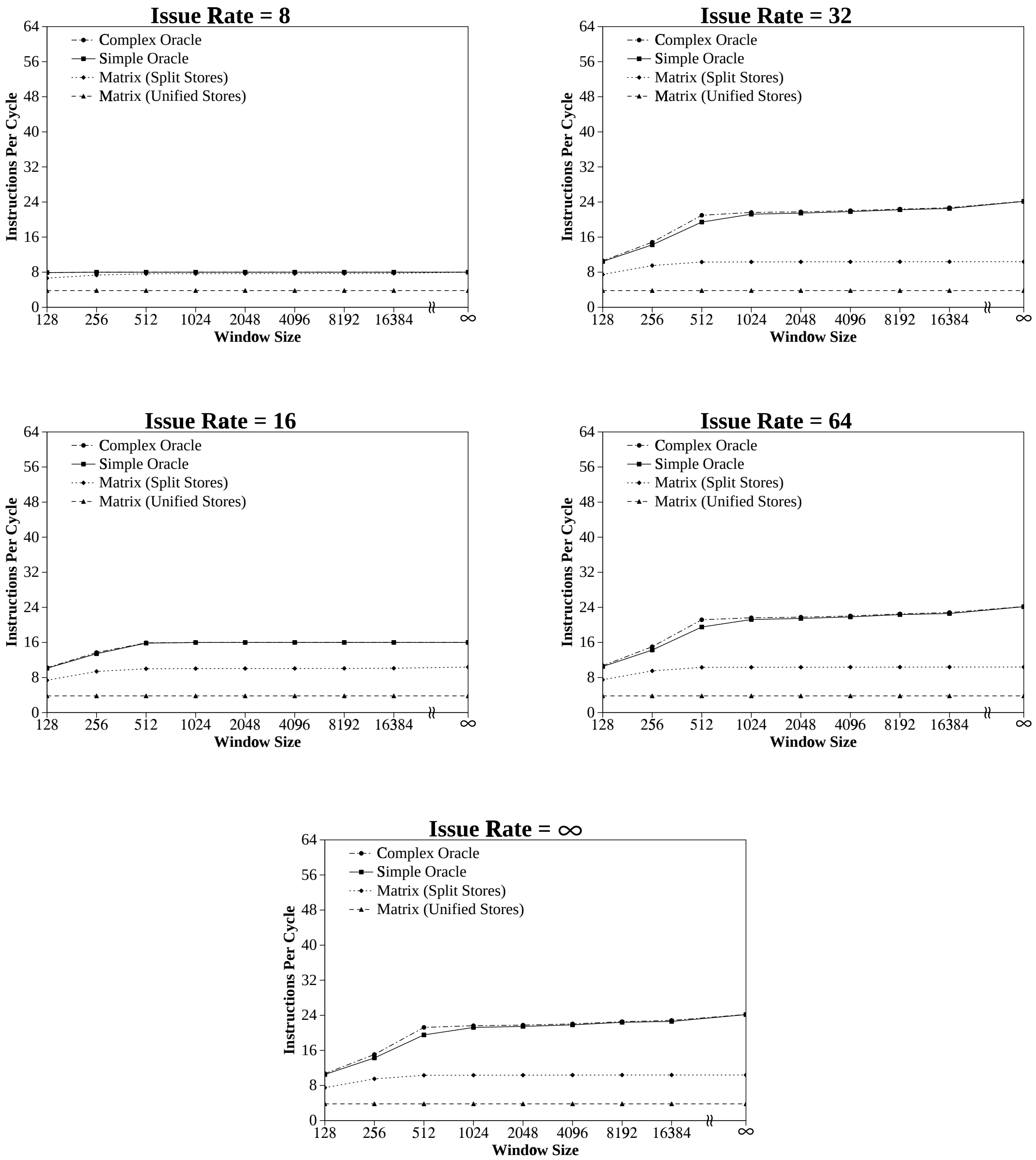


Figure A.4: Memory Disambiguation Techniques—Ijpeg



Issue Rate = 8

Issue Rate = 32

Issue Rate = 16

Issue Rate = 64

Issue Rate = ∞

Figure A.5: Memory Disambiguation Techniques—Li

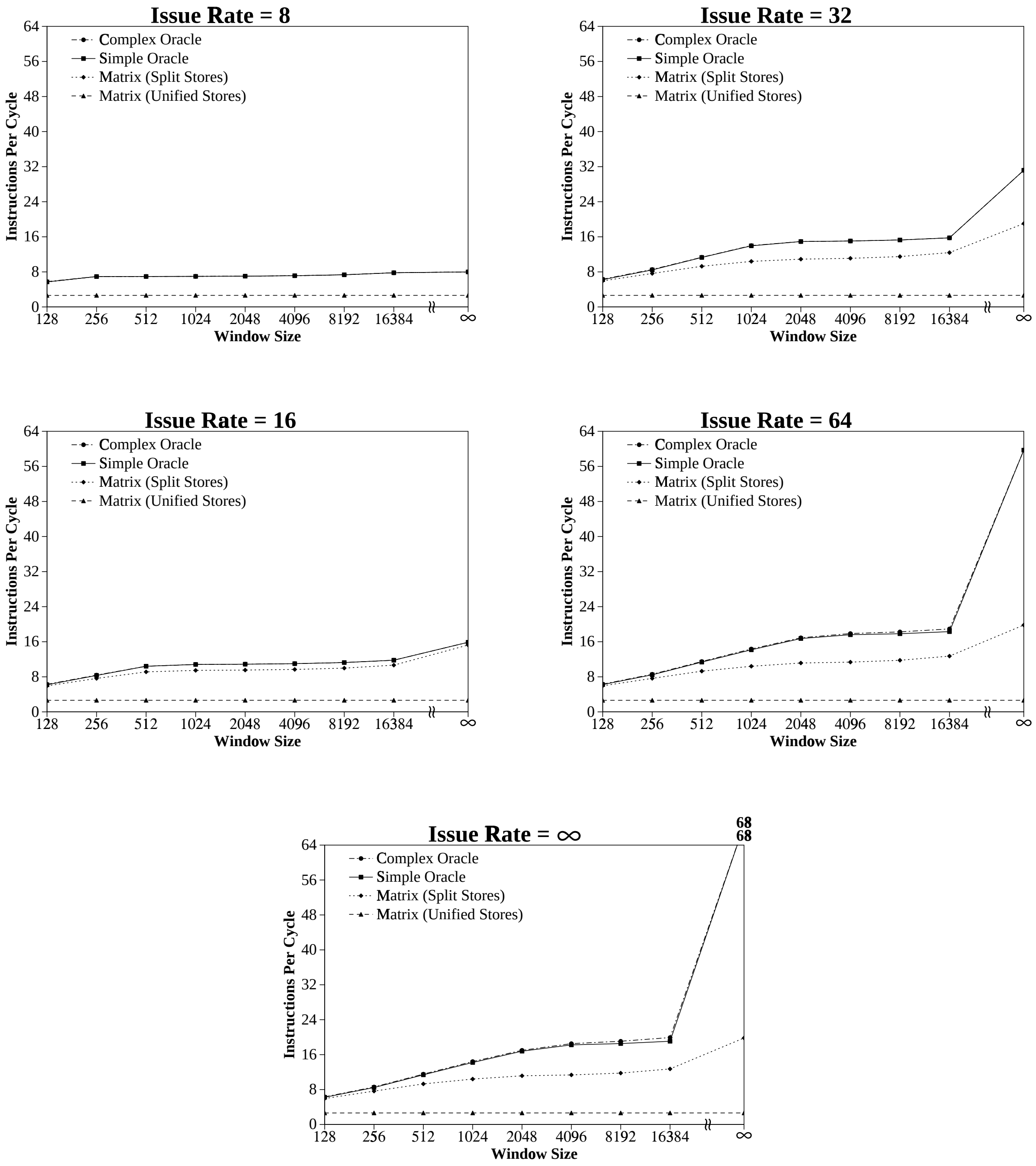
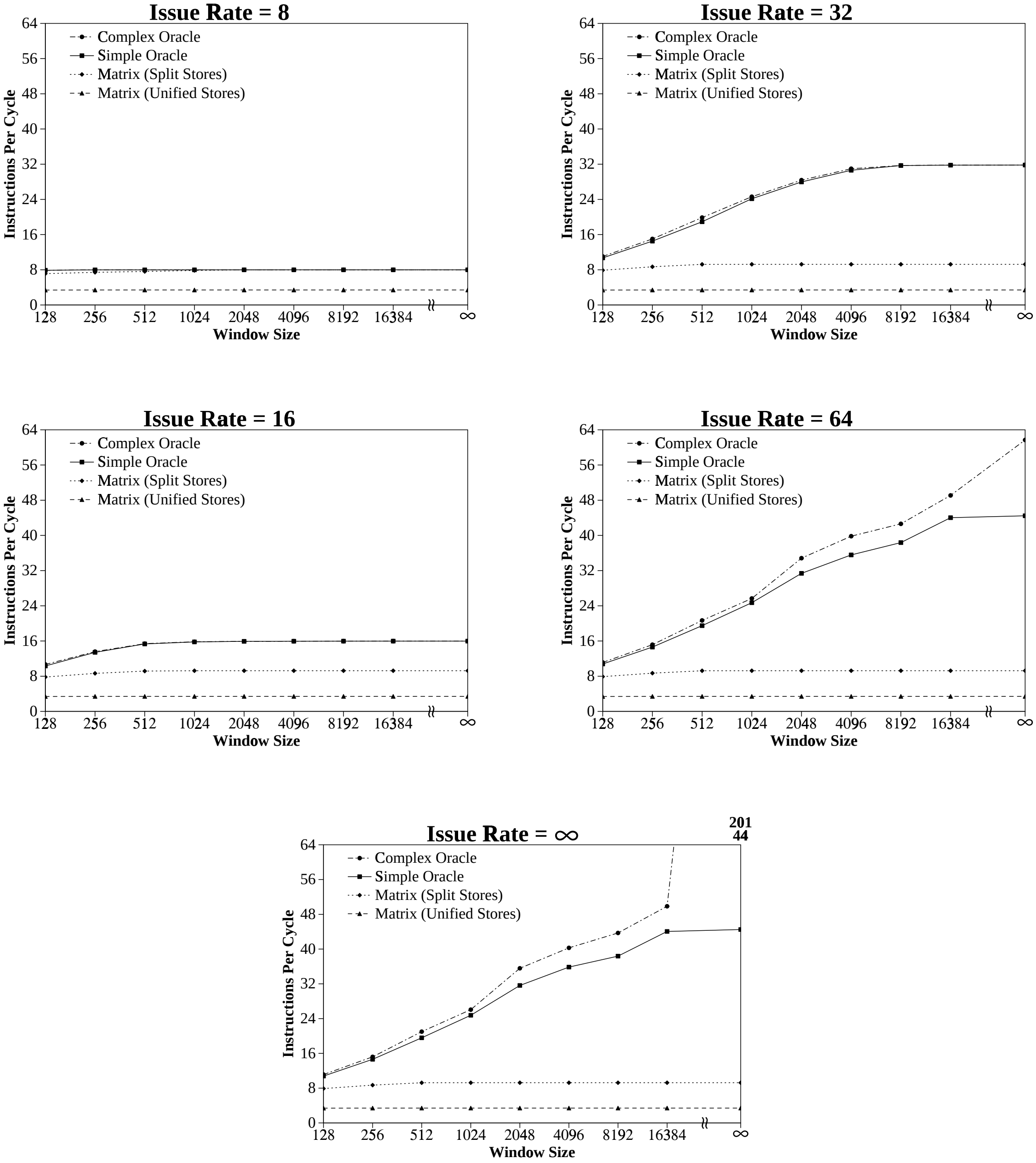


Figure A.6: Memory Disambiguation Techniques—M88k


Issue Rate = 8

Issue Rate = 32

Issue Rate = 16

Issue Rate = 64

Issue Rate = ∞

Figure A.7: Memory Disambiguation Techniques—Perl

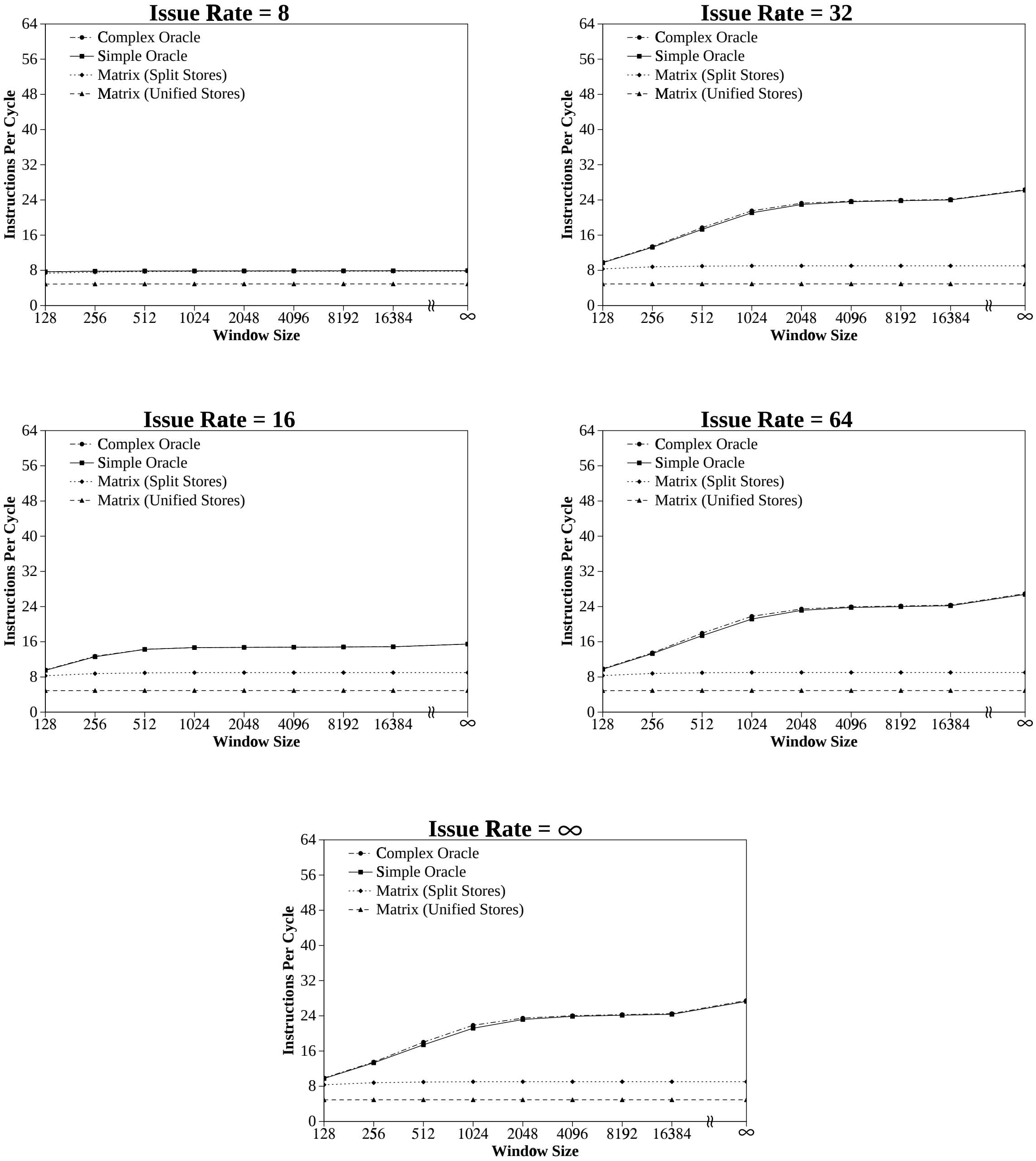


Figure A.8: Memory Disambiguation Techniques—Vortex

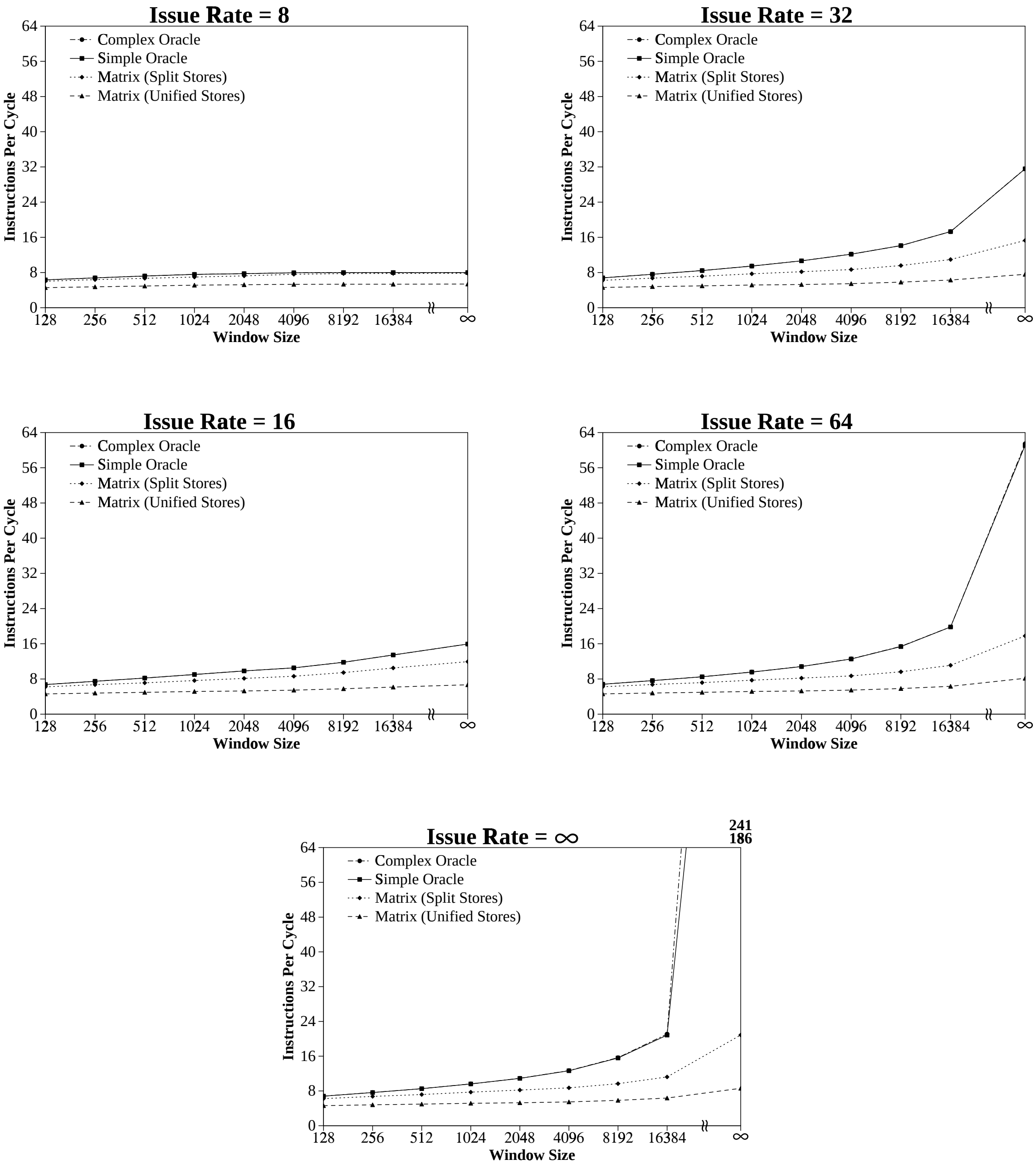


Figure A.9: Memory Disambiguation Techniques—Chess

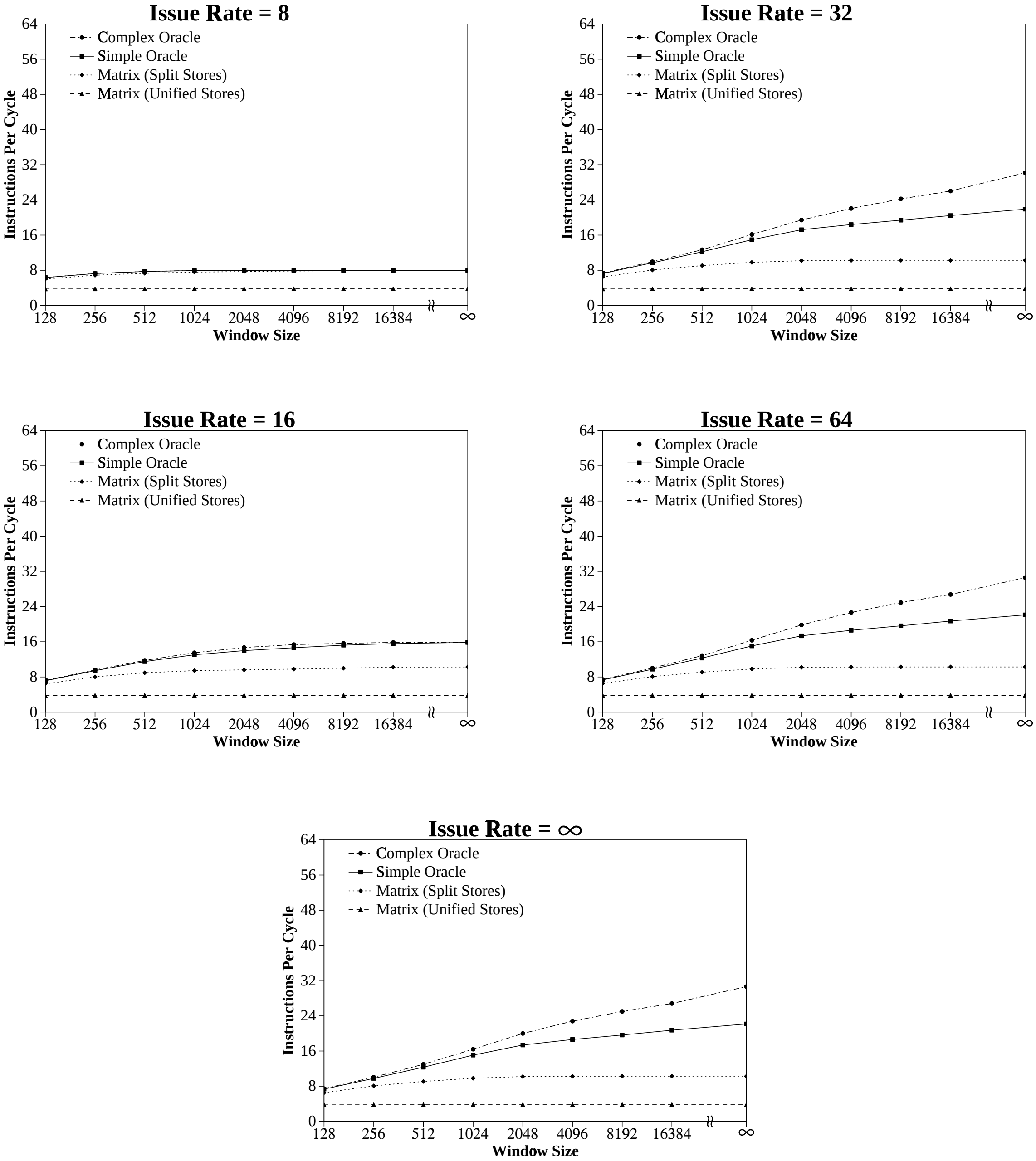


Figure A.10: Memory Disambiguation Techniques—Groff

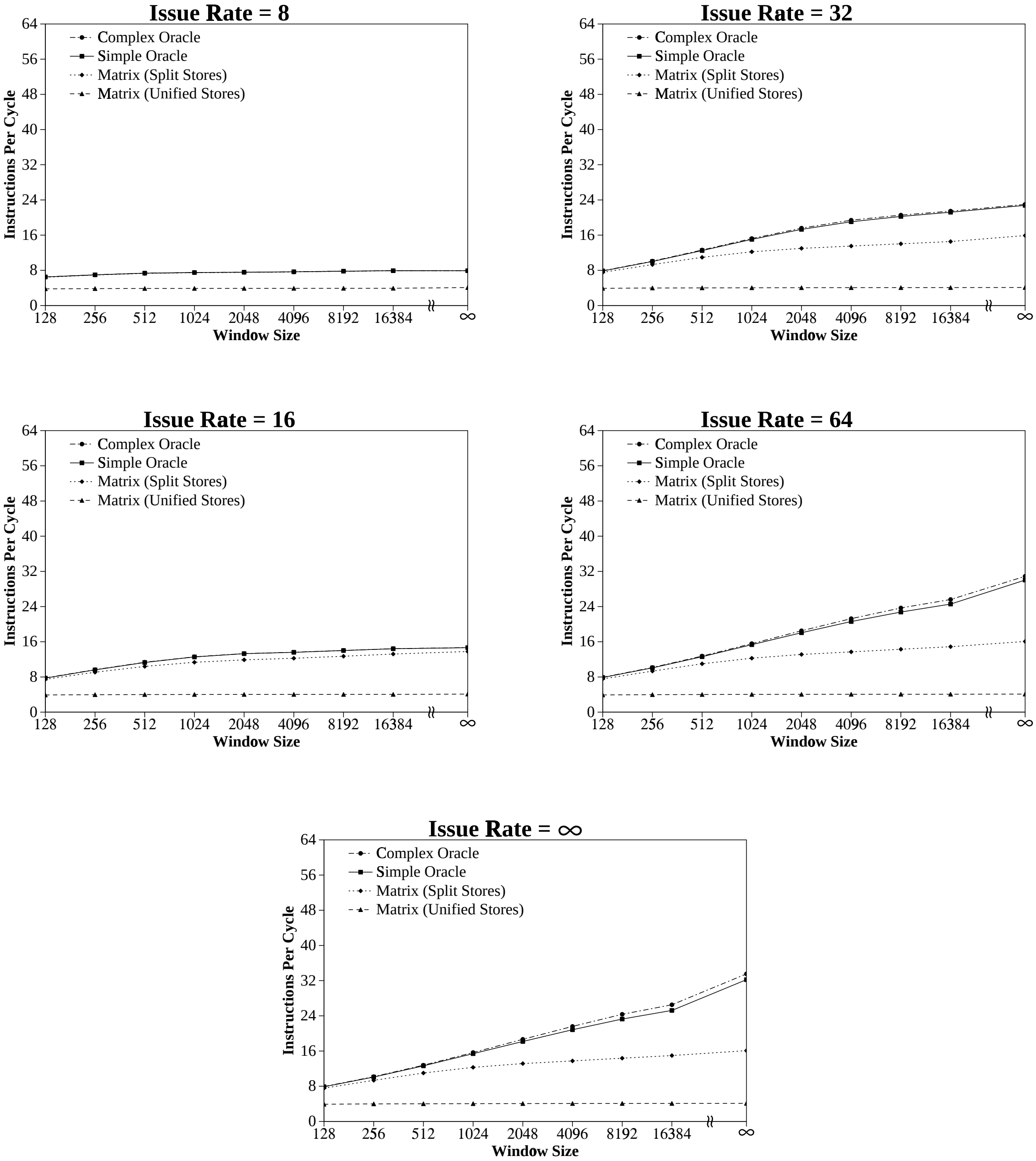


Figure A.11: Memory Disambiguation Techniques—Gs

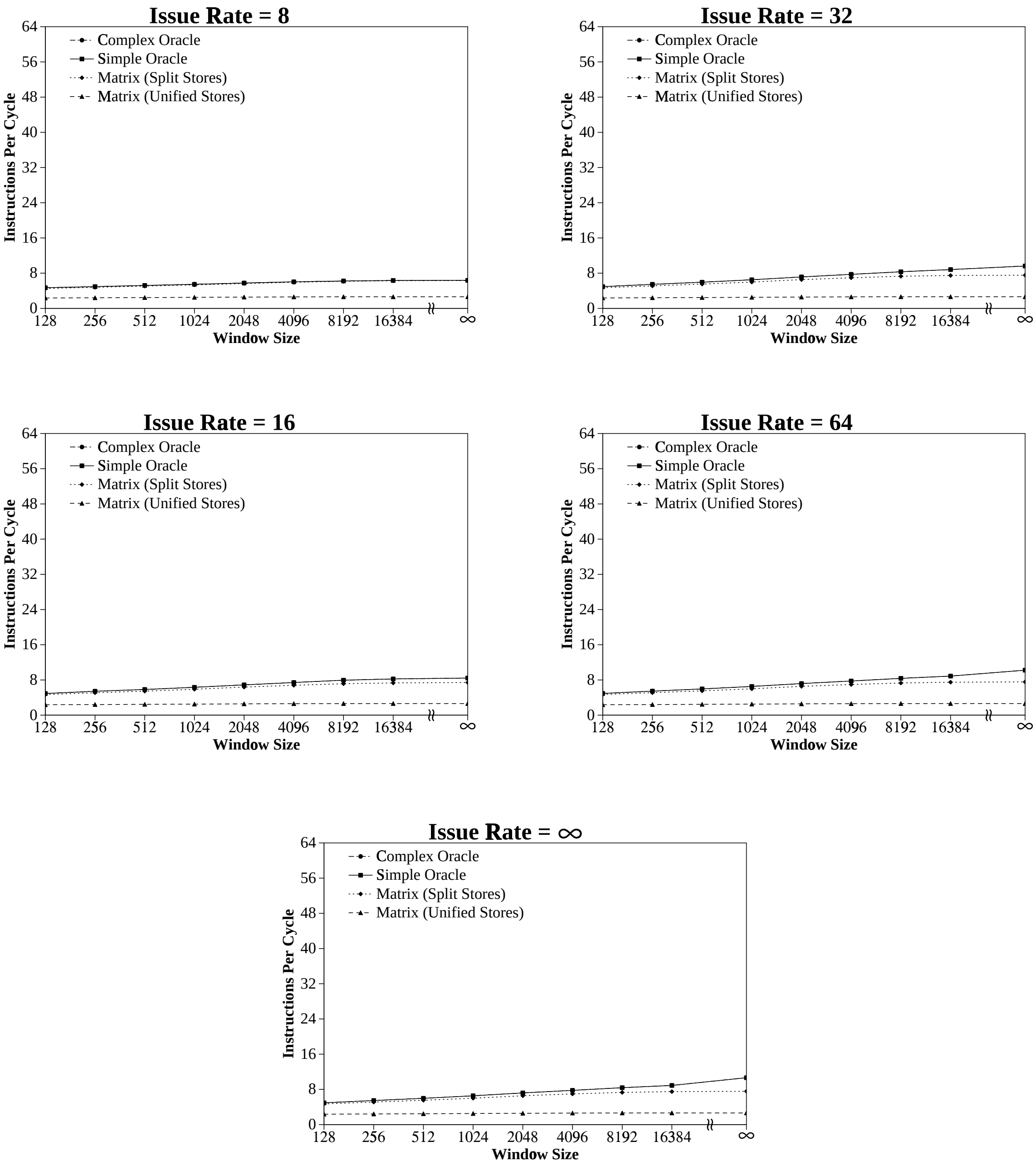


Figure A.12: Memory Disambiguation Techniques—Pgp

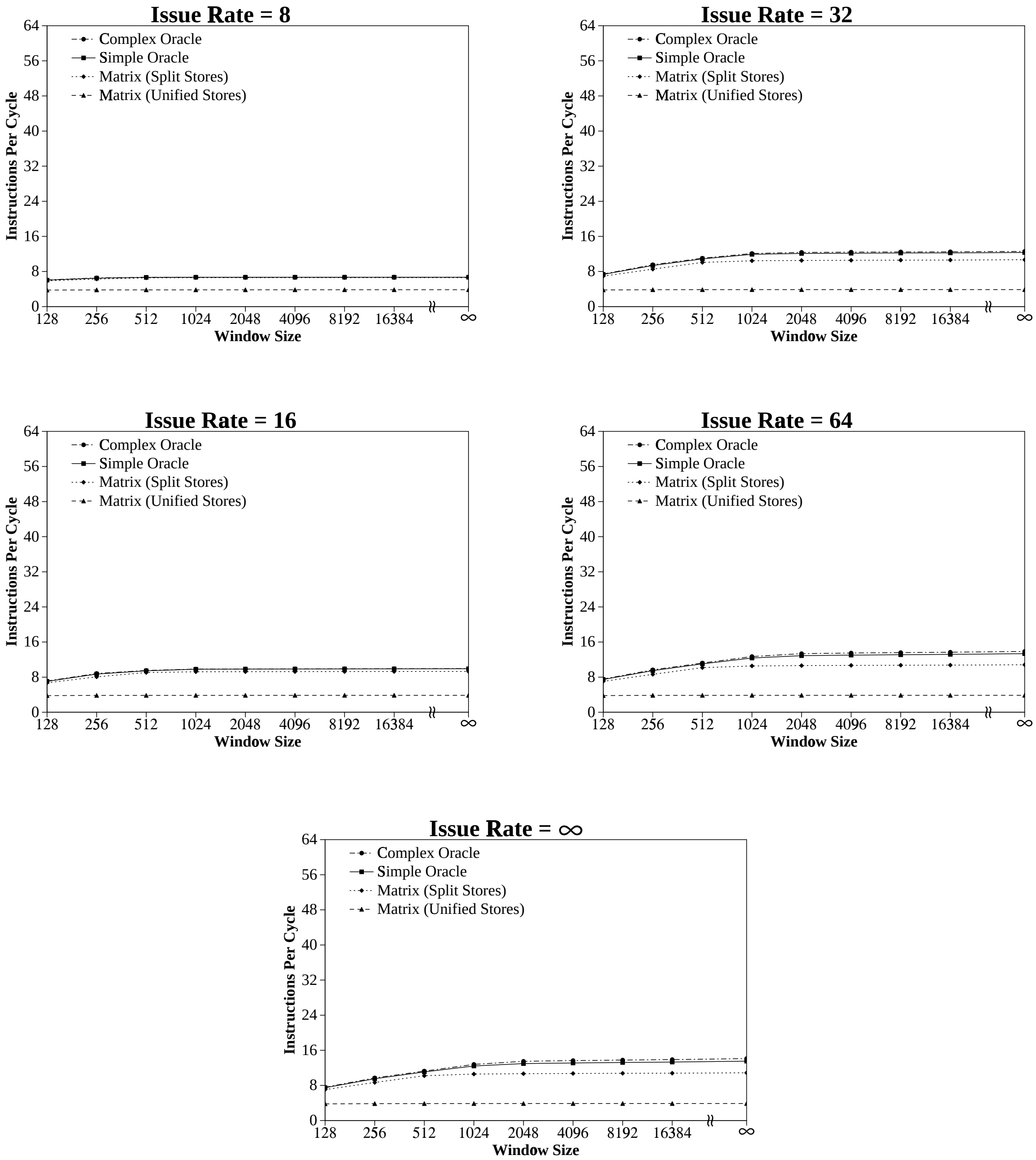


Figure A.13: Memory Disambiguation Techniques—Plot

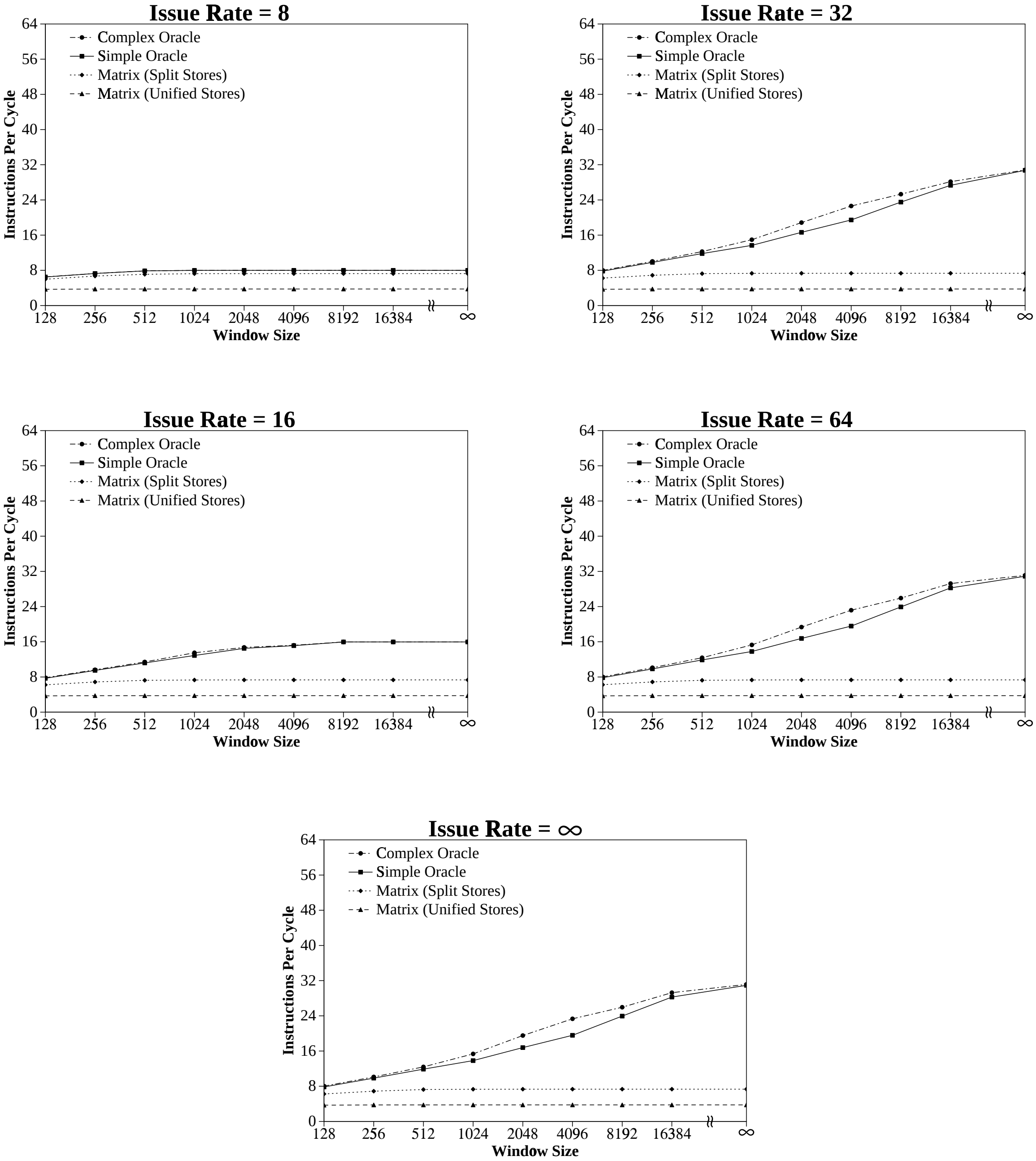


Figure A.14: Memory Disambiguation Techniques—Python

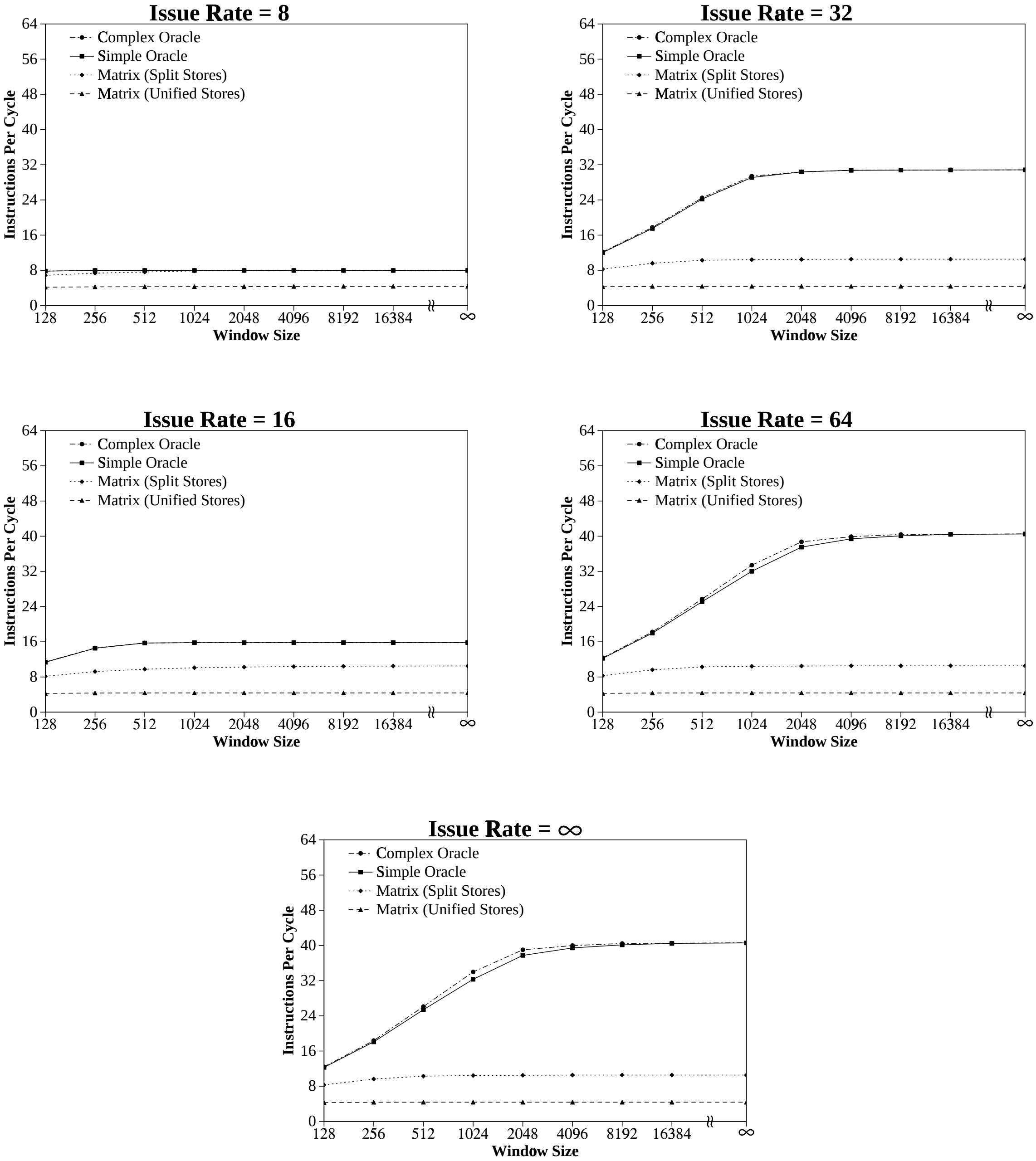


Figure A.15: Memory Disambiguation Techniques—Ss

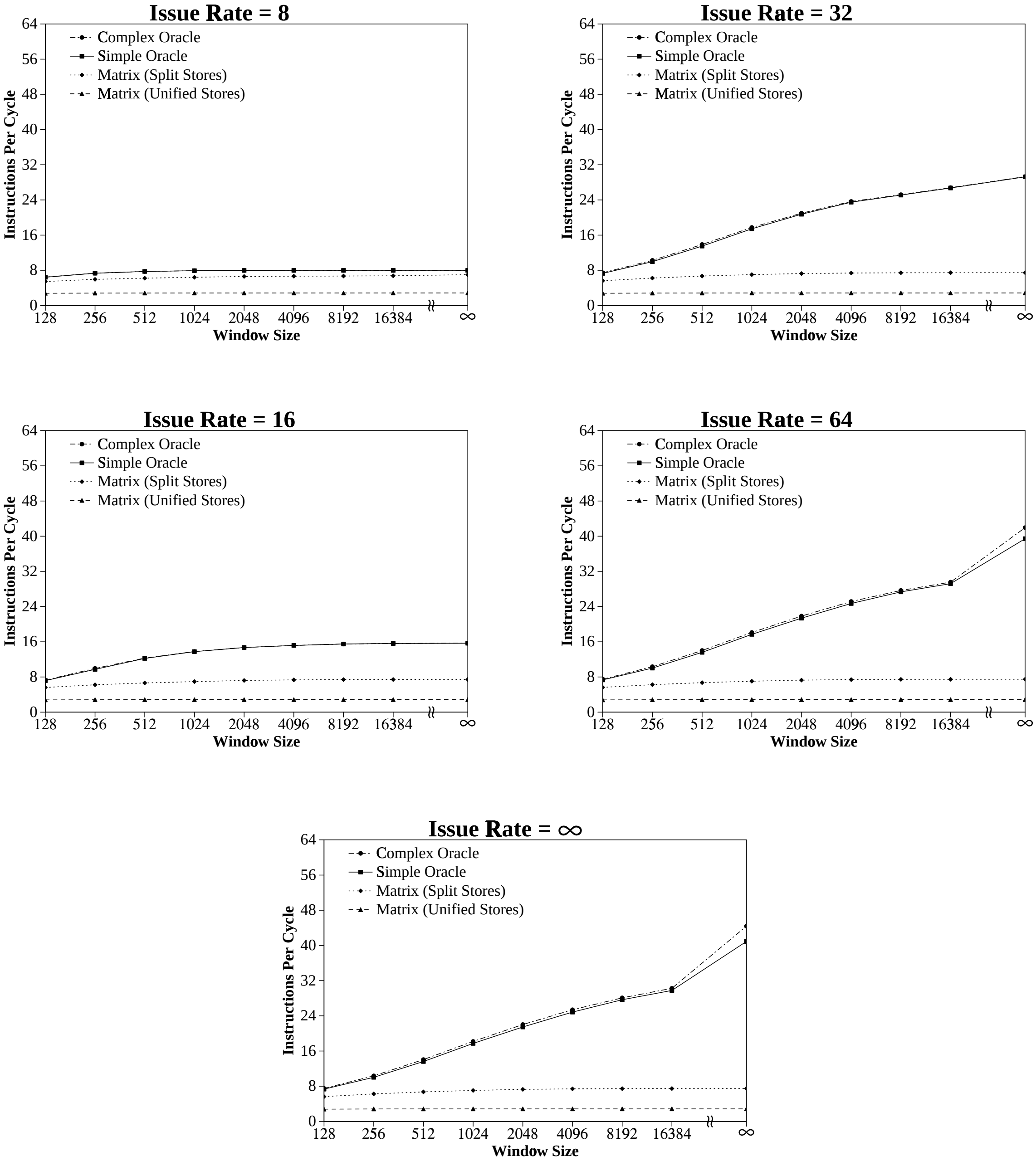
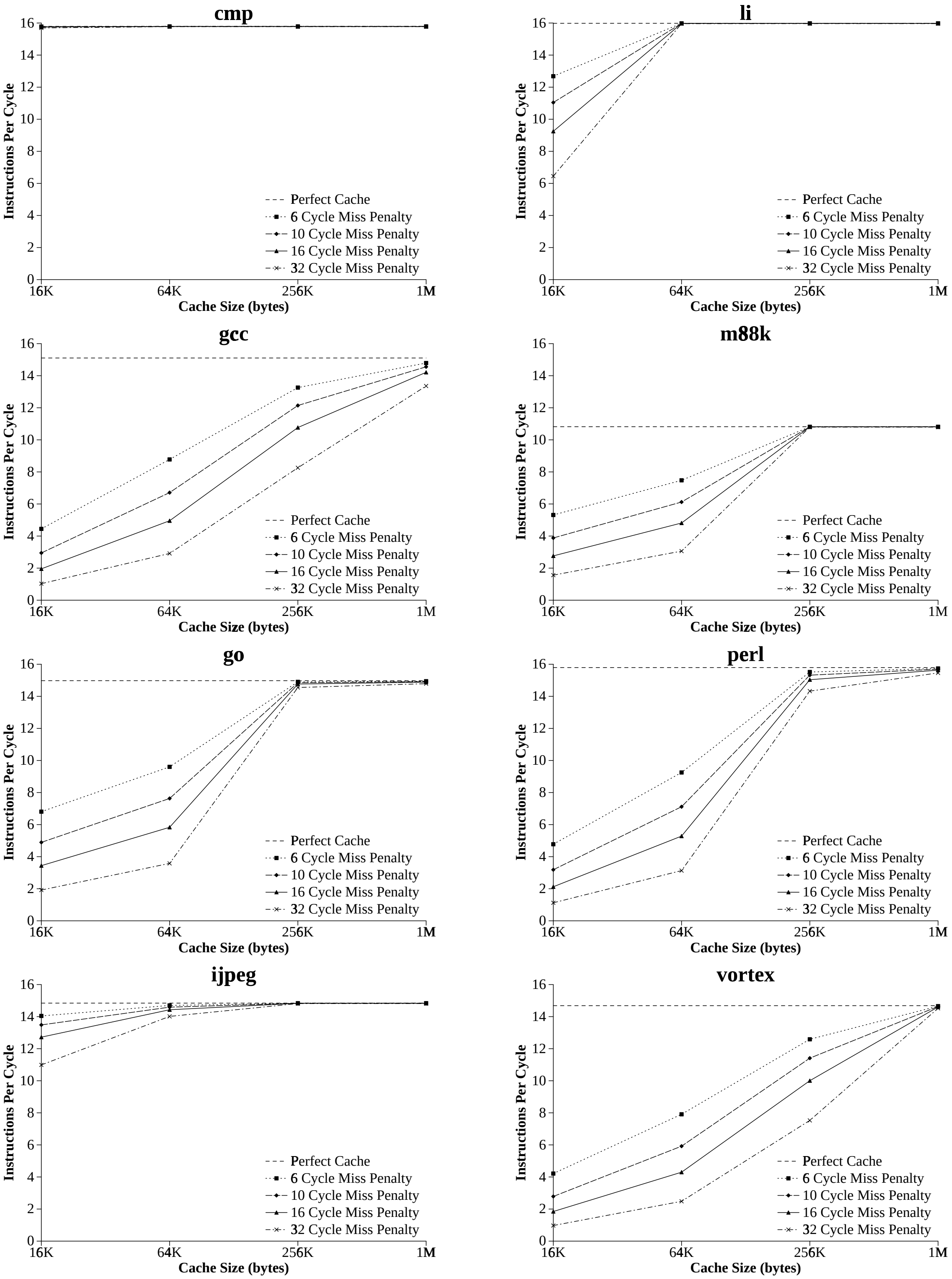


Figure A.16: Memory Disambiguation Techniques—Tex


cmp

li

gcc

m88k

go

perl

ijpeg

vortex

Figure A.17: Ideal Machine with Varied Instruction Cache Size—SPEC Benchmarks

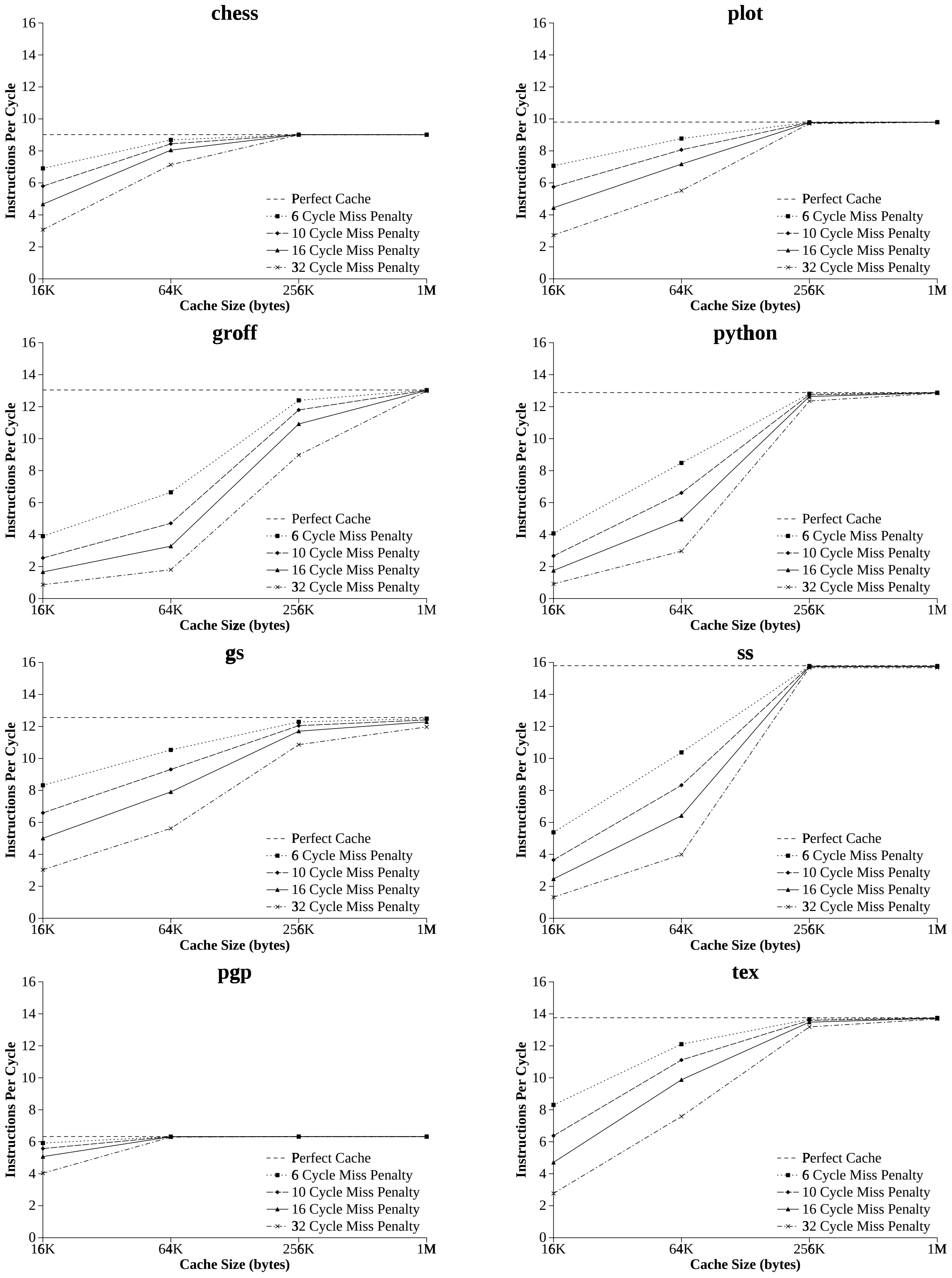


Figure A.18: Ideal Machine with Varied Instruction Cache Size—Non-SPEC Benchmarks

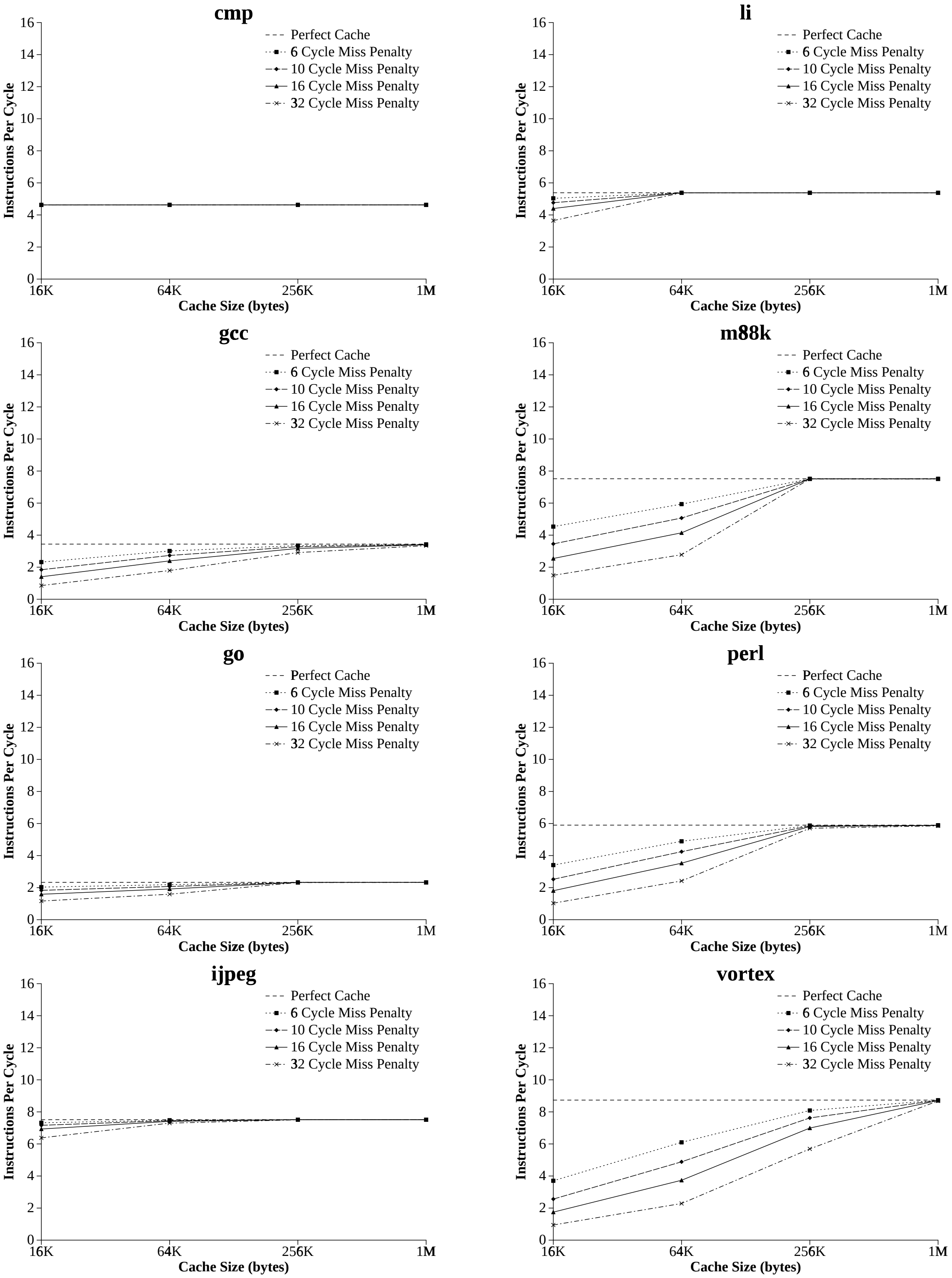


Figure A.19: Real Machine with Varied Instruction Cache Size (Constant Mispredict Penalty)—SPEC Benchmarks

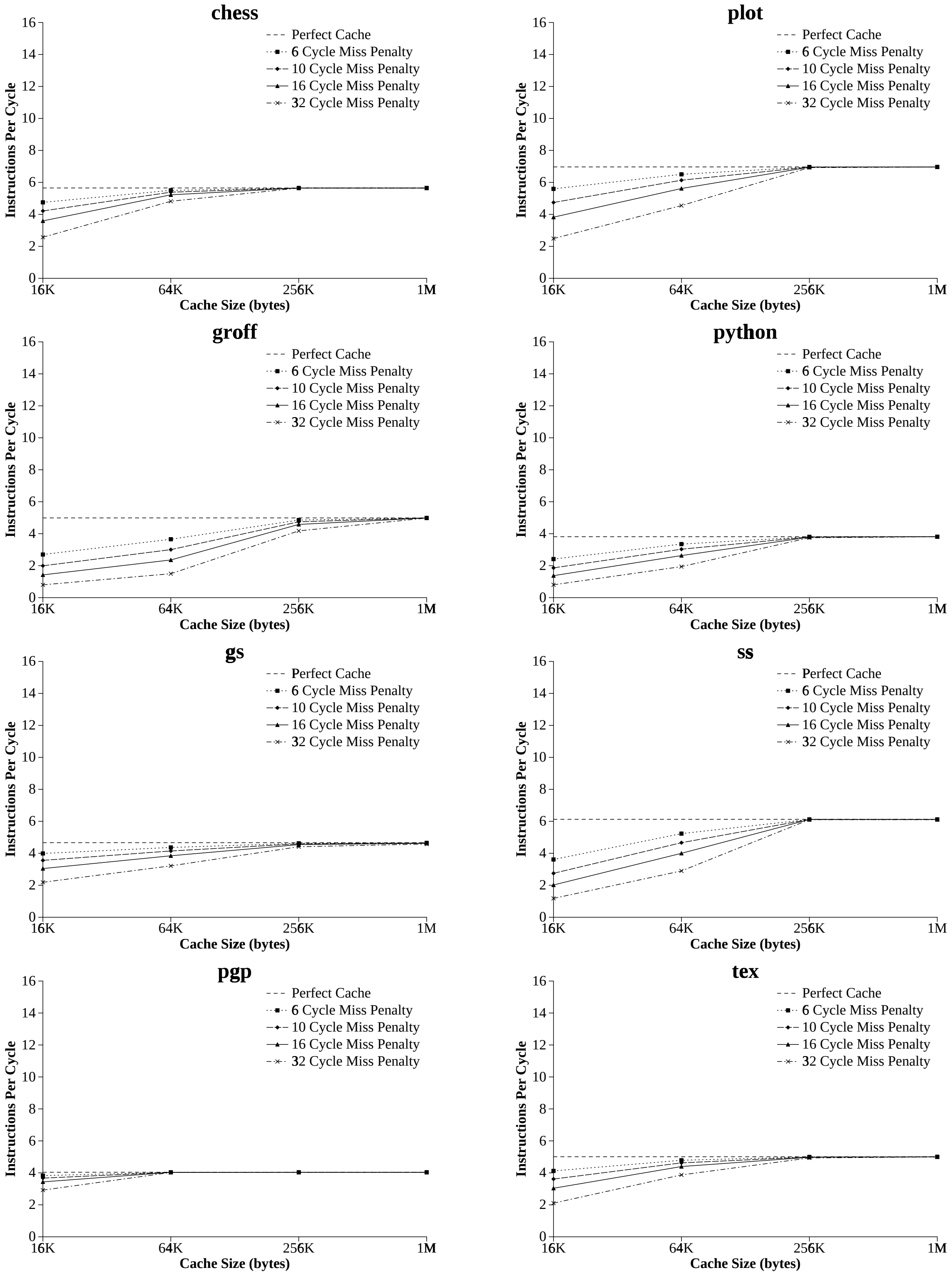


Figure A.20: Real Machine with Varied Instruction Cache Size (Constant Mispredict Penalty)—Non-SPEC Benchmarks

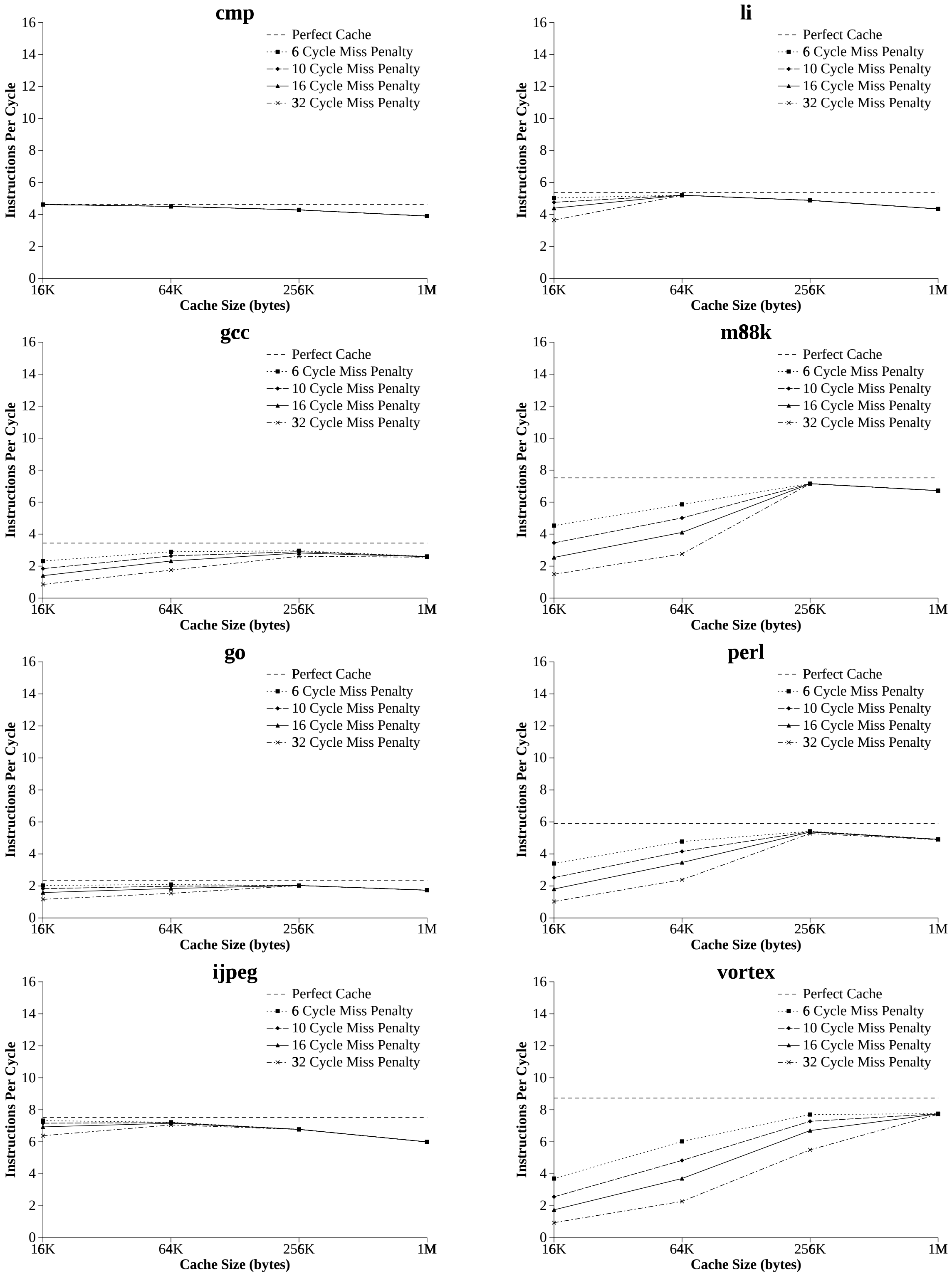


Figure A.21: Real Machine with Varied Instruction Cache Size (Scaled Mispredict Penalty)—SPEC Benchmarks

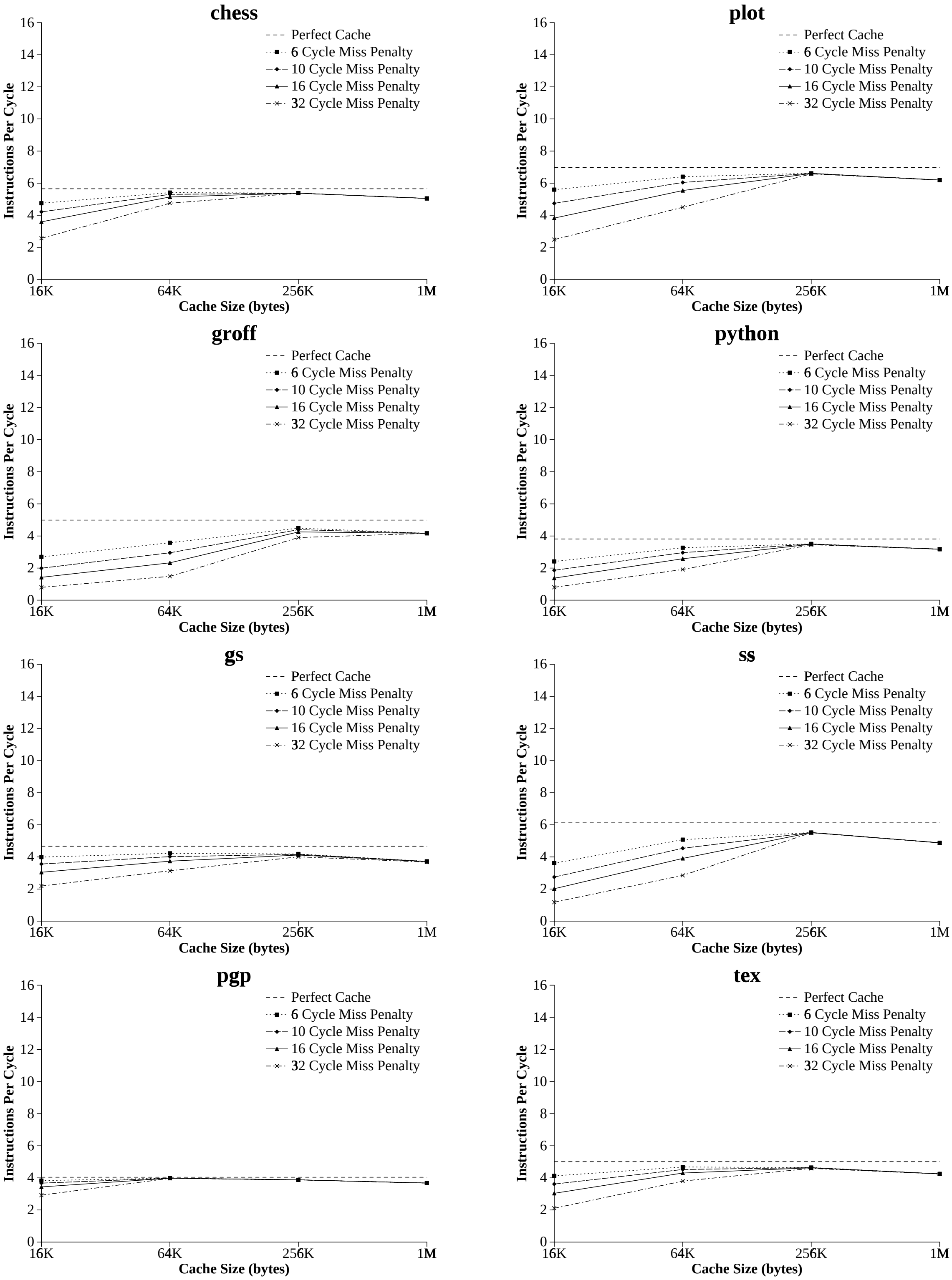


Figure A.22: Real Machine with Varied Instruction Cache Size (Scaled Mispredict Penalty)—Non-SPEC Benchmarks

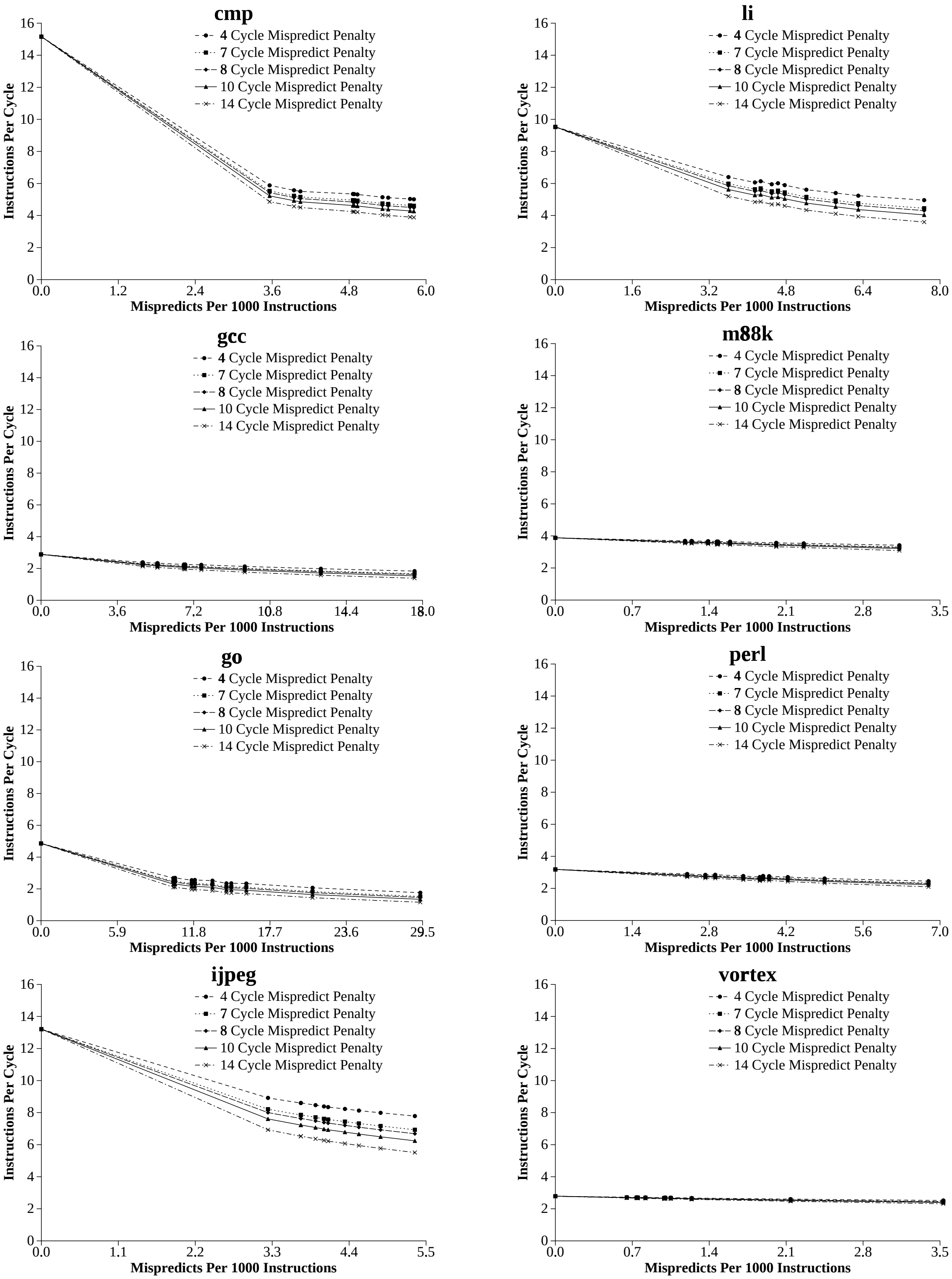


Figure A.23: Real Machine withVaried Mispredict Rate—SPEC Benchmarks

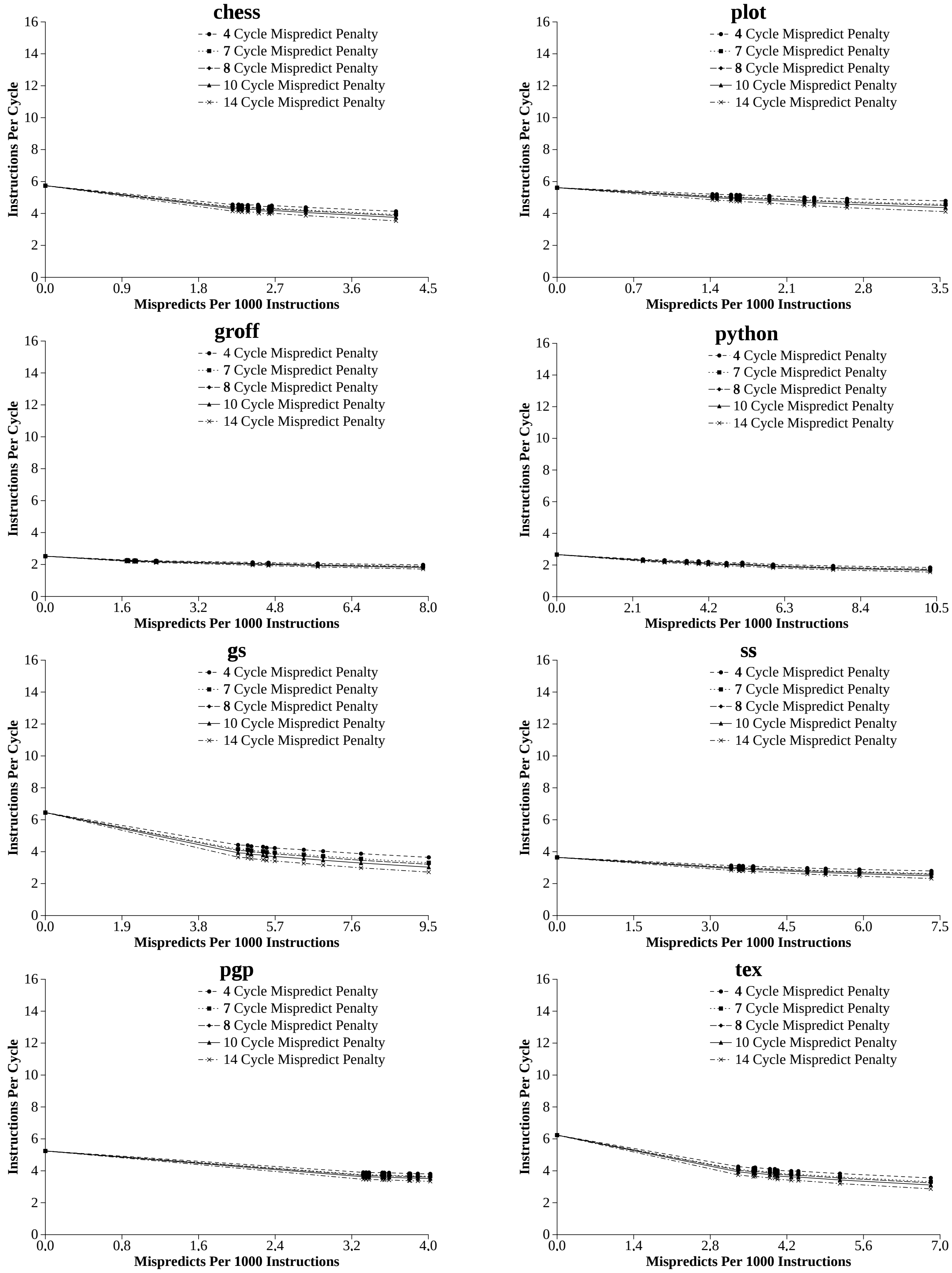
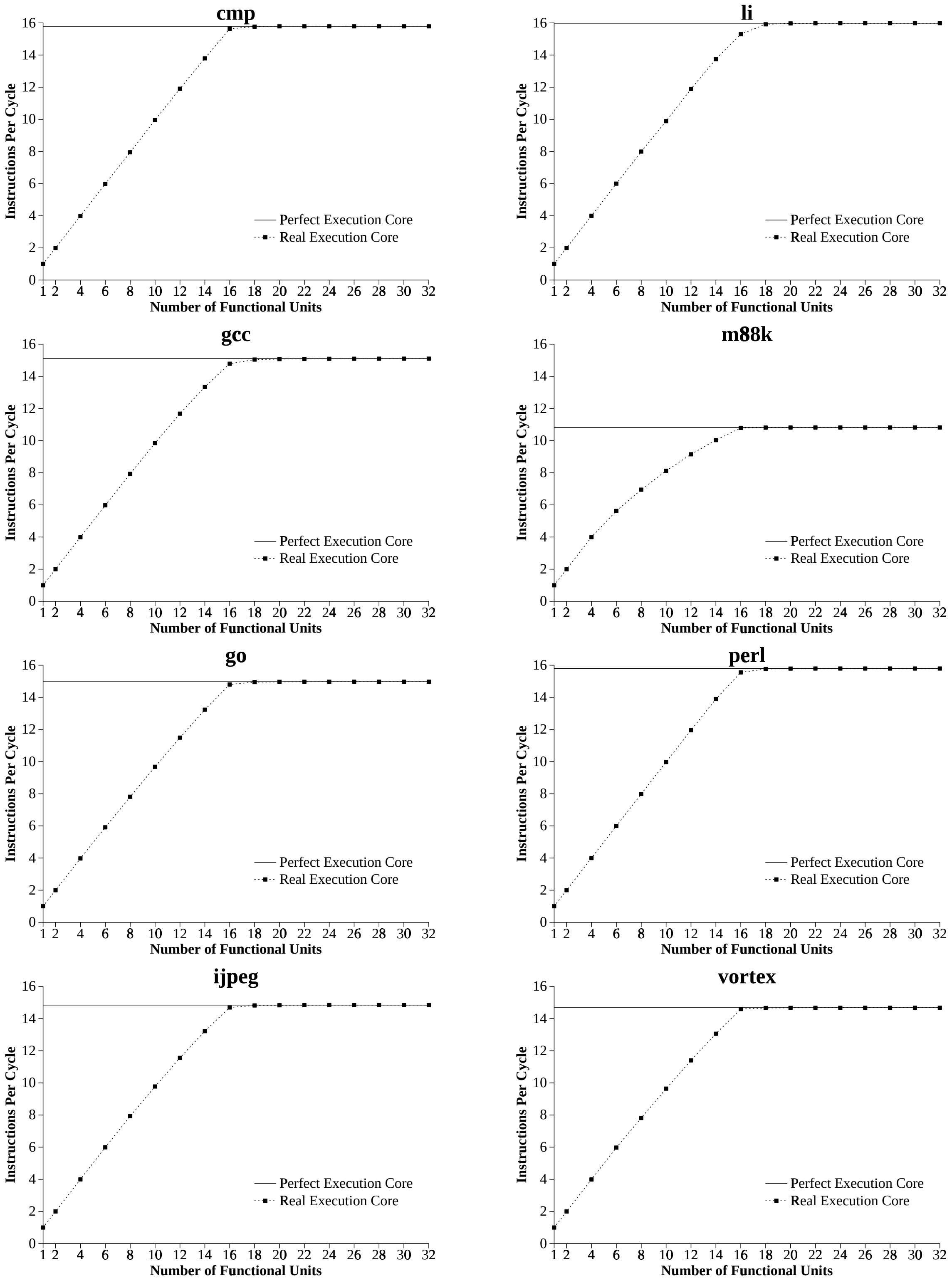


Figure A.24: Real Machine with Varied Mispredict Rate—Non-SPEC Benchmarks


cmp

li

gcc

m88k

go

perl

ijpeg

vortex

Figure A.25: Ideal Machine with Varied Execution Core Size—SPEC Benchmarks

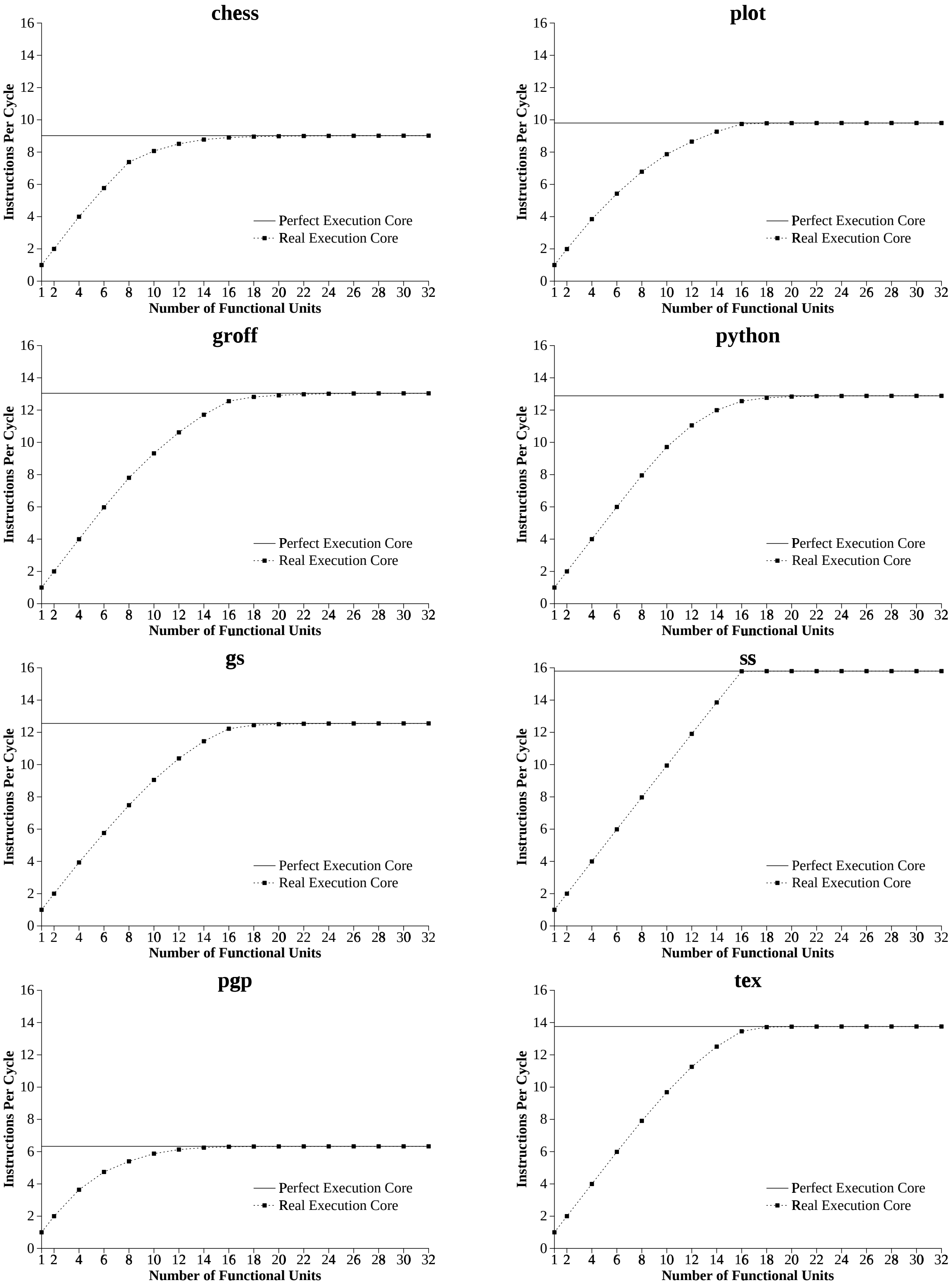


Figure A.26: Ideal Machine with Varied Execution Core Size—Non-SPEC Benchmarks

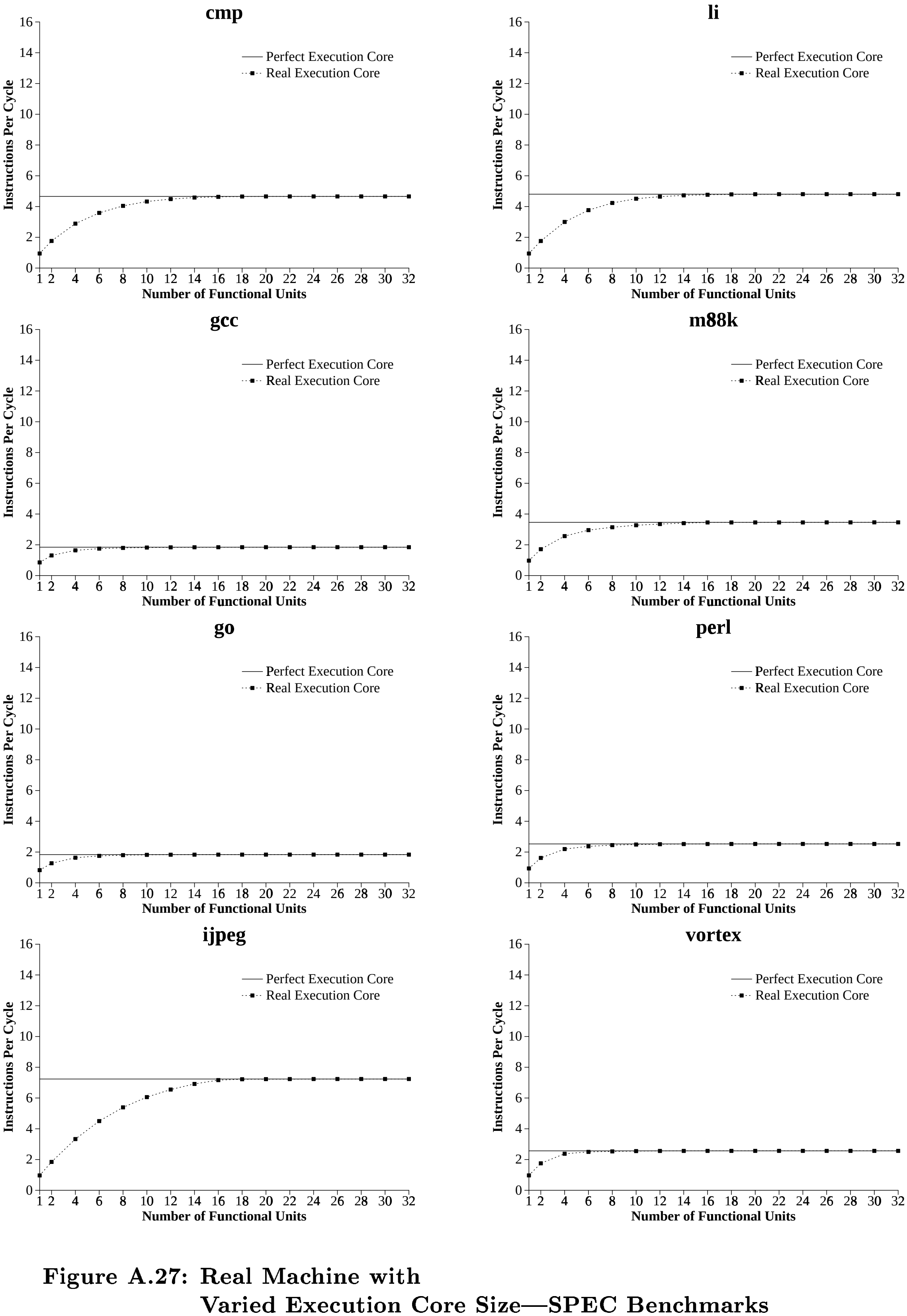


Figure A.27: Real Machine with Varied Execution Core Size—SPEC Benchmarks

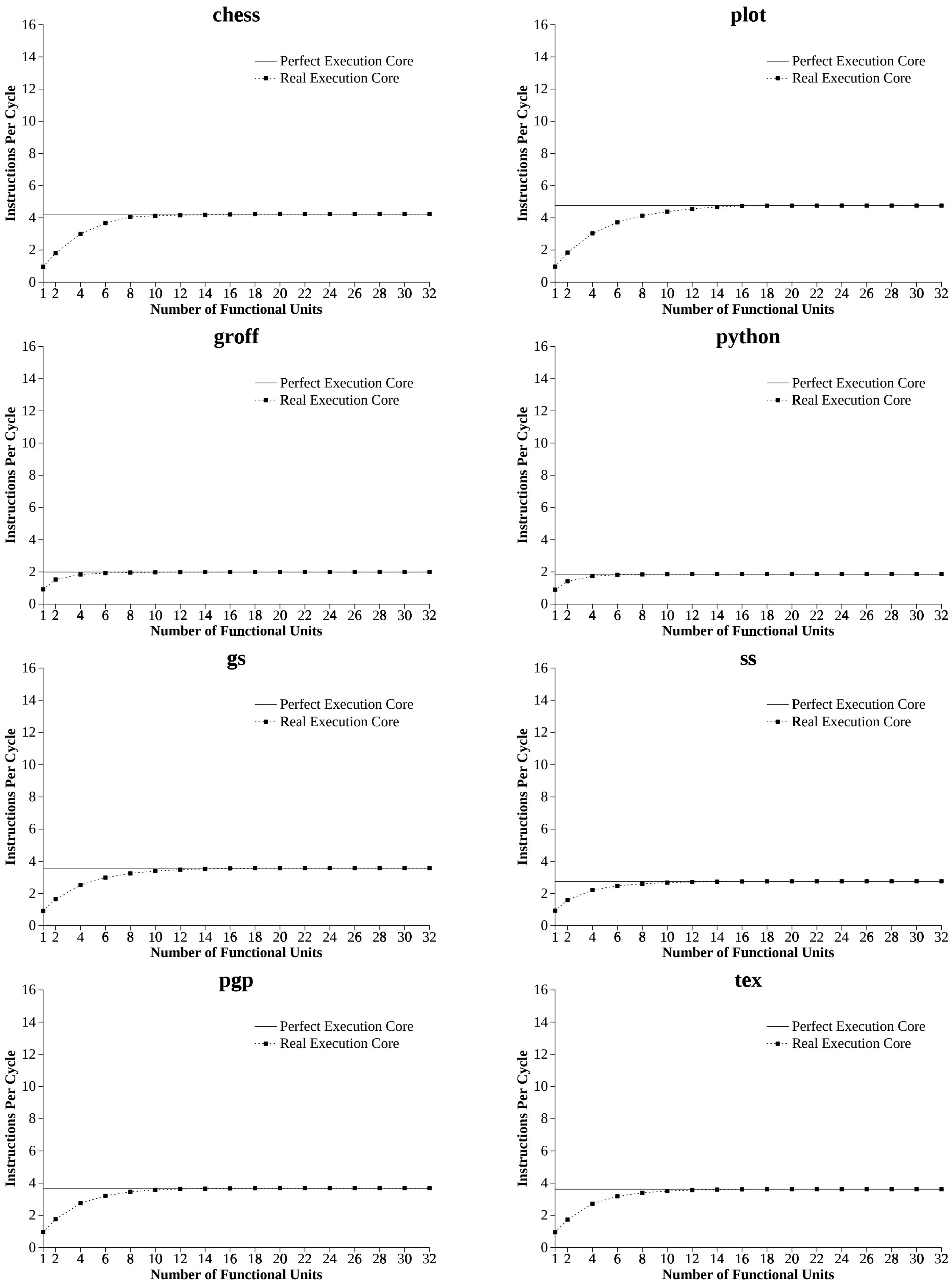


Figure A.28: Real Machine with Varied Execution Core Size—Non-SPEC Benchmarks

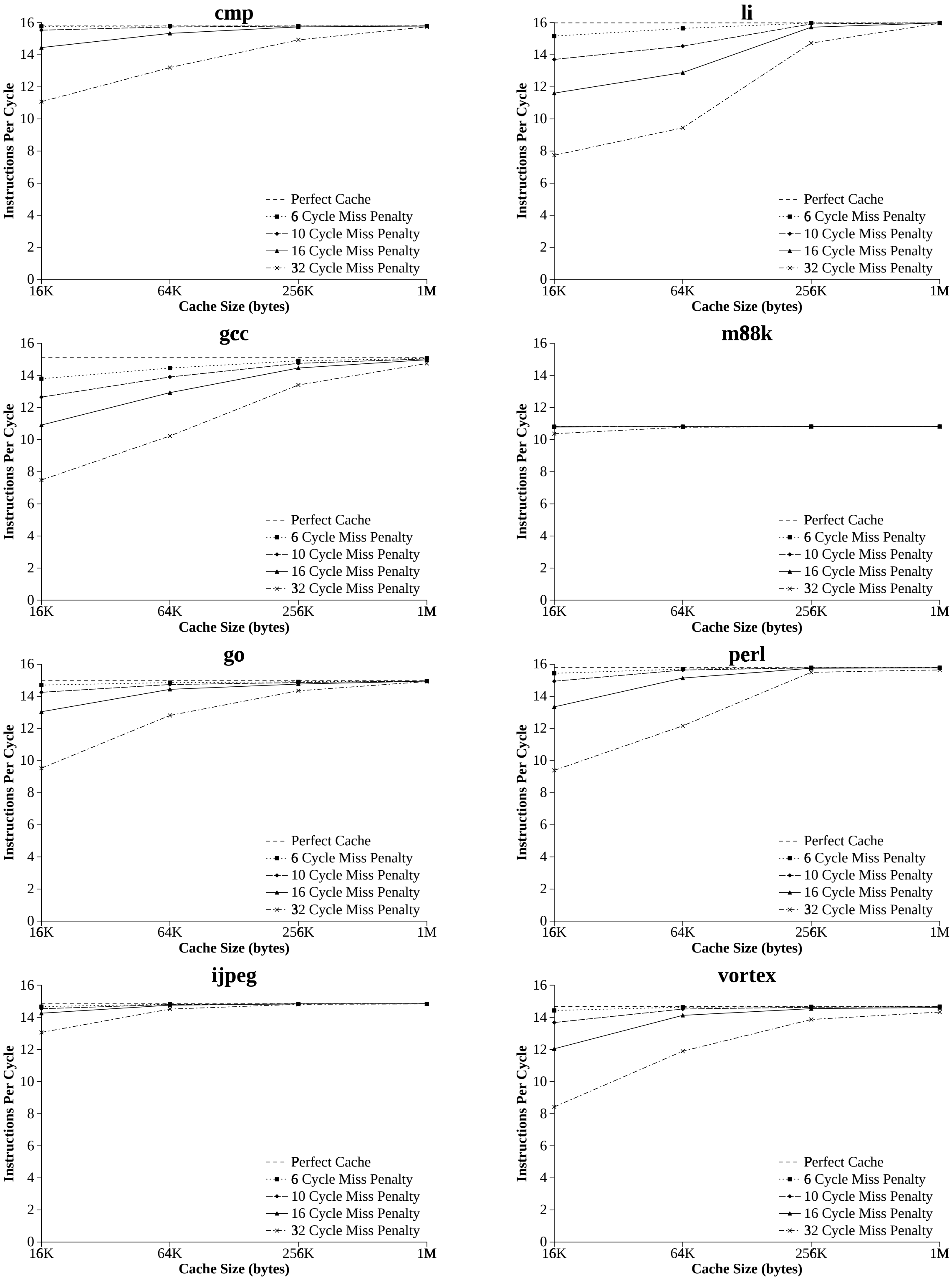
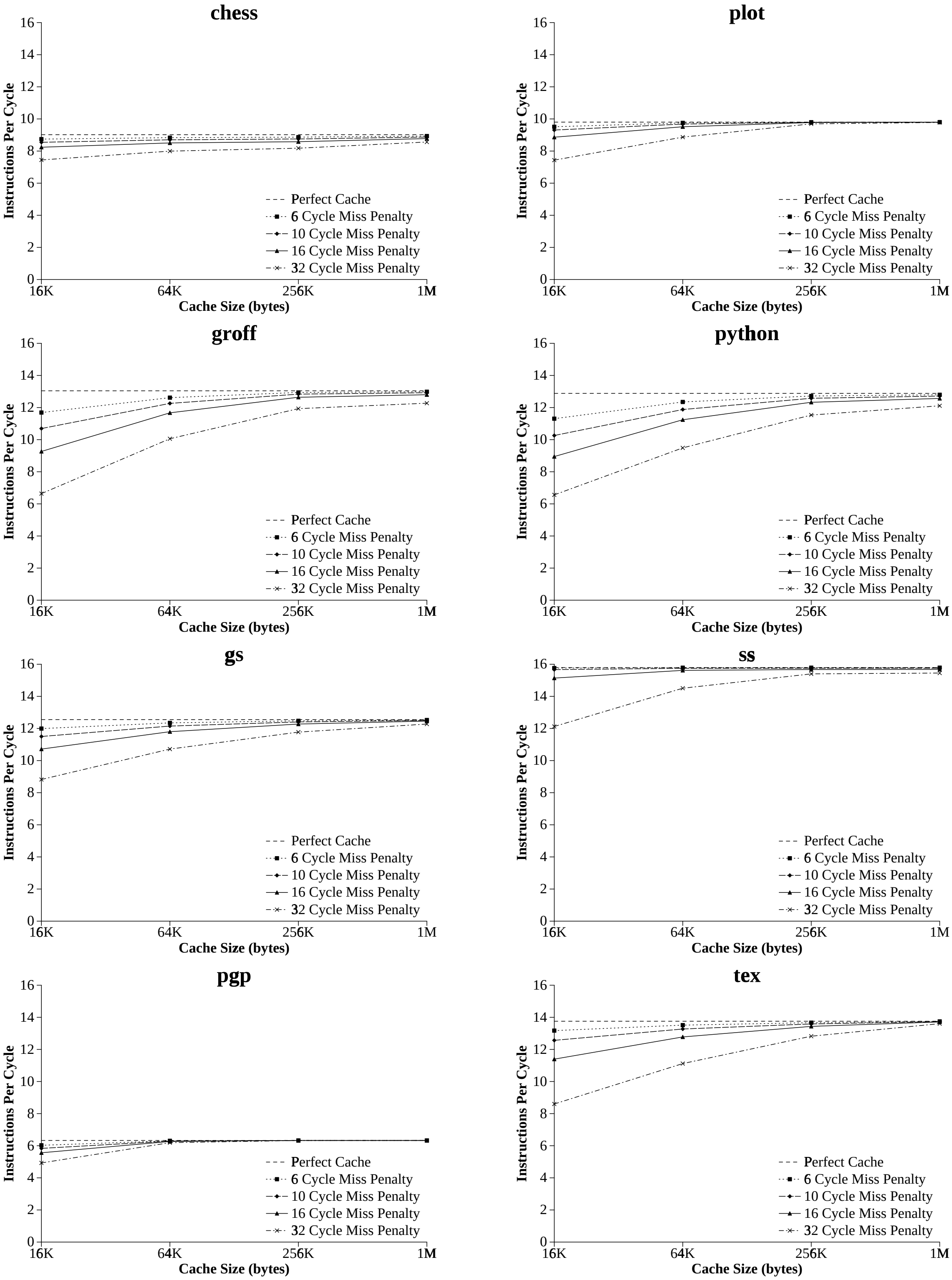


Figure A.29: Ideal Machine with Varied Data Cache Size (Constant Load Latency)—SPEC Benchmarks



chess

plot

groff

python

gs

ss

pgp

tex

Figure A.30: Ideal Machine with Varied Data Cache Size (Constant Load Latency)—Non-SPEC Benchmarks

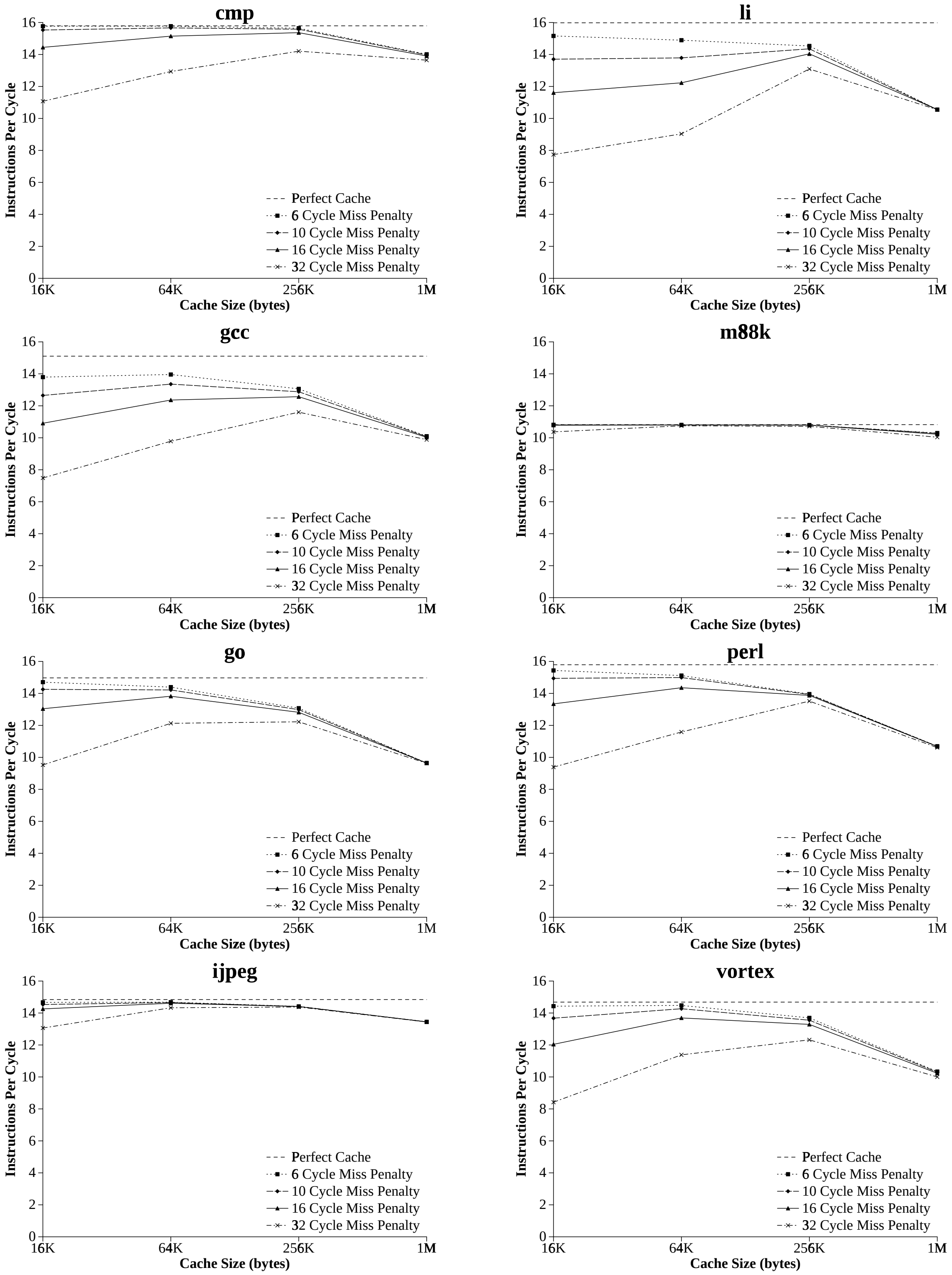


Figure A.31: Ideal Machine with Varied Data Cache Size (Scaled Load Latency)—SPEC Benchmarks

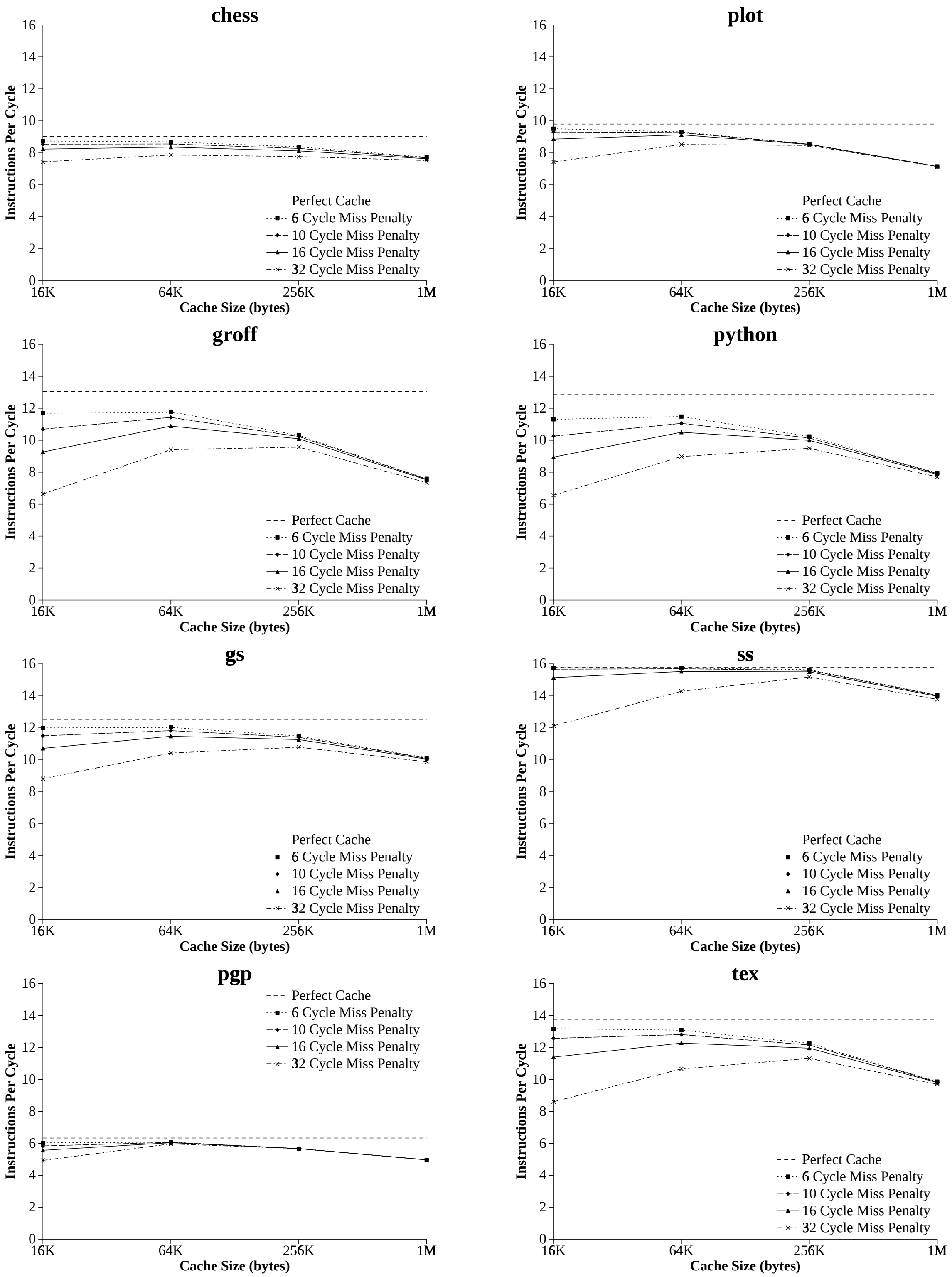


Figure A.32: Ideal Machine with Varied Data Cache Size (Scaled Load Latency)—Non-SPEC Benchmarks

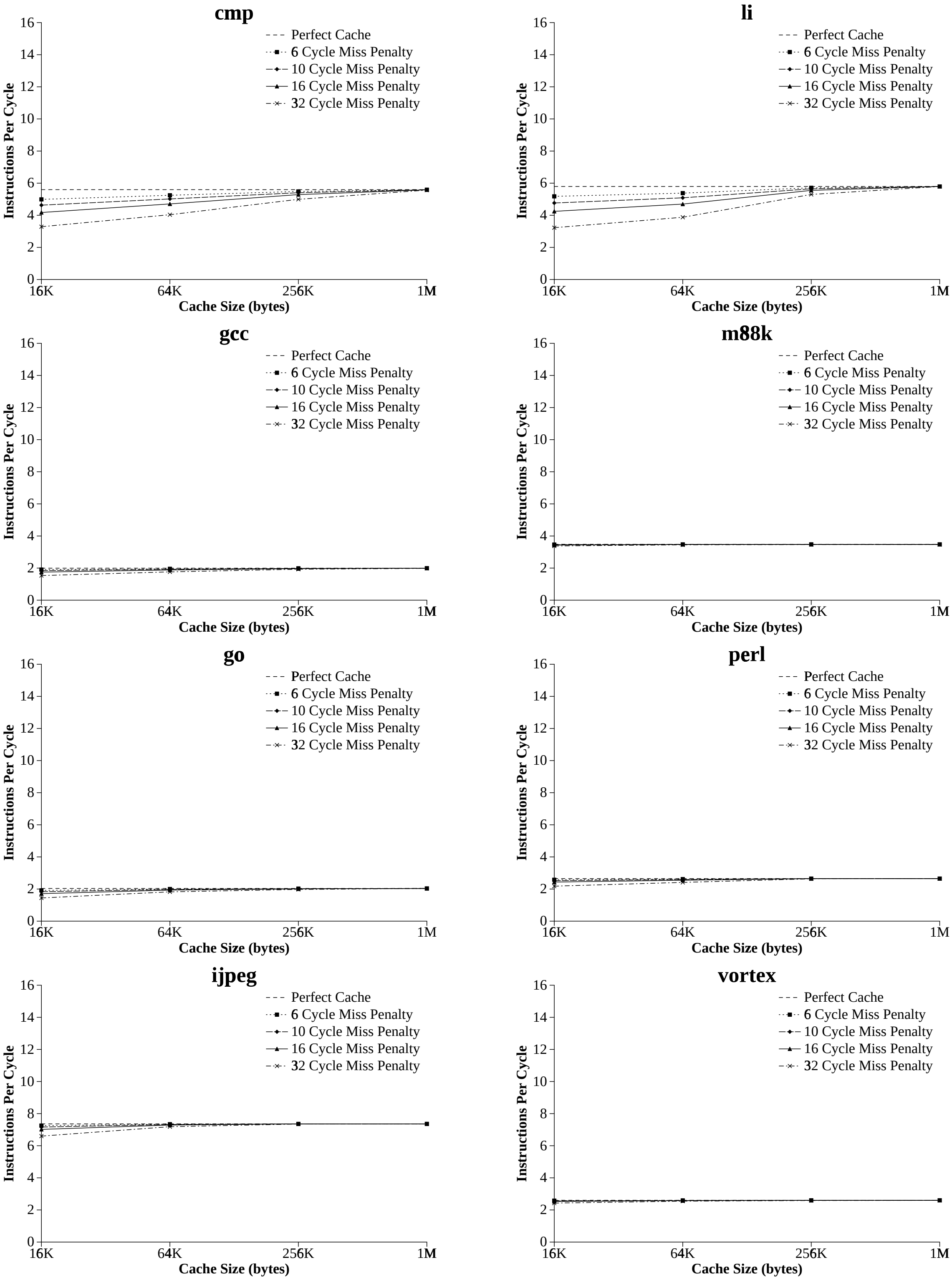


Figure A.33: Real Machine with Varied Data Cache Size (Constant Load Latency)—SPEC Benchmarks

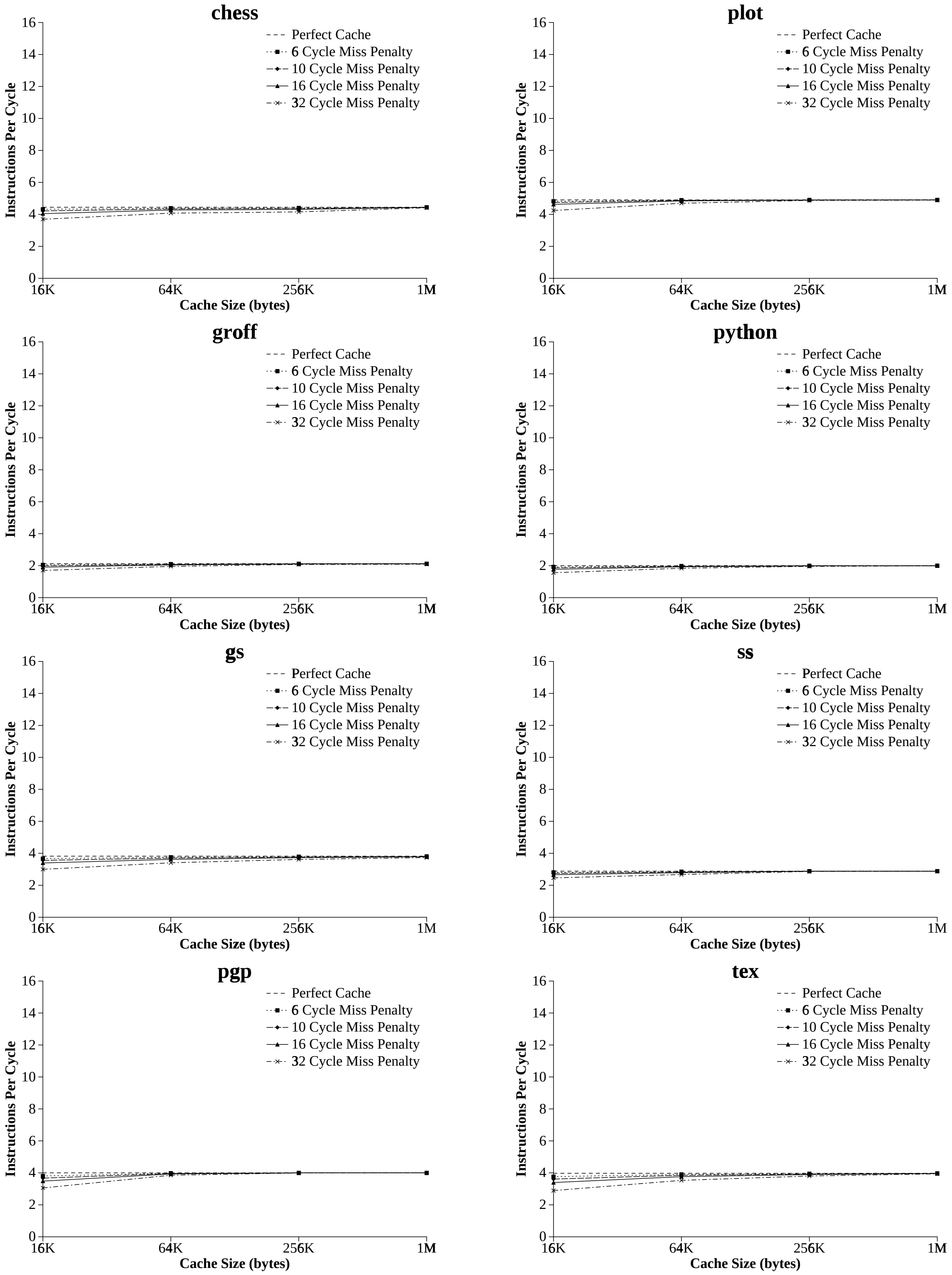


Figure A.34: Real Machine with Varied Data Cache Size (Constant Load Latency)—Non-SPEC Benchmarks

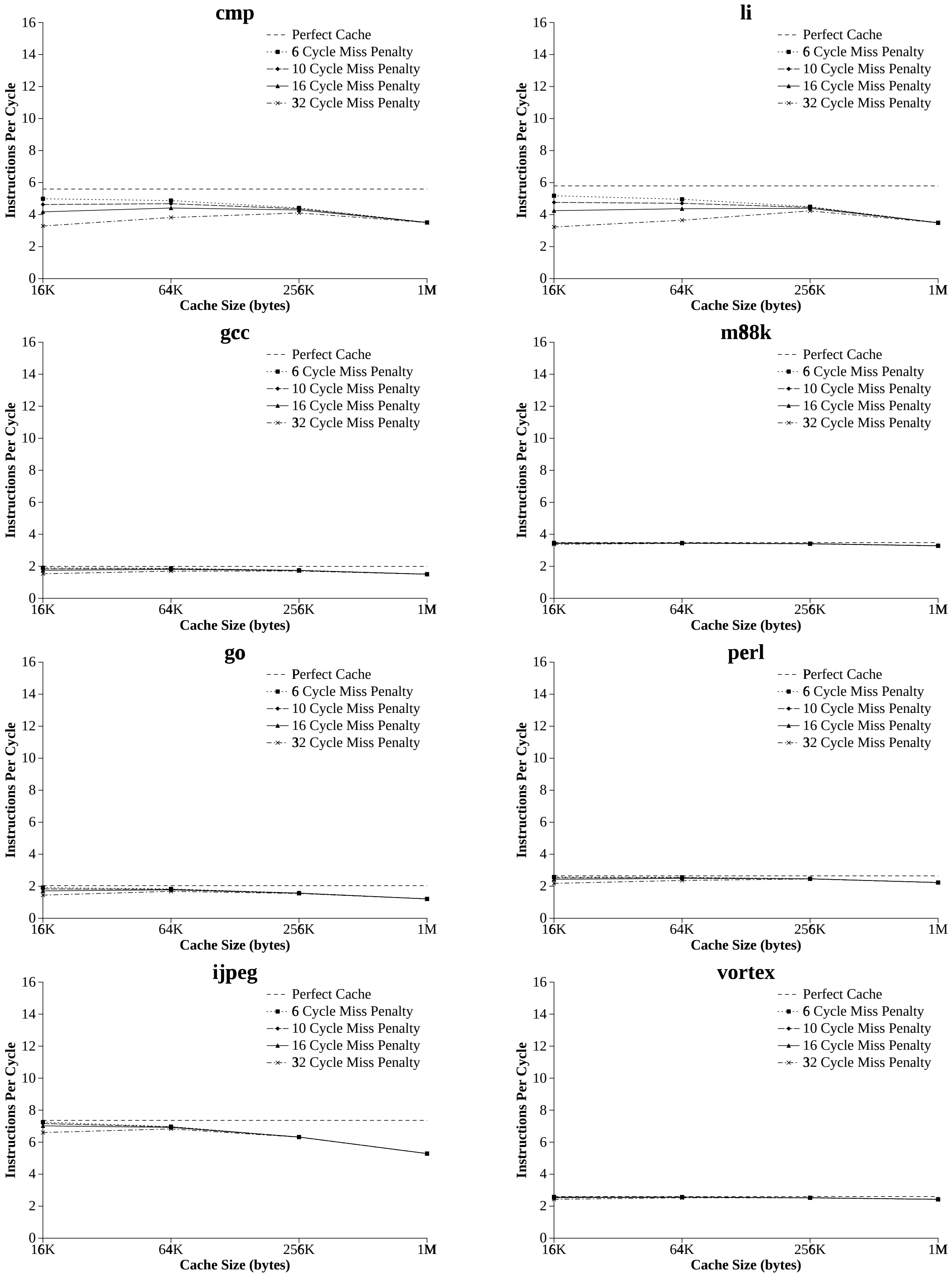


Figure A.35: Real Machine with Varied Data Cache Size (Scaled Load Latency)—SPEC Benchmarks

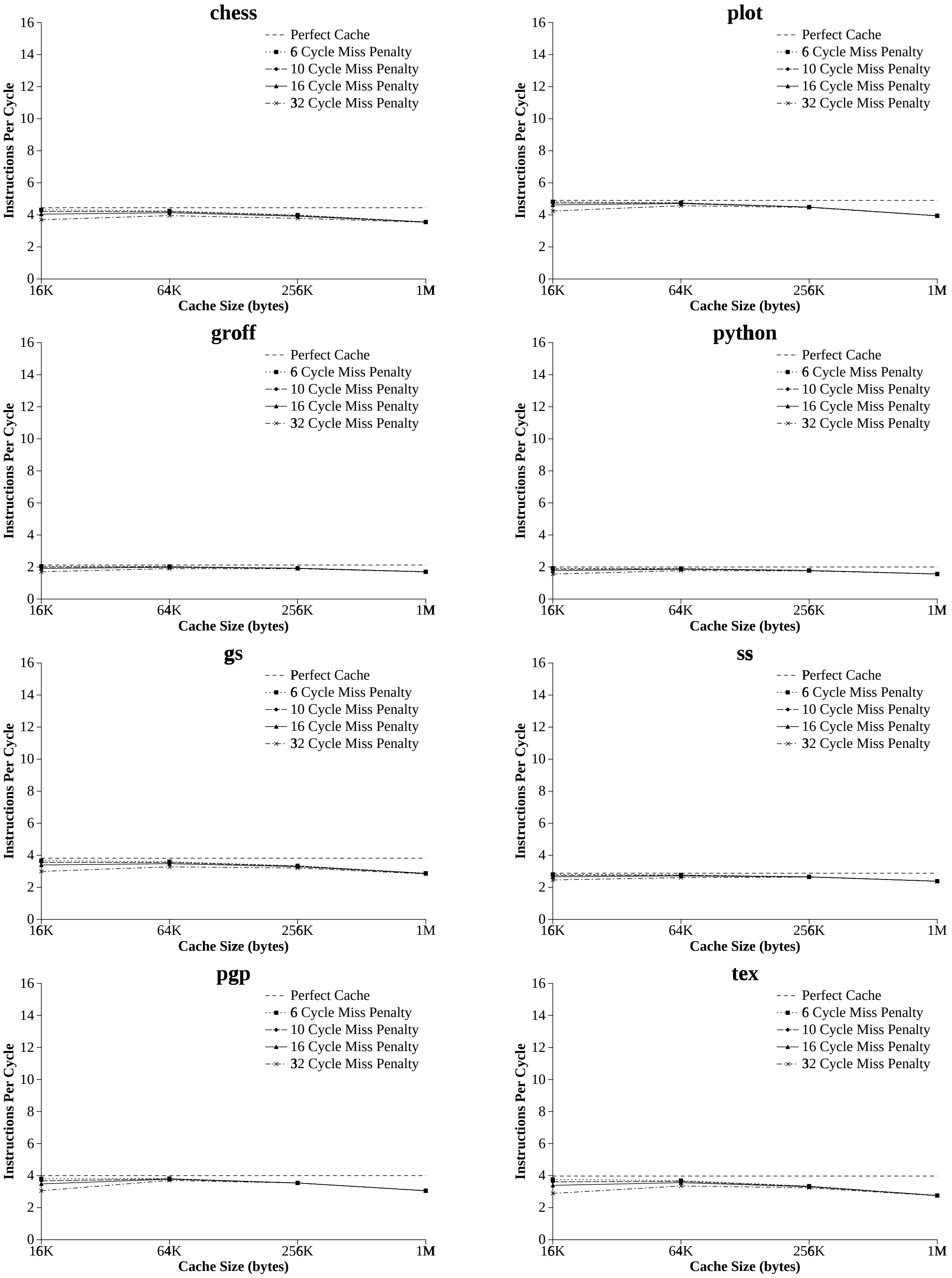


Figure A.36: Real Machine with Varied Data Cache Size (Scaled Load Latency)—Non-SPEC Benchmarks

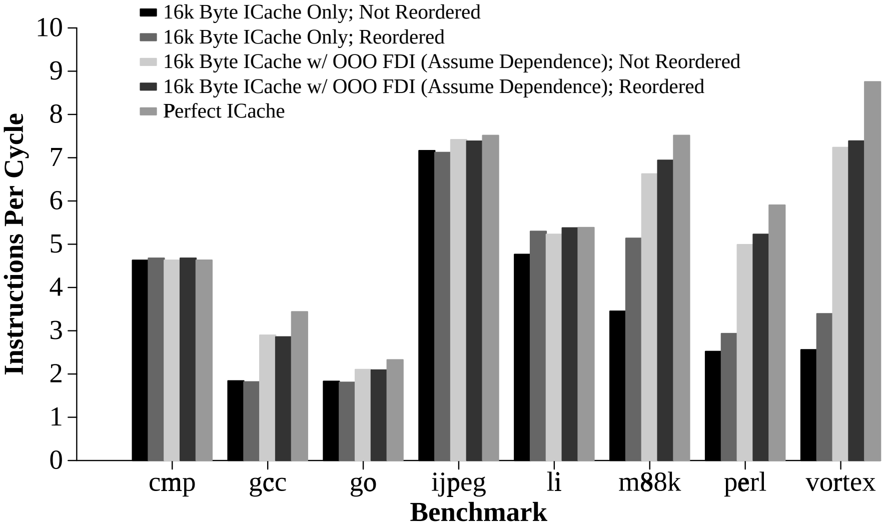


Figure A.37: Impact of Out-of-Order Fetch/Decode/Issue and Procedure Reordering—SPEC Benchmarks

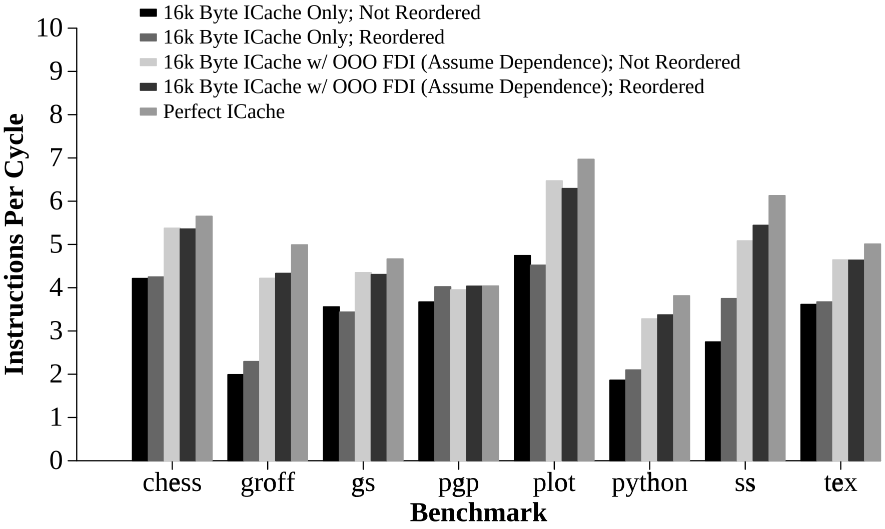


Figure A.38: Impact of Out-of-Order Fetch/Decode/Issue and Procedure Reordering—Non-SPEC Benchmarks

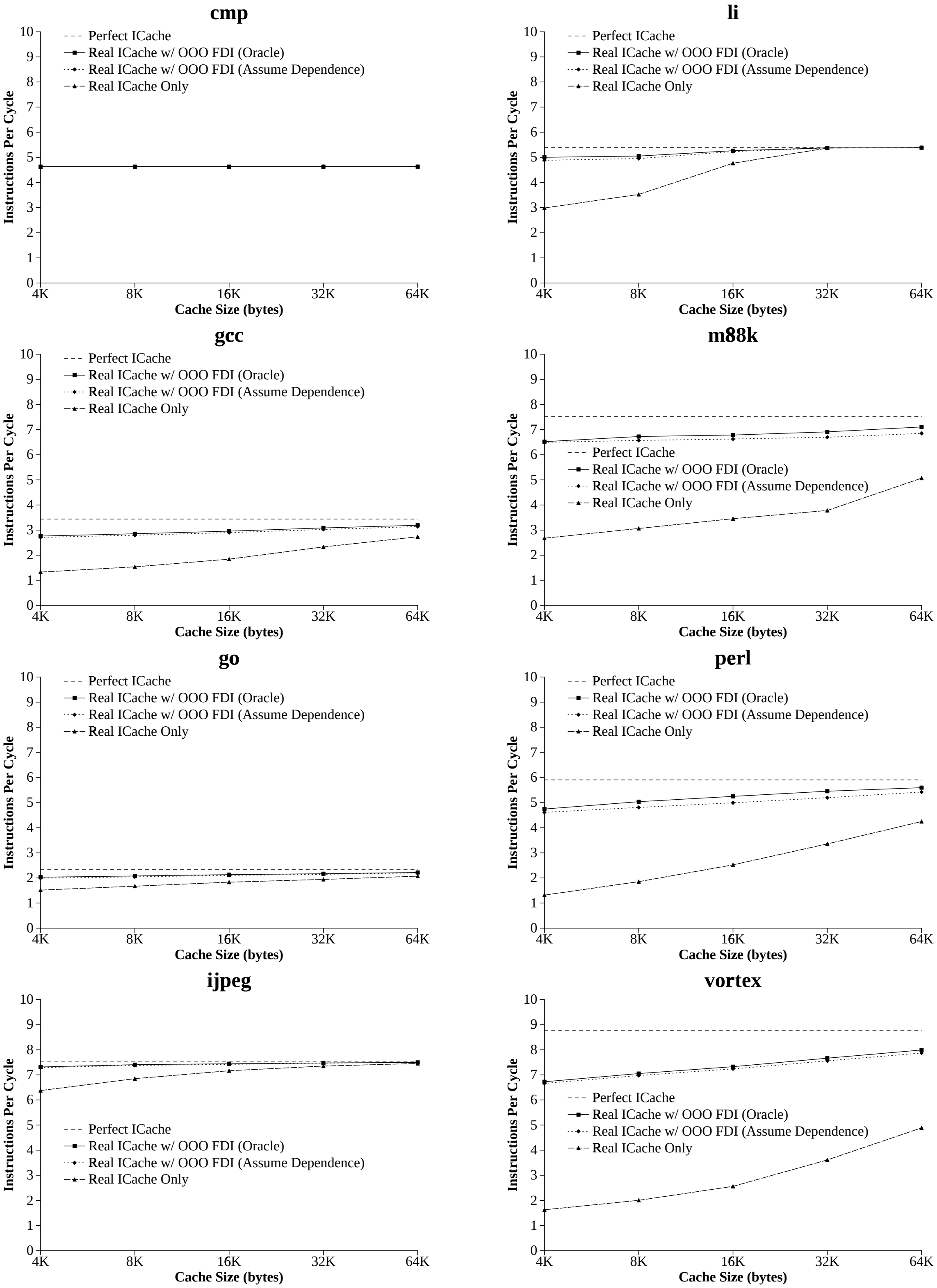


Figure A.39: Varied Instruction Cache Size (Constant Mispredict Penalty)—SPEC Benchmarks

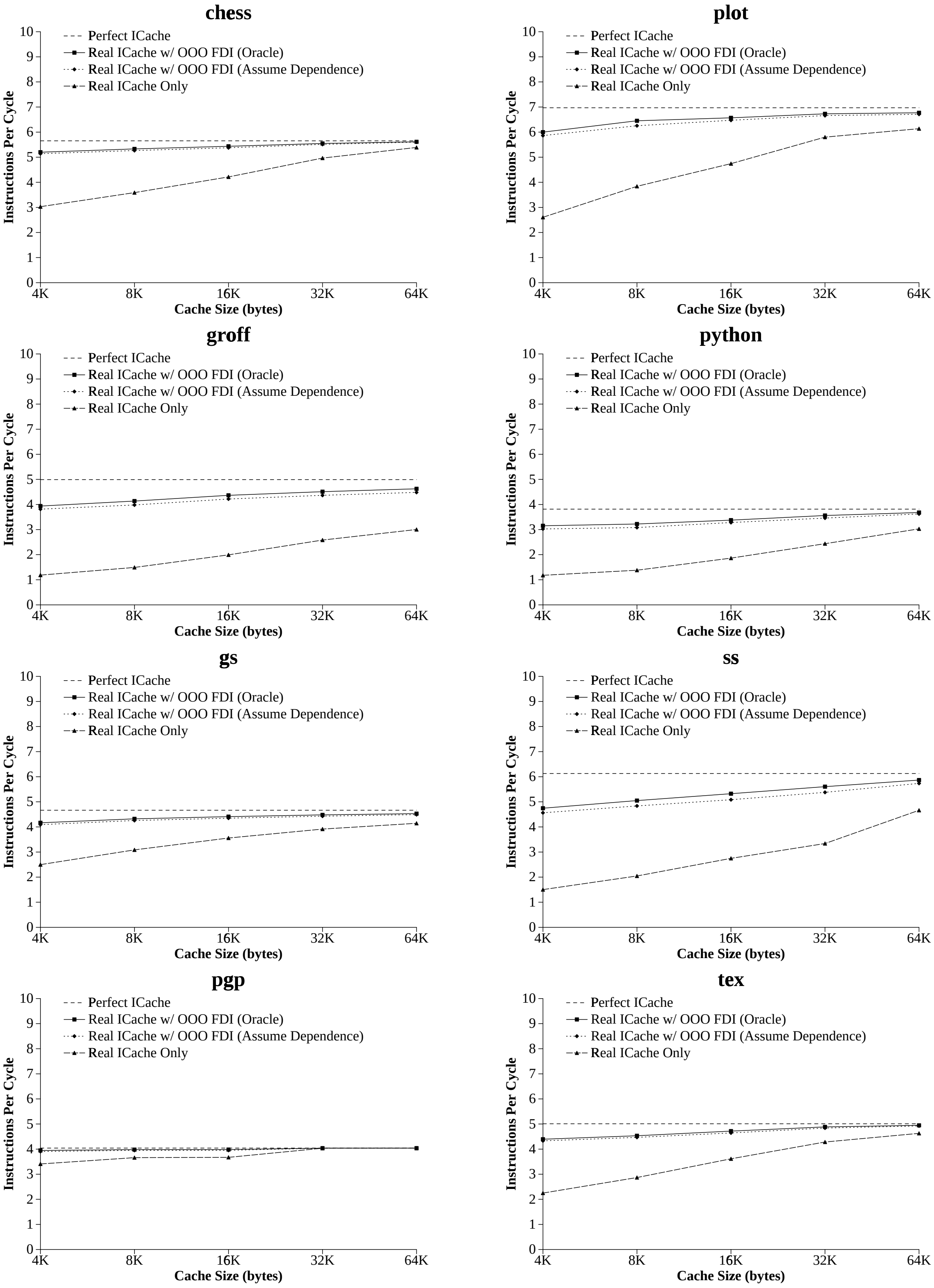


Figure A.40: Varied Instruction Cache Size (Constant Mispredict Penalty)—Non-SPEC Benchmarks

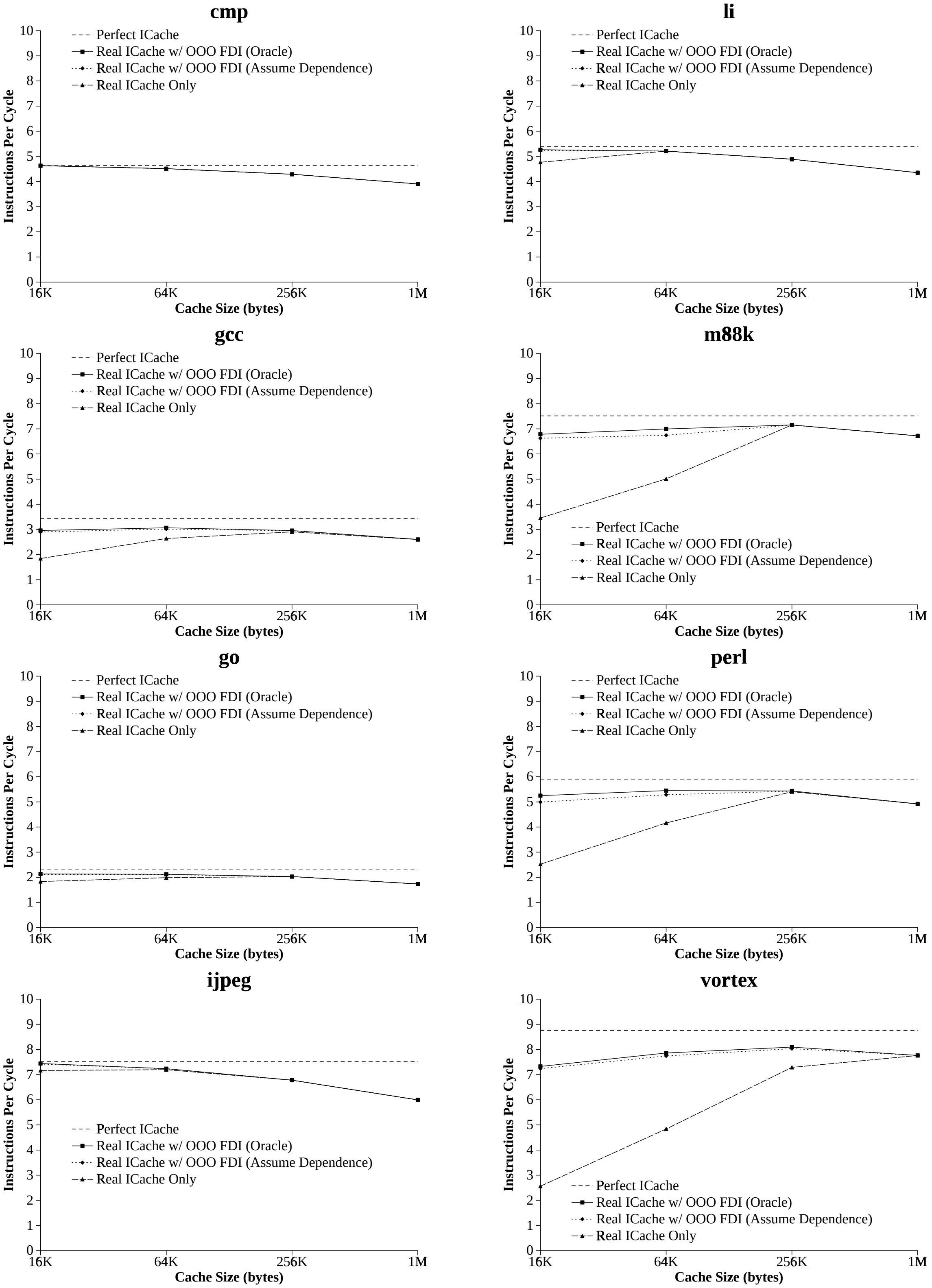


Figure A.41: Varied Instruction Cache Size
(Scaled Mispredict Penalty)—SPEC Benchmarks

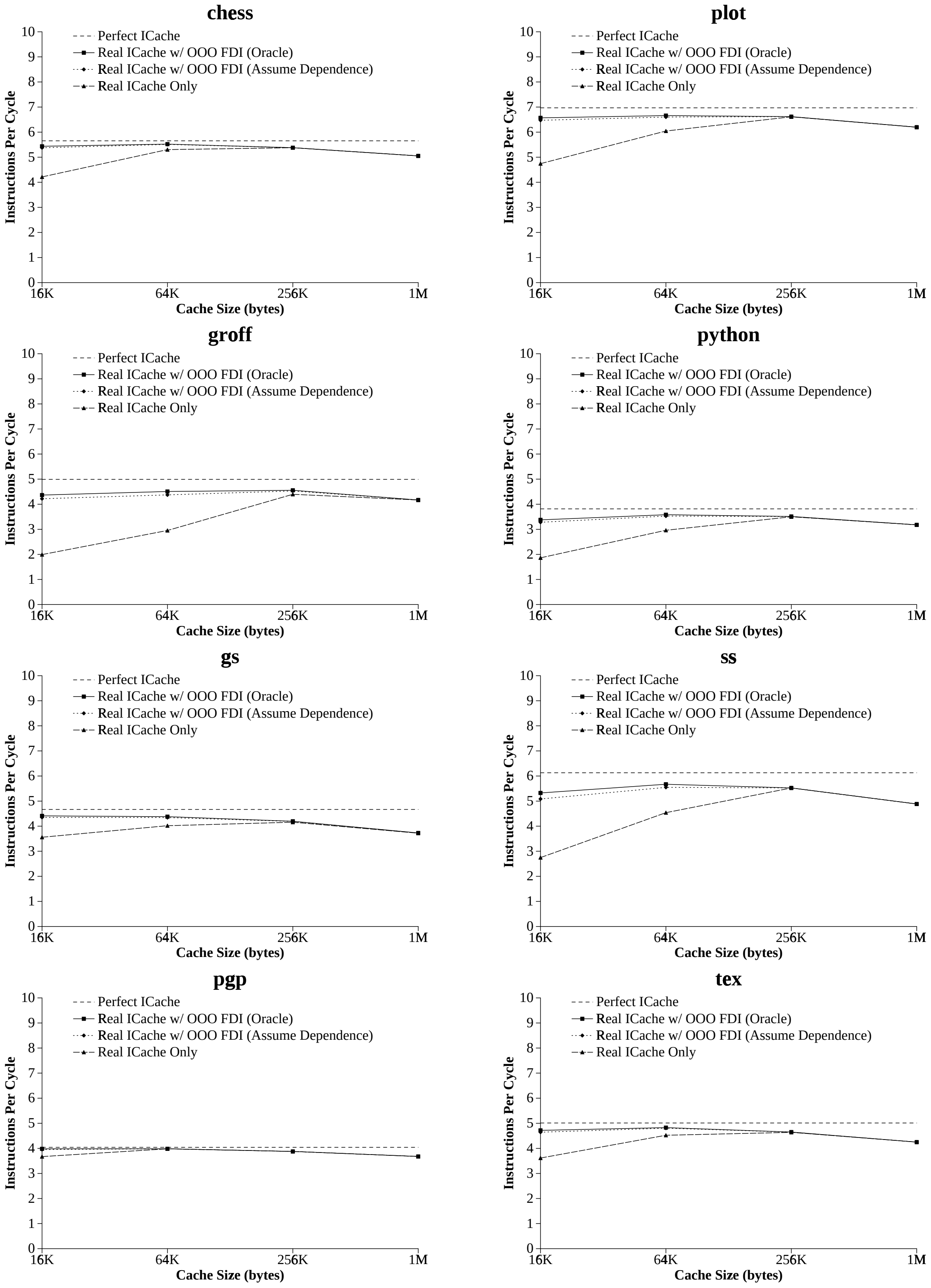


Figure A.42: Varied Instruction Cache Size (Scaled Mispredict Penalty)—Non-SPEC Benchmarks

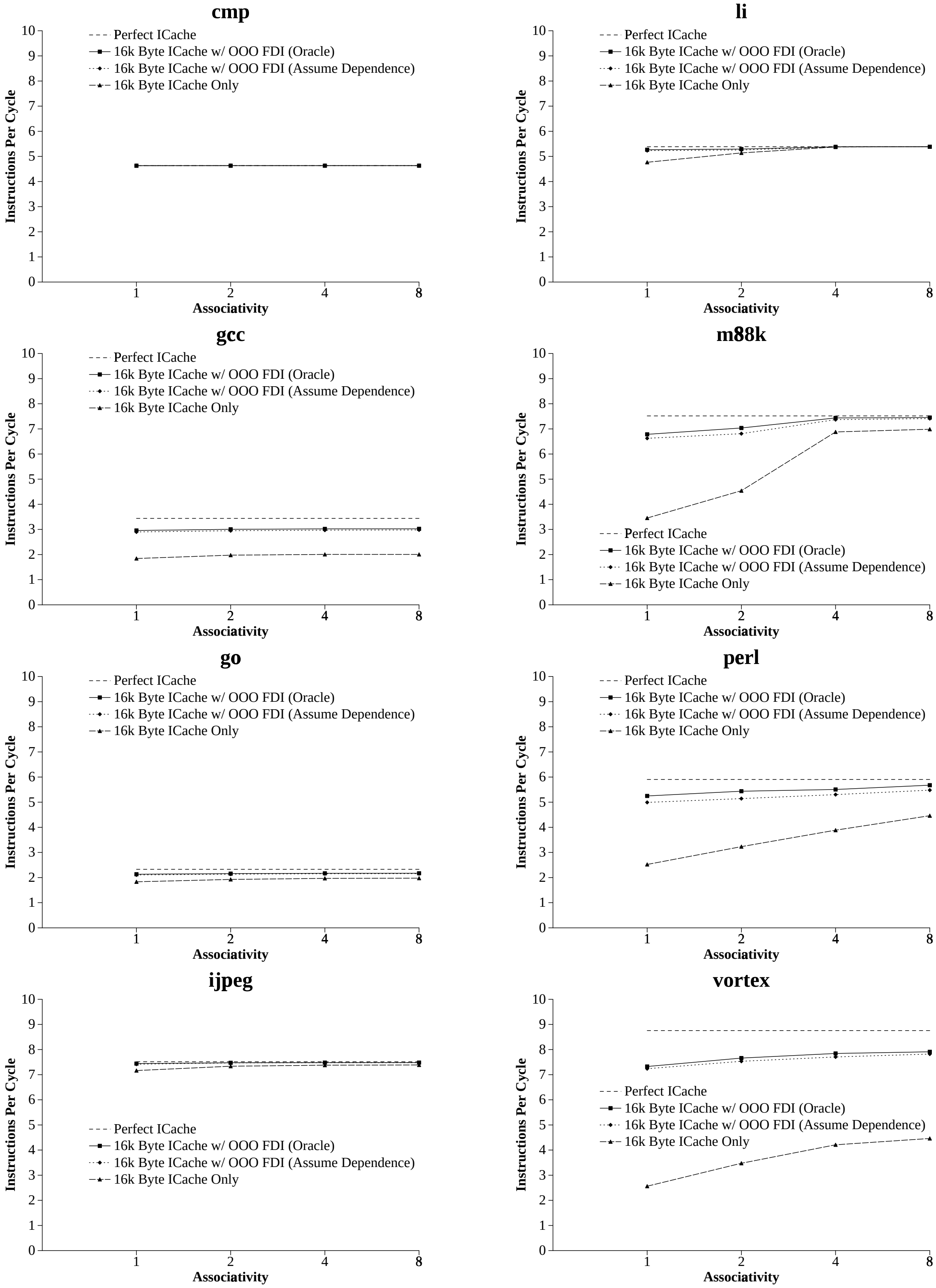


Figure A.43: Varied Instruction Cache Associativity—SPEC Benchmarks

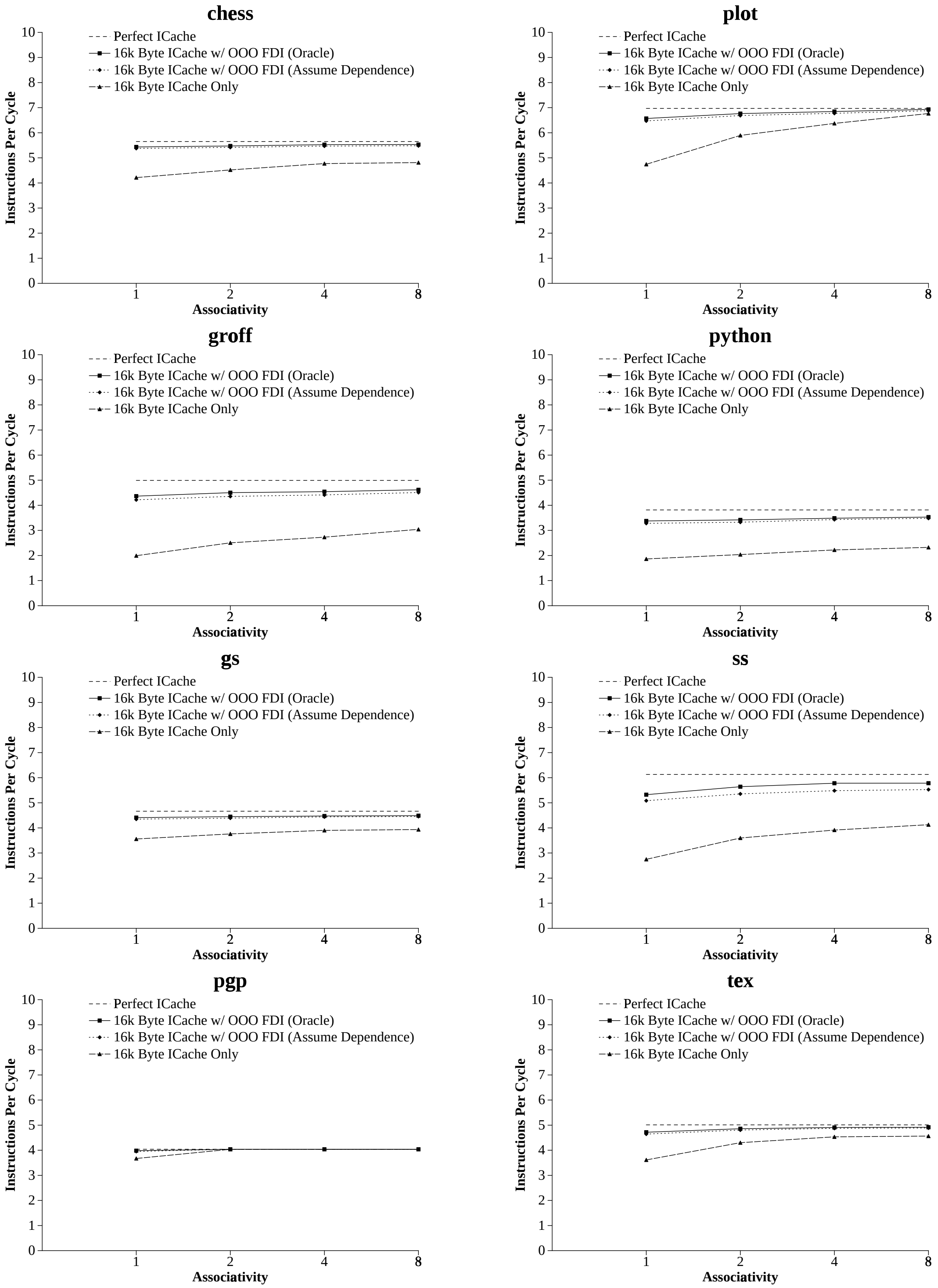


Figure A.44: Varied Instruction Cache Associativity—Non-SPEC Benchmarks

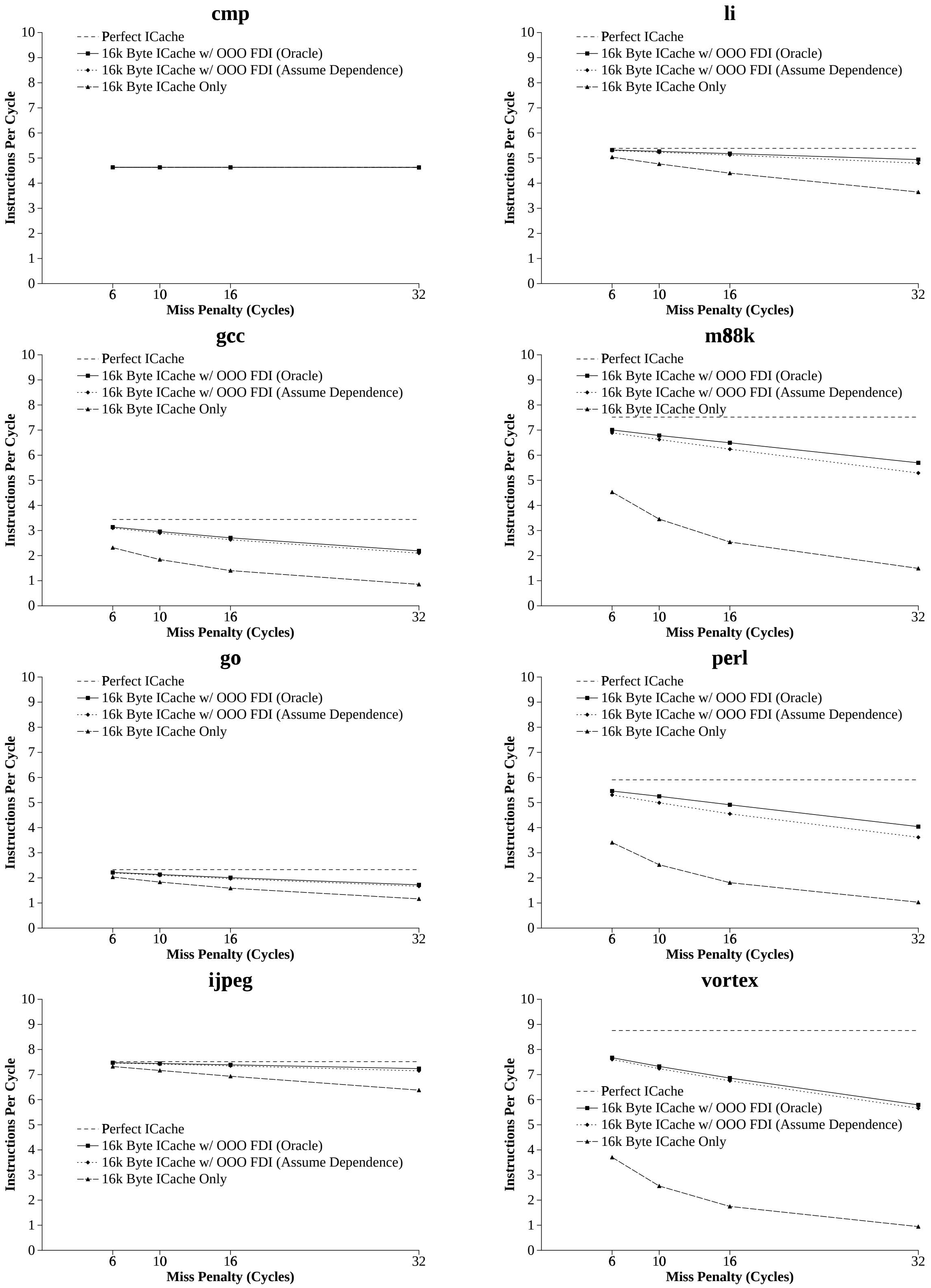


Figure A.45: Varied Instruction Cache Miss Penalty—SPEC Benchmarks

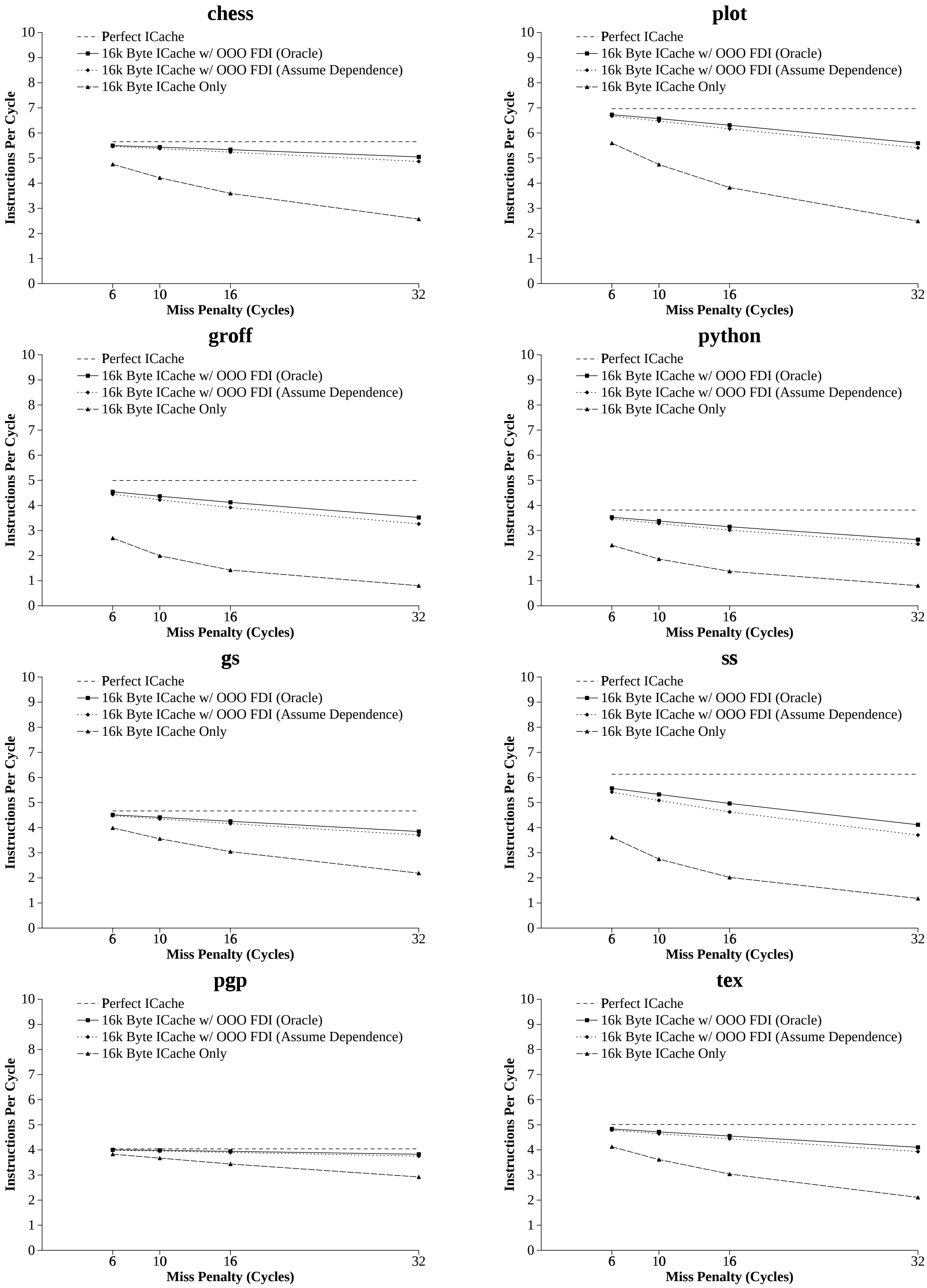


Figure A.46: Varied Instruction Cache Miss Penalty—Non-SPEC Benchmarks

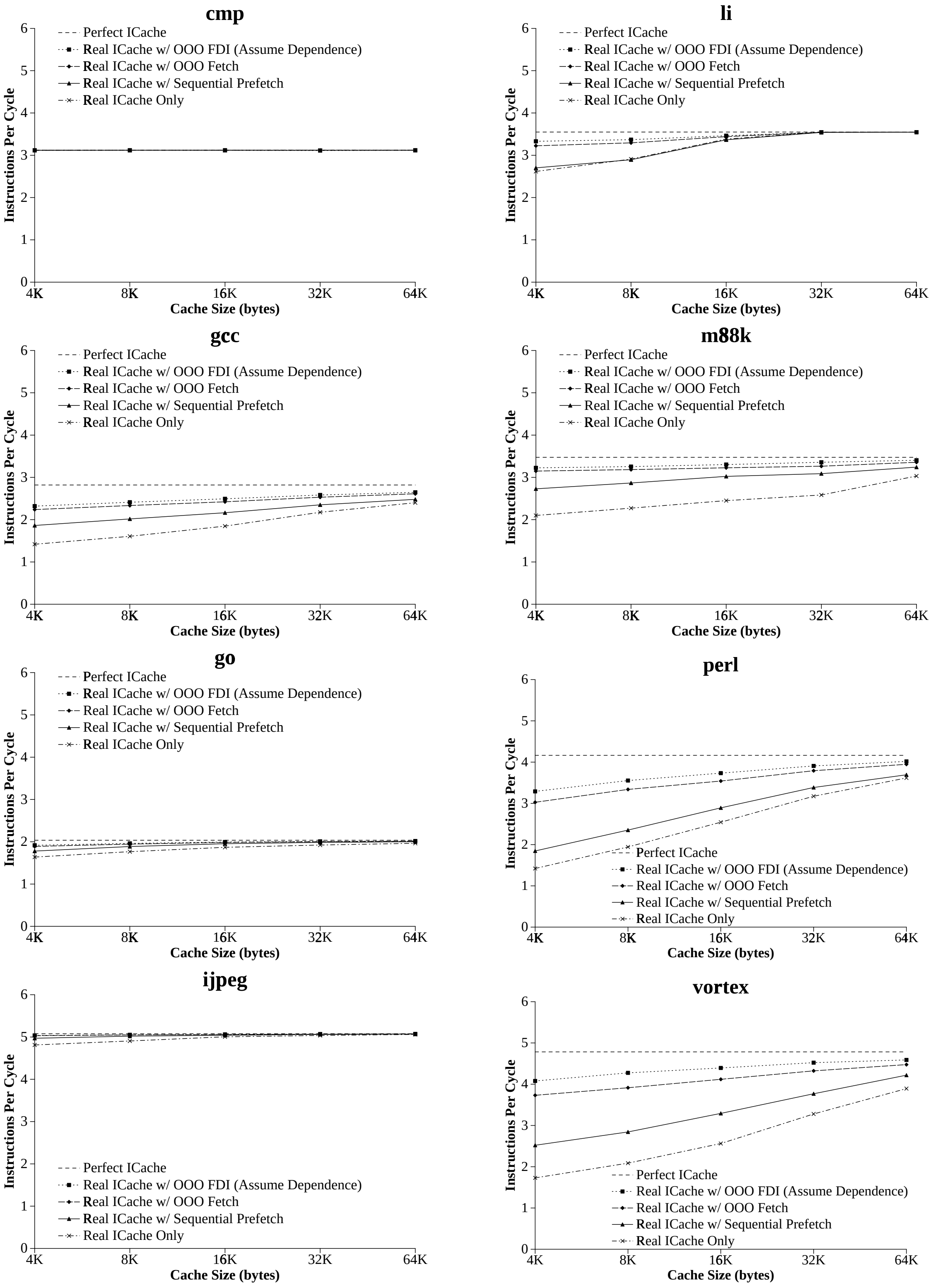


Figure A.47: Varied Instruction Cache Size—SPEC Benchmarks

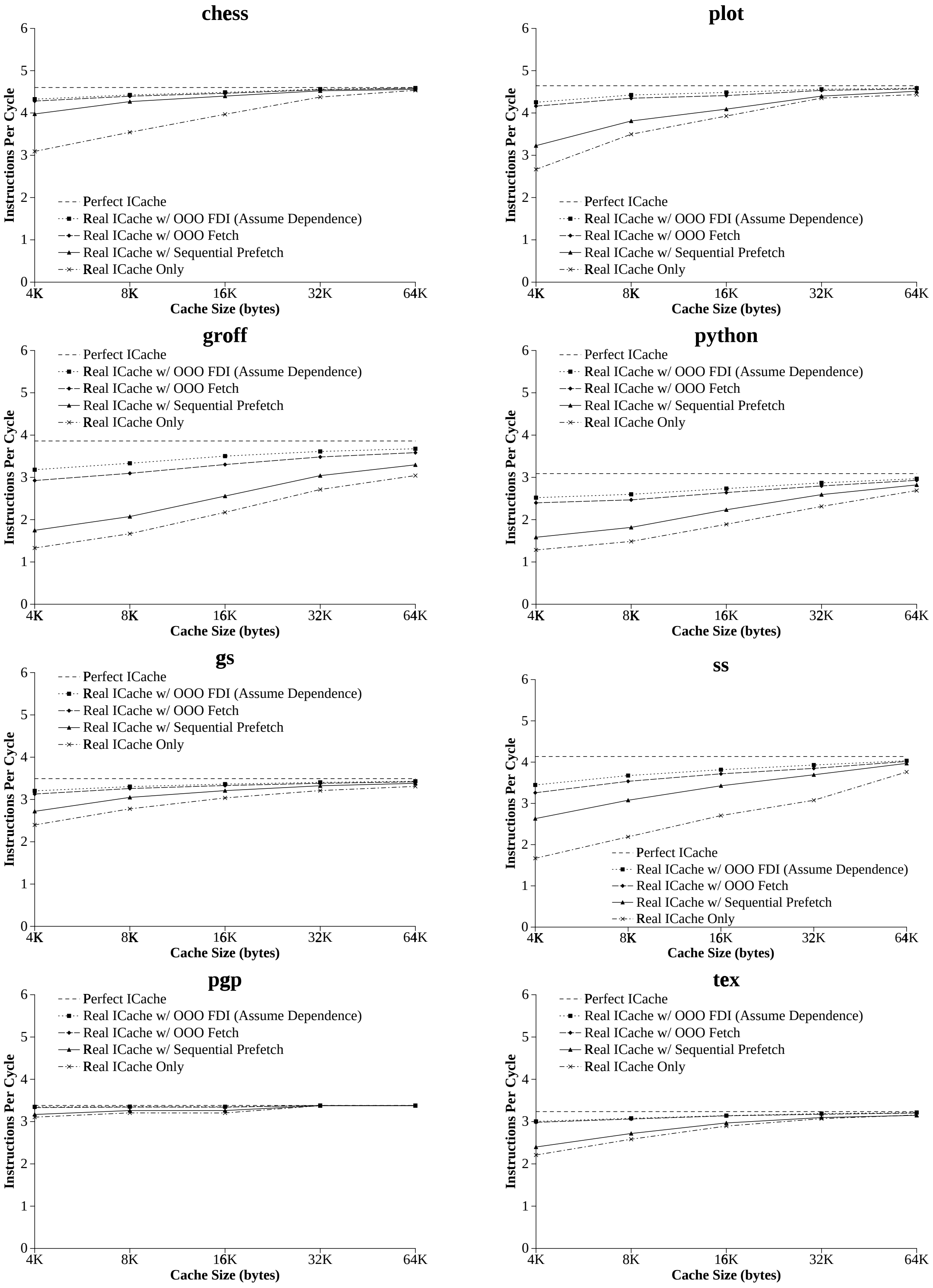


Figure A.48: Varied Instruction Cache Size—Non-SPEC Benchmarks

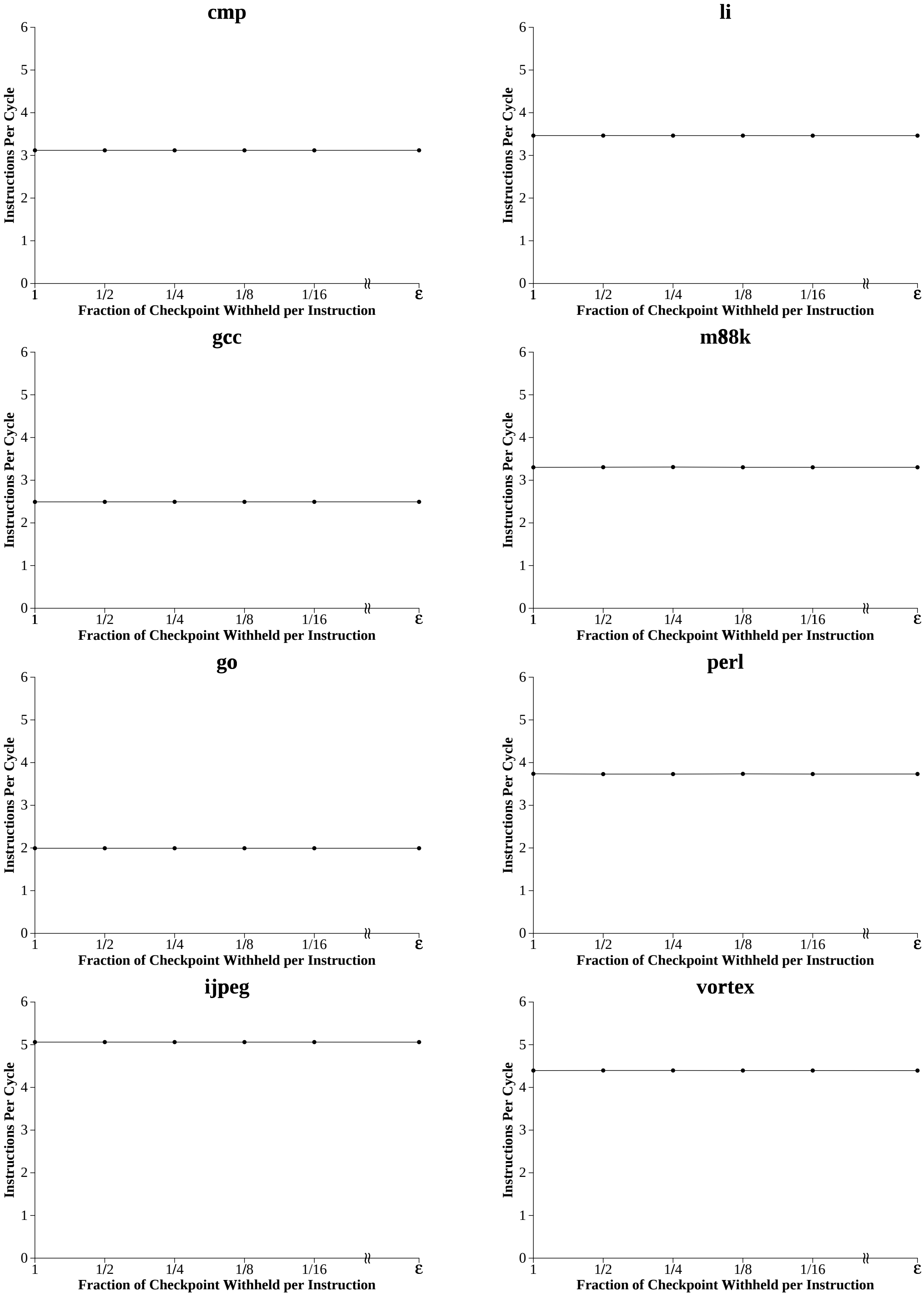


Figure A.49: Varied Checkpoint Withholding—SPEC Benchmarks

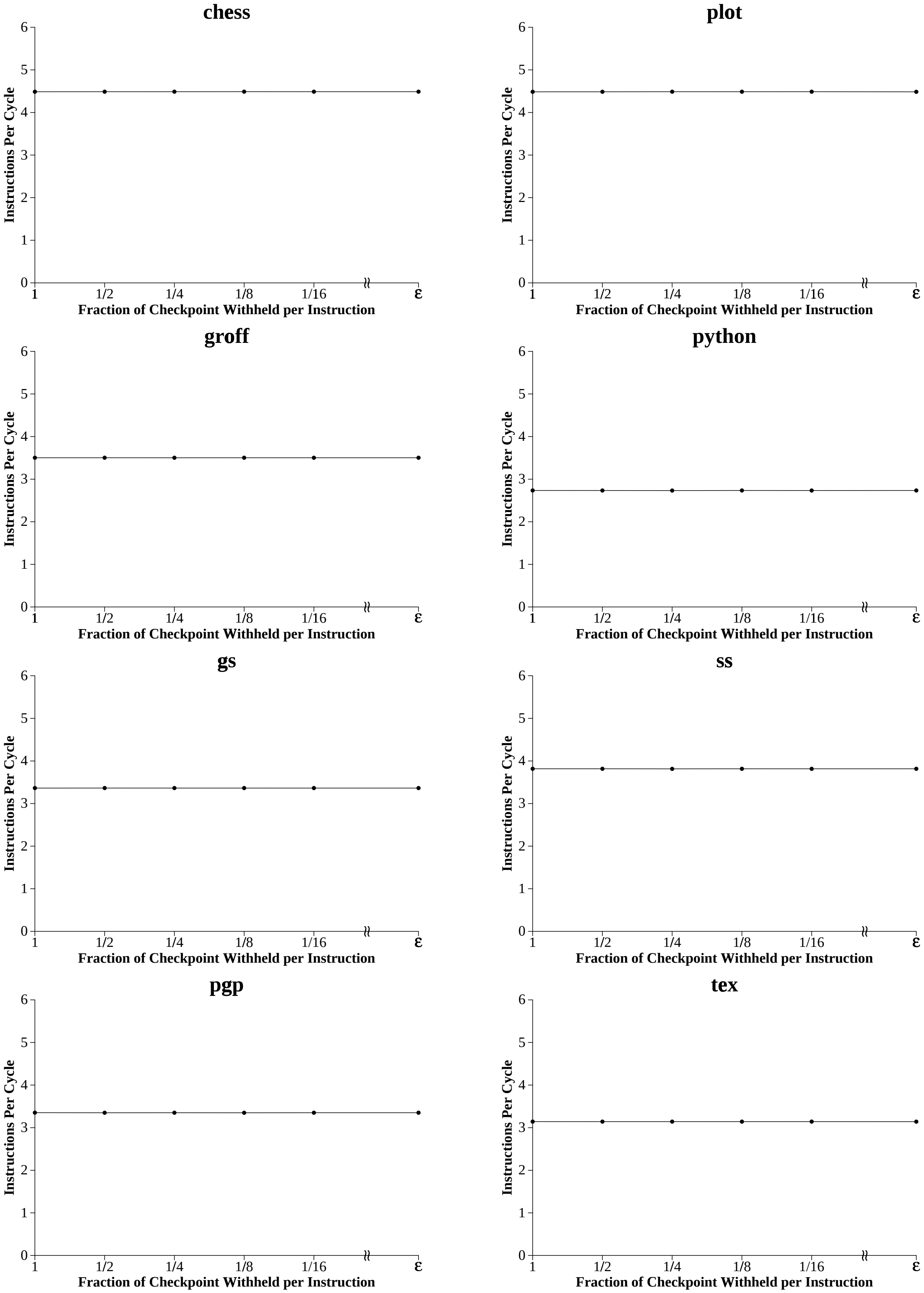


Figure A.50: Varied Checkpoint Withholding—Non-SPEC Benchmarks

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] A. Agarwal and S. Pudar, "Column-associative caches: A technique for reducing the miss rate of direct-mapped caches," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pp. 179–190, 1993.

[2] A. Agarwal, *Analysis of Cache Performance for Operating Systems and Multiprogramming*, Kluwer Academic Publishers, 1989.

[3] A. Agarwal, J. Hennessy, and M. Horowitz, "Cache performance of operating systems and multiprogramming," *ACM Transactions on Computer Systems*, vol. 6, no. 4, pp. 393–431, November 1988.

[4] A. V. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison Wesley, 1987.

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

[6] Arvind and R. S. Nikhil, "Executing a program on the MIT tagged-token dataflow architecture," *IEEE Transactions on Computers*, vol. 39, no. 3, pp. 300–318, March 1990.

[7] T. M. Austin and G. S. Sohi, "Dynamic dependency analysis of ordinary programs," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 342–351, 1992.

[8] P. Barnes, "A 500MHz 64b RISC CPU with 1.5MB on-chip cache," in *1999 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, February 1999.

[9] B. N. Bershad, D. Lee, T. H. Romer, and J. B. Chen, "Avoiding conflict misses dynamically in large direct-mapped caches," in *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 158–170, 1994.

[10] R. S. Bird, "Tabulation techniques for recursive programs," *Computing Surveys*, vol. 12, no. 4, pp. 403–417, December 1980.

[11] M. T. Bohr, "Interconnect scaling—the real limiter to high performance ULSI," in *Technical Digest of the International Electron Devices Meeting*, pp. 241–244, December 1995.

[12] M. Butler and Y. Patt, "The effect of real data cache behavior on the performance of a microarchitecture that supports dynamic scheduling," in *Proceedings of the 24th Annual ACM/IEEE International Symposium on Microarchitecture*, 1991.

[13] M. Butler and Y. Patt, "An investigation of the performance of various dynamic scheduling techniques," in *Proceedings of the 25th Annual ACM/IEEE International Symposium on Microarchitecture*, 1992.

[14] M. Butler, T.-Y. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow, "Single instruction stream parallelism is greater than two," in *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pp. 276–286, 1991.

[15] M. G. Butler, *Aggressive Execution Engines for Surpassing Single Basic Block Execution*, PhD thesis, University of Michigan, 1993.

[16] B. Calder and D. Grunwald, "Reducing branch costs via branch alignment," in *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 242–251, 1994.

[17] B. Calder and D. Grunwald, "Next cache line and set prediction," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 287–296, 1995.

[18] J. Chang, H. Chao, and K. So, "Cache design of a sub-micron CMOS system/370," in *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pp. 208–213, 1987.

[19] P.-Y. Chang, E. Hao, and Y. N. Patt, "Predicting indirect jumps using a target cache," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 274–283, 1997.

[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 Annual International Symposium on Computer Architecture*, 1999.

[21] I.-C. K. Chen, C.-C. Lee, and T. N. Mudge, "Instruction prefetching using branch prediction information," in *1997 IEEE International Conference on Computer Design*, pp. 593–601, 1997.

[22] G. Z. Chrysos and J. S. Emer, "Memory dependence prediction using store sets," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pp. 142–153, 1998.

[23] T. M. Conte, K. N. Menezes, P. M. Mills, and B. A. Patel, "Optimization of instruction fetch mechanisms for high issue rates," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995.

[24] V. Cuppu, B. Jacob, B. Davis, and T. Mudge, "A performance comparison of contemporary DRAM architectures," in *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pp. 222–233, 1999.

[25] J. B. Dennis, "Data flow supercomputers," *IEEE Computer*, vol. 13, no. 11, pp. 48–56, November 1980.

[26] K. Diefendorff, "Jalapeno powers cyrix's M3," *Microprocessor Report*, November 1998.

[27] K. Diefendorff, "Xeon replaces Pentium Pro," *Microprocessor Report*, July 1998.

[28] *DIGITAL UNIX Version 4.0D Reference Pages Documentation Kit*, Digital Equipment Corporation, 1997.

[29] *Digital Semiconductor 21164 Alpha Microprocessor Product Brief*, Digital Equipment Corporation, Hudson, MA, March 1997. Technical Document EC-QP97D-TE.

[30] N. Drach and A. Seznec, "MIDEE: Smoothing branch and instruction cache miss penalties on deep pipelines," in *Proceedings of the 26th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 193–201, 1993.

[31] S. Dutta and M. Franklin, "Control flow prediction with Tree-Like subgraphs for superscalar processors," in *Proceedings of the 28th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 258–263, 1995.

[32] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic, "Memory-System design considerations for Dynamically-Scheduled processors," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 133–143, 1997.

[33] K. I. Farkas, N. P. Jouppi, and P. Chow, "How useful are Non-Blocking loads, stream buffers and speculative execution in multiple issue processors?," in *Proceedings of the First IEEE International Symposium on High Performance Computer Architecture*, pp. 78–89, 1995.

[34] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Transactions on Computers*, vol. C-30, no. 7, pp. 478–490, July 1981.

[35] M. Franklin and M. Smotherman, "A Fill-Unit approach to multiple instruction issue," in *Proceedings of the 27th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 162–171, 1994.

[36] M. Franklin and G. S. Sohi, "The expandable split window paradigm for exploiting fine-grain parallelism," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 58–67, 1992.

[37] M. Franklin and G. S. Sohi, "ARB: A hardware mechanism for dynamic reordering of memory references," *IEEE Transactions on Computers*, vol. 45, no. 5, pp. 552–571, May 1996.

[38] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, 1979.

[39] B. A. Gieseke, R. L. Allmon, D. W. Bailey, B. J. Benschneider, S. M. Britton, J. D. Clouser, H. R. F. III, J. A. Farrell, M. K. Gowan, C. L. Houghton, J. B. Keller, T. H. Lee, D. L. Leibholz, S. C. Lowell, M. D. Matson, R. J. Matthew, V. Peng, M. D. Quinn, D. A. Priore, M. J. Smith, and K. E. Wilcox, "A 600MHz superscalar RISC microprocessor with out-of-order execution," in *1997 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, pp. 176–178, February 1997.

[40] G. F. Grohoski, "Machine organization of the IBM RISC system/6000 processor," *IBM Journal of Research and Development*, vol. 34, no. 1, pp. 37–58, January 1990.

[41] R. Gupta and C.-H. Chi, "Improving instruction cache behavior by reducing cache pollution," in *Proceedings of Supercomputing '90*, pp. 82–91, 1990.

[42] J. R. Gurd, C. C. Kirkham, and I. Watson, "The manchester prototype dataflow computer," *Communications of the ACM*, vol. 28, no. 1, pp. 34–52, January 1985.

[43] L. Gwennap, "Digital 21264 sets new standard," *Microprocessor Report*, pp. 11–16, October 1996.

[44] L. Gwennap, "Klamath extends P6 family," *Microprocessor Report*, February 1997.

[45] L. Gwennap, "Alpha 21364 to ease memory bottleneck," *Microprocessor Report*, October 1998.

[46] E. Hao, P.-Y. Chang, M. Evers, and Y. N. Patt, "Increasing the instruction fetch rate via block-structured instruction set architectures," in *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, 1996.

[47] E. Hao, P.-Y. Chang, M. Evers, and Y. N. Patt, "Increasing the instruction fetch rate via block-structured instruction set architectures," *International Journal of Parallel Programming*, vol. 26, no. 4, pp. 449–478, August 1998.

[48] A. H. Hashemi, D. R. Kaeli, and B. Calder, "Efficient procedure mapping using cache line coloring," in *Proceedings of the ACM SIGPLAN'97 Conference on Programming Language Design and Implementation*, pp. 171–182, 1997.

[49] A. H. Hashemi, D. R. Kaeli, and B. Calder, "Procedure mapping using static call graph estimation," in *Proceedings of the Workshop on the Interaction between Compilers and Computer Architectures*, 1997.

[50] W. W. Hwu and P. P. Chang, "Achieving high instruction cache performance with an optimizing compiler," in *Proceedings of the 16th Annual International Symposium on Computer Architecture*, 1989.

[51] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The superblock: An effective technique for VLIW and superscalar compilation," *Journal of Supercomputing*, vol. 7, no. 9–50, , 1993.

[52] W. W. Hwu and Y. N. Patt, "Checkpoint repair for high-performance out-of-order execution machines," *IEEE Transactions on Computers*, vol. C-36, no. 12, , December 1987.

[53] W. W. Hwu and Y. N. Patt, "Checkpoint repair for out-of-order execution machines," in *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pp. 18–26, 1987.

[54] Q. Jacobson, E. Rotenberg, and J. E. Smith, "Path-based next trace prediction," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, 1997.

[55] M. Johnson, *Superscalar Microprocessor Design*, Prentice-Hall, Inc., 1991.

[56] T. L. Johnson, M. C. Merten, and W. mei W. Hwu, "Run-time spatial locality detection and optimization," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 57–64, 1997.

[57] D. Joseph and D. Grunwald, "Prefetching using markov predictors," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 252–263, 1997.

[58] 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 Annual International Symposium on Computer Architecture*, pp. 364–373, 1990.

[59] N. P. Jouppi and D. W. Wall, "Available instruction-level parallelism for superscalar and superpipelined machines," in *Proceedings of the 2th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 272–282, 1989.

[60] 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 ACM/IEEE Conference on Parallel Architectures and Compilation Techniques*, pp. 58–67, 1996.

[61] S. Jourdan, P. Sainrat, and D. Litaize, "Exploring configurations of functional units in an out-of-order superscalar processor," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 117–125, 1995.

[62] S. Jourdan, J. Stark, T.-H. Hsing, and Y. N. Patt, "Recovery requirements of branch prediction storage structures in the presence of mispredicted-path execution.," *International Journal of Parallel Programming*, vol. 25, no. 05, pp. 363–384, 1997.

[63] T. Juan, T. Lang, and J. J. Navarro, "The difference-bit cache," in *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pp. 114–120, 1996.

[64] J. Kalamatianos and D. R. Kaeli, "Temporal-based procedure reordering for improved instruction cache performance," in *Proceedings of the Fourth IEEE International Symposium on High Performance Computer Architecture*, pp. 224–253, 1998.

[65] J. Keller, *The 21264: A Superscalar Alpha Processor with Out-of-Order Execution*, Digital Equipment Corporation, Hudson, MA, October 1996. Microprocessor Forum presentation.

[66] R. E. Kessler and M. D. Hill, "Page placement algorithms for large real-indexed caches," *ACM Transactions on Computer Systems*, vol. 10, no. 4, pp. 338–360, November 1992.

[67] R. Kessler, R. Joss, A. Lebeck, and M. Hill, "Inexpensive implementations of set-associativity," in *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pp. 131–139, 1989.

[68] T. Kimura, K. Takeda, Y. Aimoto, N. Nakamura, T. Iwasaki, Y. Nakazawa, H. Toyoshima, M. Hamada, M. Togo, H. Nobusawa, and T. Tanigawa, "64Mb 6.8ns

random ROW access DRAM macro for ASICs," in *1999 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, February 1999.

[69] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," in *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pp. 81–87, 1981.

[70] G. Kurpanek, K. Chan, J. Zheng, E. DeLano, and W. Bryg, "PA7200: A PA-RISC processor with integrated high performance MP bus interface," in *COMPCON Digest of Papers*, February 1994.

[71] M. H. Lipasti and J. P. Shen, "Exceeding the dataflow limit via value prediction," in *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 226–237, 1996.

[72] C.-K. Luk and T. C. Mowry, "Cooperative prefetching: Compiler and hardware support for effective instruction prefetching in modern processors," in *Proceedings of the 31th Annual ACM/IEEE International Symposium on Microarchitecture*, 1998.

[73] W. L. Lynch, G. Lauterbach, and J. I. Chamdani, "Low load latency through sum-addressed memory (SAM)," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pp. 369–379, 1998.

[74] S. A. Mahlke, R. E. Hank, R. A. Bringmann, J. C. Gyllenhaal, D. M. Gallagher, and W. W. Hwu, "Characterizing the impact of predicated execution on branch prediction," in *Proceedings of the 27th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 217–227, 1994.

[75] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock," in *Proceedings of the 25th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 45–54, 1992.

[76] D. Matzke, "Will physical scalability sabotage performance gains?," *IEEE Computer*, vol. 30, pp. 37–39, September 1997.

[77] G. McFarland, *CMOS Technology Scaling and Its Impact on Cache Delay*, PhD thesis, Stanford University, 1995.

[78] S. McFarling, "Program optimization for instruction caches," in *Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 183–191, 1989.

[79] S. McFarling, "Cache replacement with dynamic exclusion," Technical Report TN-22, Digital Western Research Laboratory, November 1991.

[80] S. McFarling, "Procedure merging with instruction caches," in *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pp. 71–79, 1991.

[81] S. McFarling, "Combining branch predictors," Technical Report TN-36, Digital Western Research Laboratory, June 1993.

[82] S. Melvin and Y. Patt, "Enhancing instruction scheduling with a block-structured ISA," *International Journal of Parallel Programming*, vol. 23, no. 3, pp. 221–243, June 1995.

[83] S. Melvin and Y. N. Patt, "Exploiting fine-grained parallelism through a combination of hardware and software techniques," in *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pp. 287–297, 1991.

[84] S. W. Melvin and Y. N. Patt, "Performance benefits of large execution atomic units in dynamically scheduled machines," in *Proceedings of Supercomputing '89*, pp. 427–432, 1989.

[85] S. W. Melvin, M. C. Shebanow, and Y. N. Patt, "Hardware support for large atomic units in dynamically scheduled machines," in *Proceedings of the 21st Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 60–63, 1988.

[86] A. Mendlson, S. S. Pinter, and R. Shtokhamer, "Compile time instruction cache optimizations," in *Proceedings of the Fifth International Conference on Compiler Construction*, pp. 404–418, 1994.

[87] K. N. Menezes, S. W. Sathaye, and T. M. Conte, "Path prediction for high issue-rate processors," in *Proceedings of the 1997 ACM/IEEE Conference on Parallel Architectures and Compilation Techniques*, 1997.

[88] D. Meyer, *AMD-K7$^{(TM)}$ Technology Presentation*, Advanced Micro Devices, Inc., Sunnyvale, CA, October 1998. Microprocessor Forum presentation.

[89] I. MIPS Technologies, "R10000 microprocessor product overview," in *MIPS Open RISC Technology, MIPS Technologies*, 1994.

[90] S.-M. Moon and K. Ebcioğlu, "An efficient resource-constrained global scheduling technique for superscalar and VLIW processors," in *Proceedings of the 25th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 55–71, 1992.

[91] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi, "Dynamic speculation and synchronization of data dependences," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

[92] A. Moshovos and G. S. Sohi, "Streamlining inter-operation memory communication via data dependence prediction," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 235–245, 1997.

[93] I. Naritake, T. Sugibayashi, Y. Nakajima, S. Utsugi, M. Hamada, M. Togo, R. Kubota, T. Fujii, N. Yoshimatsu, H. Hatayama, T. Murotani, and T. Okuda, "A 12ns 8MB DRAM secondary cache for a 64b microprocessor," in *1999 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, February 1999.

[94] A. Nicolau, "Run-time disambiguation: Coping with statically unpredictable dependencies," *IEEE Transactions on Computers*, vol. 38, no. 5, pp. 663–678, May 1989.

[95] S. Palacharla and R. E. Kessler, "Evaluating stream buffers as a secondary cache replacement," in *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pp. 24–33, 1994.

[96] S. J. Patel, M. Evers, and Y. N. Patt, "Improving trace cache effectiveness with branch promotion and trace packing," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998.

[97] S. J. Patel, *Trace Cache Design for Wide-Issue Superscalar Processors*, PhD thesis, University of Michigan, Ann Arbor, 1999.

[98] S. J. Patel, D. H. Friendly, and Y. N. Patt, "Critical issues regarding the trace cache fetch mechanism," Technical Report CSE-TR-335-97, University of Michigan Technical Report, May 1997.

[99] Y. Patt, S. Patel, M. Evers, D. Friendly, and J. Stark, "One billion transistors, one uniprocessor, one chip," *IEEE Computer*, vol. 30, pp. 51–57, September 1997.

[100] Y. Patt, W. Hwu, and M. Shebanow, "HPS, a new microarchitecture: Rationale and introduction," in *Proceedings of the 18th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 103–107, 1985.

[101] 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 Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 109–116, 1985.

[102] A. Peleg and U. Weiser. *Dynamic Flow Instruction Cache Memory Organized Around Trace Segments Independant of Virtual Address Line.* U.S. Patent Number 5,381,533, 1994.

[103] K. Pettis and R. C. Hansen, "Profile guided code positioning," in *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pp. 16–27, 1990.

[104] J. Pierce and T. Mudge, "Wrong-path instruction prefetching," in *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 165–175, 1996.

[105] G. Reinman, T. Austin, and B. Calder, "A scalable front-end architecture for fast instruction delivery," in *Proceedings of the 26th Annual International Symposium on Computer Architecture*, 1999.

[106] S. E. Richardson, "Caching function results: Faster arithmetic by avoiding unnecessary computation," Technical report, Sun Microsystems Laboratories, 1992.

[107] J. A. Rivers and E. S. Davidson, "Reducing conflicts in direct-mapped caches with a temporality-based design," in *Proceedings of the 1996 International Conference on Parallel Processing*, pp. 151–162, 1996.

[108] E. Rotenberg, S. Bennett, and J. E. Smith, "Trace cache: a low latency approach to high bandwidth instruction fetching," in *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, 1996.

[109] A. Seznec, "A case for two-way skewed-associative caches," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pp. 169–178, 1993.

[110] A. Seznec, "DASC cache," in *Proceedings of the First IEEE International Symposium on High Performance Computer Architecture*, 1995.

[111] A. Seznec, S. Jourdan, P. Sainrat, and P. Michaud, "Multiple-block ahead branch predictors," in *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[112] R. L. Sites, *Alpha Architecture Reference Manual*, Digital Press, Burlington, MA, 1992.

[113] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark, "Improving prediction for procedure returns with return-address-stack repair mechanisms," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pp. 259–271, 1998.

[114] A. J. Smith, "Sequential program prefetching in memory hierarchies," *IEEE Computer*, vol. 11, no. 12, pp. 7–21, December 1978.

[115] A. J. Smith, "Cache memories," *Computing Surveys*, vol. 14, no. 4, pp. 473–530, 1982.

[116] J. E. Smith, G. E. Dermer, B. D. Vanderwarn, S. D. Klinger, C. M. Rozewski, D. L. Fowler, K. R. Scidmore, and J. P. Laudon, "The ZS-1 central processor," in *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 199–204, 1987.

[117] J. E. Smith and W.-C. Hsu, "Prefetching in supercomputer instruction caches," in *Proceedings of Supercomputing '92*, pp. 588–597, November 1992.

[118] K. So and R. N. Rechtschaffen, "Cache operations by MRU change," *IEEE Transactions on Computers*, vol. 37, no. 6, pp. 700–709, June 1988.

[119] *Welcome to SPEC*, The Standard Performance Evaluation Corporation. http://www.specbench.org/.

[120] J. Stark, P. Racunas, and Y. N. Patt, "Reducing the performance impact of instruction cache misses by writing instructions into the reservation stations out-of-order," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 34–43, 1997.

[121] J. Stark, M. Evers, and Y. N. Patt, "Variable length path branch prediction," in *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 170–179, 1998.

[122] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of Research and Development*, vol. 11, pp. 25–33, January 1967.

[123] N. Topham and A. G. J. González, "The design and performance of a conflict-avoiding cache," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 71–80, 1997.

[124] J. Torrellas, C. Xia, and R. Daigle, "Optimizing instruction cache performance for operating system intensive workloads," in *Proceedings of the First IEEE International Symposium on High Performance Computer Architecture*, pp. 360–369, 1995.

[125] D. M. Tullsen, S. J. Eggers, J. S. Emer, and H. M. Levy, "Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor," in *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pp. 191–202, 1996.

[126] G. S. Tyson and T. M. Austin, "Improving the accuracy and performance of memory communication through renaming," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, 1997.

[127] S. Vajapeyam and T. Mitra, "Improving superscalar instruction dispatch and issue by exploiting dynamic code sequences," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 1–12, 1997.

[128] D. W. Wall, "Limits of instruction-level parallelism," in *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 176–188, 1991.

[129] W.-D. Weber and A. Gupta, "Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: Preliminary results," in *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pp. 273–280, 1989.

[130] S. Weiss and J. E. Smith, "Instruction issue logic in pipelined supercomputers," *IEEE Transactions on Computers*, vol. C-33, no. 11, pp. 1013–1022, November 1984.

[131] C. Xia and J. Torrellas, "Instruction prefetching of system codes with layout optimized for reduced cache misses," in *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, 1996.

[132] W. Yamamoto, M. J. Serrano, A. R. Talcott, R. C. Wood, and M. Nemirovsky, "Performance estimation of multistreamed, superscalar processors," in *Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences*, pp. 195–204, 1994.

[133] T.-Y. Yeh, D. Marr, and Y. N. Patt, "Increasing the instruction fetch rate via multiple branch prediction and branch address cache," in *Proceedings of the International Conference on Supercomputing*, pp. 67–76, 1993.

[134] T.-Y. Yeh and Y. N. Patt, "Alternative implementations of two-level adaptive branch prediction," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 124–134, 1992.

[135] T.-Y. Yeh and Y. N. Patt, "Branch history table indexing to prevent pipeline bubbles in wide-issue superscalar processors," in *Proceedings of the 26th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 164–175, 1993.

[136] T.-Y. Yeh and Y. N. Patt, "A comparison of dynamic branch predictors that use two levels of branch history," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pp. 257–266, 1993.

[137] R. Yung, "Design decisions influencing the UltraSPARC's instruction fetch architecture," in *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, 1996.