MULTIPLE-PASS PIPELINING: ENHANCING IN-ORDER MICROARCHITECTURES TO OUT-OF-ORDER PERFORMANCE

BY

RONALD D. BARNES, JR.

B.S., University of Oklahoma, 1998
M.S., University of Illinois at Urbana-Champaign, 2002

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2005

Urbana, Illinois

ABSTRACT

Out-of-program-order execution has become almost a ubiquitous characteristic of modern processors because of its ability to tolerate variable memory-instruction latency. As designs are becoming increasingly power-conscious, the cost and complexity of the components of out-of-order execution are becoming problematic. Compilers have generally proven adept at planning useful static instruction-level parallelism, but relying solely on the compiler's instruction arrangement has been shown to perform poorly when cache misses occur. This work proposes two multiple-pass pipelining "flea-flicker" microarchitectural techniques, *two-pass pipelining* and *multipass pipelining*, both of which exploit a static compiler's meticulous scheduling as well as advance execution beyond otherwise stalled instructions without the complexity of true out-of-order execution.

With two-pass pipelining, programs execute on two in-order back-end pipelines coupled by a queue. The "advance" pipeline often defers instructions dispatching with unready operands rather than stalling. The "backup" pipeline allows concurrent resolution of instructions deferred by the first pipeline allowing overlapping of useful "advanced" execution with miss resolution. Multipass pipelining is based upon a similar

concept, but overcomes the shortfalls of two-pass pipelining through simultaneous execution of architectural and advance instructions on a common pipeline in a simultaneous multithreading-like fashion. These techniques perform similarly to achievable out-of-order designs while comparing favorably in terms of power and complexity. An accompanying compiler technique and instruction marking further enhances the handling of miss latencies and reduces fruitless speculative execution by statically denoting instructions that, when stalled, indicate there is little opportunity for advanced execution.

To Susan and Zach.

ACKNOWLEDGMENTS

I would first like to acknowledge the guidance of my adviser, Professor Wen-mei Hwu, and thank him for allowing me to pursue computer architecture studies in his IMPACT research group. My committee members, William Sanders, Sanjay Patel, and Craig Zilles, have also provided valuable feedback and encouragement along the way. I would especially like to recognize the valuable input in the development on two-pass pipelining gained through collaboration with Professor Patel.

I would also like to thank my wife, Susan, and my son, Zachary. Without their support and encouragement, I doubt I would have completed this work. They put up with my almost constant absence, and I appreciate their continued love and support.

I owe considerable thanks to the many past and current members of the IMPACT research group, whose work has contributed both this research and to my development as a researcher. Erik Nystrom also contributed significantly to the development of two-pass pipelining, and I am grateful for the many hours he dedicated to the creation of our simulation infrastructure. John Sias not only contributed to the two-pass pipelining design but provided invaluable help in the development and implementation of critical load identification described in this work. I would like to thank Professor Nacho Navarro

for his contribution to two-pass pipelining. Matthew Merten was an informal mentor for my initial years in the IMPACT group. I learned a great deal from Matthew and greatly enjoyed our collaboration on the Runtime Optimization Architecture. Hillery Hunter not only tolerated sharing an office with me, but she also provided useful feedback regarding power issues. I would like to thank Shane Ryoo for his help with last-minute deadlines. Finally I would like to thank David August, Dan Connors, Sain-Zee Ueng, Ian Steiner, James Player, and Chris Rodrigues for all of their immeasurable assistance over the years.

Lastly, I would like to recognize Hewlett-Packard, Advanced Micro Devices, Microsoft, the DARPA/MARCO Gigascale Systems Research Center, University of Illinois, and Department of Electrical and Computer Engineering for their financial support of my graduate studies.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

Out-of-order execution is a common microarchitectural strategy that allows the processor itself to determine how to efficiently order instruction execution. Under this execution model, the microarchitecture typically chooses multiple instructions for execution in parallel from a running program each clock cycle. The cost of long latency operations can be hidden by the concurrent execution of other instructions. Furthermore, since this selection is dynamic, the ordering of instruction execution can be adjusted to adapt to run-time conditions. Primarily because of this ability to adapt to run-time events (such as data-cache misses), out-of-order execution is used in the majority of high-performance microprocessors [1, 2, 3].

This general dynamic reordering ability effectively hides data cache miss delays, accommodating load latencies as they vary at run time by sustaining useful computation during cache misses. However, the mechanisms providing it replicate, at great expense, much work which was already done effectively at compile time. Aggressive register renaming [4] is used in out-of-order designs to eliminate output and antidependences that restrict the motion of instructions. This duplicates much of the effort of compile-time register allocation. Dynamic scheduling [5, 6, 7] relies on complex scheduling queues and

large instruction windows to find ready instructions, and in choosing the order of instruction execution based on dependencies, repeats the work of the compile-time scheduler. These mechanisms incur significant power consumption, add instruction pipeline latency, reduce predictability of performance, and occupy substantial additional chip real estate.

A static, in-order execution model would, in contrast, generally be less complex. Microarchitectures that execute instruction strictly according to the compiler's specified plan of execution [8] avoid the overheads that are associated with out-of-order execution. While compilers have generally proven adept at planning useful static instruction-level parallelism (ILP) for in-order microarchitectures, the efficient accommodation of unanticipable latencies, like those of load instructions, remains a vexing problem. The inability of in-order microarchitectures to adapt to cache-miss latency is a critical deficiency in in-order processors, because the execution time contribution of cache miss stall cycles is significant in the current generation of microprocessors and is expected to increase with the widening gap between processor and memory speeds [9].

This chapter will introduce multiple-pass "flea-flicker"[1] pipelining, microarchitectural techniques that exploit a static compiler's meticulous scheduling while also providing for *persistent*, advance execution beyond otherwise stalled instructions. Before elaborating on the proposed microarchitectural extensions, it is useful to delineate the opportunity

[1] In American football, the *flea-flicker* offense tries to catch the defense off guard with the addition of a forward pass to a lateral pass play. Defenders covering the ball carrier thus miss the tackle and, hopefully, the ensuing play. Multiple-pass pipelining utilizes two (or more) passes of preexecution/execution to achieve its performance efficacy.

exploited with the help of a case study in a modern instruction-set architecture, the Intel Itanium Architecture [10, 11, 12].

1.1 Case Study: Intel Itanium 2

Several instruction set architecture features have been proposed to offer the compiler features to enhancing instruction level parallelism. Large register files grant the broad computation restructuring ability needed to overlap the execution latency of instructions. Explicit control speculation features allow the compiler to mitigate control dependences, further increasing static scheduling freedom. Predication enables the compiler to optimize program decision and to overlap independent control constructs while minimizing code growth. In the absence of unanticipated run-time delays such as cache miss-induced stalls, the compiler can effectively use these features, utilizing execution resources, overlapping execution latencies, and working around execution constraints [8, 13]. For example, when run-time stall cycles are discounted, the Intel reference compiler (Intel ecc v.7.0) [14] at a high level of optimization (-O3 -ipo -prof_use) can achieve an average throughput of 2.5 instructions per cycle (IPC) across SPECint2000 [15] benchmarks for a 1.0GHz Itanium 2 processor with 3 MB of L3 cache.

The Itanium architecture is an implementation of Explicitly Parallel Instruction Computing (EPIC) in which the above mentioned instruction set features are provided to allow the compiler to effectively expose ILP to the microarchitecture. EPIC compilers express the program parallelism using an encoding technique in which groups of instructions

intended by the compiler to issue together in a single processor cycle are explicitly delimited. All instructions within such an "issue group" are guaranteed by the compiler to be free from true register dependences, and thus can be executed in parallel [11]. Because the Itanium 2 microarchitecture is in-order, if an issue group contains an instruction whose operands are not ready, the entire group and all groups behind it are stalled. This design accommodates wide instruction issue by reducing the complexity of the issue logic, but introduces the likelihood of "artificial"[2] dependences between instructions of unanticipated latency and instructions grouped with or subsequent to their consumers.

A large proportion of EPIC execution time is spent stalled waiting for data cache misses to return. For example, when SPECint2000 is compiled and executed as detailed above, 38% of execution cycles are consumed by data memory access-related stalls. Furthermore, depending on the benchmark, between 10% and 95% of these stall cycles are incurred due to accesses satisfied in the second-level cache, despite its having a latency of only five cycles. As suggested previously, the compiler's carefully generated, highly parallel schedule is being disrupted by the injection of many, unanticipated memory latencies. For a closer examination of performance issues on the Itanium 2 processor, readers are directed to [16].

[2]These dependences are artificial in the sense that they would not be observed in a dependence-graph based execution of the program's instructions, as in an out-of-order microprocessor.

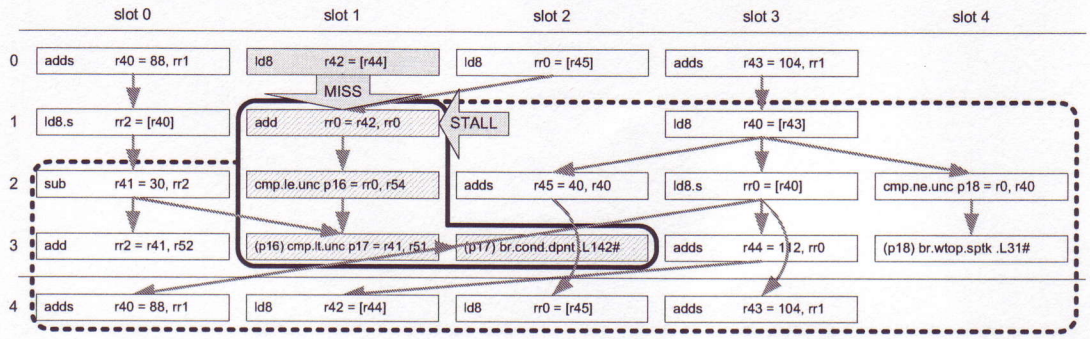


Figure 1.1 Cache miss stall and artificial dependences in *mcf*.

1.1.1 An example from *mcf*

Figure 1.1 shows an example from one of the most significant loops in the SPECint2000 benchmark with the most pronounced data cache problems, *mcf*. The figure, in which each row constitutes one issue group and arrows indicate data dependences, shows one loop iteration plus one issue group from the next. In a typical EPIC machine, on the indicated cache miss stall caused by the consumption of r42 in group 1, all subsequent instructions (dotted box) are prevented from issuing until the load is resolved, although only those instructions enclosed in the solid box are truly dependent on the cache miss. (Since the last of these is a branch, the instructions subsequent to the branch are, strictly speaking, control dependent on the cache miss, but a prediction effectively breaks this control dependence.) This severely hampers processor performance from its potential, evidenced by studies that have found that large numbers of useful instructions following a data cache miss are independent of the miss [17].

An out-of-order processor would begin the processing of instructions such as the load in slot 3 of group 1 during the miss latency, potentially overlapping multiple cache misses. To achieve such economy here, however, the compiler must explicitly schedule the code in such a way as to defer the stall (for example, by moving the consuming add after the load in slot 3 of group 1). In general, the compiler cannot anticipate statically which loads will miss and when. As load latencies vary dynamically, the best a compiler can hope to achieve is a schedule to maximize performance in the most common cases. A limited degree of dynamic execution could easily overcome this problem, but true out-of-order execution, in addition to unnecessarily complicating the design, would render an efficient implementation of EPIC features difficult to achieve. In particular, register renaming, a standard assumption of out-of-order design, very substantially complicates the implementation of predicated execution, a fundamental feature of EPIC design [18].

Run-ahead preexecution is one type of limited dynamic execution mechanism, in which instructions subsequent to the stalling instruction are executed during the stall to prepare microarchitectural state for more efficient execution of future instructions. A checkpointing-based, in-order preexecution mechanism (expanding on the ideas proposed by Dundas and Mudge [19, 20] can capitalize on the execution of instructions following a miss consumer, but it provides only a limited tolerance of cache miss, primarily through overlapping the handling of long, independent misses. The multiple-pass pipelining design provides much more effective tolerance of both long and short misses while continuing to allow efficient exploitation of the compiler's generally good schedule.

1.1.2 Addressing the data-cache stall problem with large caches

As described in the previous section, in-order processors like the Itanium 2 are unable to satisfactorily tolerate cache miss latencies. The compiler can attempt to hide foreseeable cache latency through explicit data prefetching, but compiler-directed prefetching [21, 22, 23] is typically only effective for anticipable, long-latency misses in applications with largely regular accesses. The main strategy available to in-order processor architects to address the cache miss tolerance deficiency is to increase cache sizes to reduce the average latency of a memory access. Not only is this approach expensive, but since only outer levels of cache can typically be made very large, this approach is only effective at dealing with distant cache misses. It does little to address the more diffuse stalls due to difficult-to-anticipate, first- or second-level misses that often thwart the effective exploitation of static schedules.

An attempt to address in-order cache-miss stalls with larger cache sizes is demonstrated within the case-studied Itanium Processor Family. Figure 1.1 charts the progression of Itanium processor generations over the 5 years since its introduction. Each line describes a particular implementation of the five generations of Itanium processors. Published Standard Performance Evaluation Corporation (SPEC) ratios [15] are given for two benchmarks, *mcf* and *gzip* for each implementation. As mentioned earlier in this chapter, the performance of *mcf* is dominated by cache misses. *gzip* on the other hand is one of the more compute-bound SPECint2000 benchmarks.

Table 1.1 Five generations of Itanium processors.

Generation	Frequency	Caches	Process	Transistors	*mcf*	*gzip*
First*	800 MHz	2-cycle 16 kB L1, 96 kB L2, 2 MB* L3	180 nm, 1.6 V	25 M	**187**	**332**
Second†	1 GHz	1-cycle 16 kB L1, 256 kB L2, 1.5 MB L3	180 nm, 1.5 V	221 M	**684**	**630**
Third†	1.3 GHz	1-cycle 16 kB L1, 256 kB L2, 3 MB L3	130 nm, 1.35 V		**1106**	**809**
	1.4 GHz	1-cycle 16 kB L1, 256 kB L2, 3 MB L3	130 nm, 1.35 V		**1114**	**872**
	1.5 GHz	1-cycle 16 kB L1, 256 kB L2, 6 MB L3	130 nm, 1.35 V	410 M	**1961**	**918**
‡	1.6 GHz	1-cycle 16 kB L1, 256 kB L2, 6 MB L3	130 nm, 1.35 V		**2340**	**1069**
Fourth‡	1.6 GHz	1-cycle 16 kB L1, 256 kB L2, 9 MB L3	130 nm, 1.35 V	592 M	**2665**	**1069**
Fifth◇	2.0 GHz	1-cycle 16 kB L1, 256 kB L2, 12 MB L3	90 nm, 1.2 V	1.72 B		

⋆ *off-chip*
◇ *announced*

∗ Intel C++ Compiler 5.0.1
† Intel C++ Compiler 8.0
‡ Intel C++ Compiler 8.1

The SPEC ratios in bold in Table 1.1 are the measure of performance of the corresponding benchmark. The effectiveness of the increasing cache sizes at addressing the problem of in-order cache stalls can be considered when examining the scaling of application performance with the increase of both frequency and cache sizes in each processor generation.

A significant performance increase was seen for both benchmarks moving from the first generation [24] of the Itanium family and the second generation [25] (the first implementation of the Itanium 2 processor). Not only did the frequency and caches improve from the first to second generations, but the core microarchitecture changed as well, with a reduction in the first-level data-cache latency, a reduction in the branch misprediction

resolution pipeline length, and an increase in the number of functional units. Additionally, compiler technology became much more capable since the first Itanium introduction.

More interesting, in terms of evaluating the effectiveness of increasing cache sizes, is the progress of the next three generations of the Itanium family, each of which are generations of the Itanium 2 processor. Moving from the second generation to the third [26], the core microarchitecture remained basically the same, while a 30% increase in frequency was seen, and the third level of cache increased from 1.5 MB to 3 MB for the implementations charted in Figure 1.1. The benchmark *mcf* saw a dramatic 1.6× increase in performance while *gzip*'s 1.28× speedup closely tracked the increased frequency. On the next charted implementation, the 1.4-GHz Itanium 2 with a 3-MB level-three cache, *mcf* saw virtually no improvement, while *gzip*'s speedup of 1.08× again closely tracked the frequency gain. This trend continued moving to higher-frequency and larger-cache designs up to the fourth Itanium family generation [27]. The published performance of these implementations on the benchmark *mcf* was seen to improve with increases in cache size and improvements in compiler technology (in particular, compiler-directed data-cache prefetching techniques [28]), while the performance on the more-compute-bound benchmark *gzip* much more closely tracked increases in frequency.

Table 1.1 demonstrates that the technique of adding more cache is effective at reducing the average latency of memory accesses for applications with large data footprints like *mcf*. However, increases in level-three cache size do little to improve the performance of benchmarks like *gzip*, even though, on the second generation of the Itanium family,

gzip spends 20% of its execution stalled on data cache misses [16]. Most of *gzip*'s misses are shorter misses. Additionally, while the average latency of a memory access in *mcf* has been reduced by the addition of large caches, the majority of *mcf*'s execution still remains spent stalled on lower-level cache misses. Further increases in the size of lower levels of cache are unlikely due to tight timing constraints.

As cache sizes have increased, microprocessor die area and transistor count has increasingly been dedicated to cache. Since the third generation of the Itanium family, the majority of chip area has been consumed by third-level cache. By the fourth generation, 2/3 of die area was consumed by cache. The announced fifth generation Itanium processor[3] [29] has 1.72 billion transistors, over 90% of which are cache. While process scaling has shrunk the absolute area of the core, the total die area has remained large (and actually increased for the fourth generation). This accretion in area due to cache impacts the cost of manufacturing each chip in a straightforward way.

Aside from the direct financial cost of maintaining large low-level caches and increasing the size of the outer-level of on-chip cache, this investment in cache has a substantial power cost as well. As process sizes and threshold voltages (V_t) shrink, transistor leakage current is becoming responsible for a greater amount of power consumption. The International Technology Roadmap for Semiconductors [30] predicts that leakage power will soon be the dominant source of microprocessor power consumption. As the overwhelming majority of on-chip transistors are in data caches, such caches are a prime

[3]This processor has two Itanium 2 cores, each with a 12-MB third-level cache.

source of leakage power [31]. In the first Itanium 2 implementation, all leakage power accounted for only 5% of total chip power consumption. In the next generation, leakage power increased to 21% of chip power (with 7% of chip power leaking from the third-level cache). Through the aggressive use of long channel and high V_t transistors [32], leakage power has been announced to have been held on average at 25% of total chip power in the latest Itanium generation [29] (with 5% of total power leaking from the third-level cache). More troubling, however, is that the total leakage power has been published to vary as much as 2× due to silicon processing variability.

Apart from area and power costs, the large numbers of cache transistors have other costs as well. As process sizes have shrunk, soft errors have become more likely, and the large numbers of cache transistors make the caches susceptible to such errors [33]. Enhanced error correction is needed to tolerate soft errors and maintain reliability [29]. Additionally, in nanometer technologies, processing variation makes failures in SRAM cells more common [33]. Adaptability to failures of cache lines is required to provide acceptable reliability [29].

As evidenced by the case study of the Itanium 2 processor, the technique of using large caches to reduce the average latency of a memory access, thus the susceptibility to in-order cache-miss stalls, is successful in improving performance for benchmarks with significant numbers of far misses, while doing little to reduce the cost of unpredictable, nearer misses (since increases in the size of the lower-latency, nearest levels of cache are

unlikely). Performance gains from the seemingly simple technique of increasing cache sizes comes at a high cost in terms of die area, power and design complexity.

Multipass pipelining addresses in-order processors' intolerance of cache misses by providing a more adaptable microarchitecture. Chapter 4 will show that this approach will enable equivalent performance with smaller cache sizes, and will continue to provide performance even at large cache sizes by tolerating the more frequent and less predictable short cache misses that interrupt even the best compiler-planned execution.

1.2 Introductory Example

Figure 1.2 demonstrates the enhanced execution behavior that multiple-pass pipelining techniques achieve over the traditional in-order execution model. The example in Figure 1.2 shows an execution and memory access time-line repeated for several different models of execution, each labeled on the left. For each model, the execution activity is divided into actual instruction execution (**EXE**) and the handling of data cache misses caused by executing load instructions (**MEM**). In the example, the **EXE** line represents many executing instructions, but a selected few instructions of interest for the purposes of the example are shown as lettered (`A-F`) points on the time-line. Instructions `A`, `C`, and `E` are load instructions that miss in the data cache. Two types of misses are shown in Figure 1.2: relatively long misses labeled **L2 MISS** and relatively short misses labeled **L1 MISS**. The handling of these misses is shown as the bold **MEM** component of the

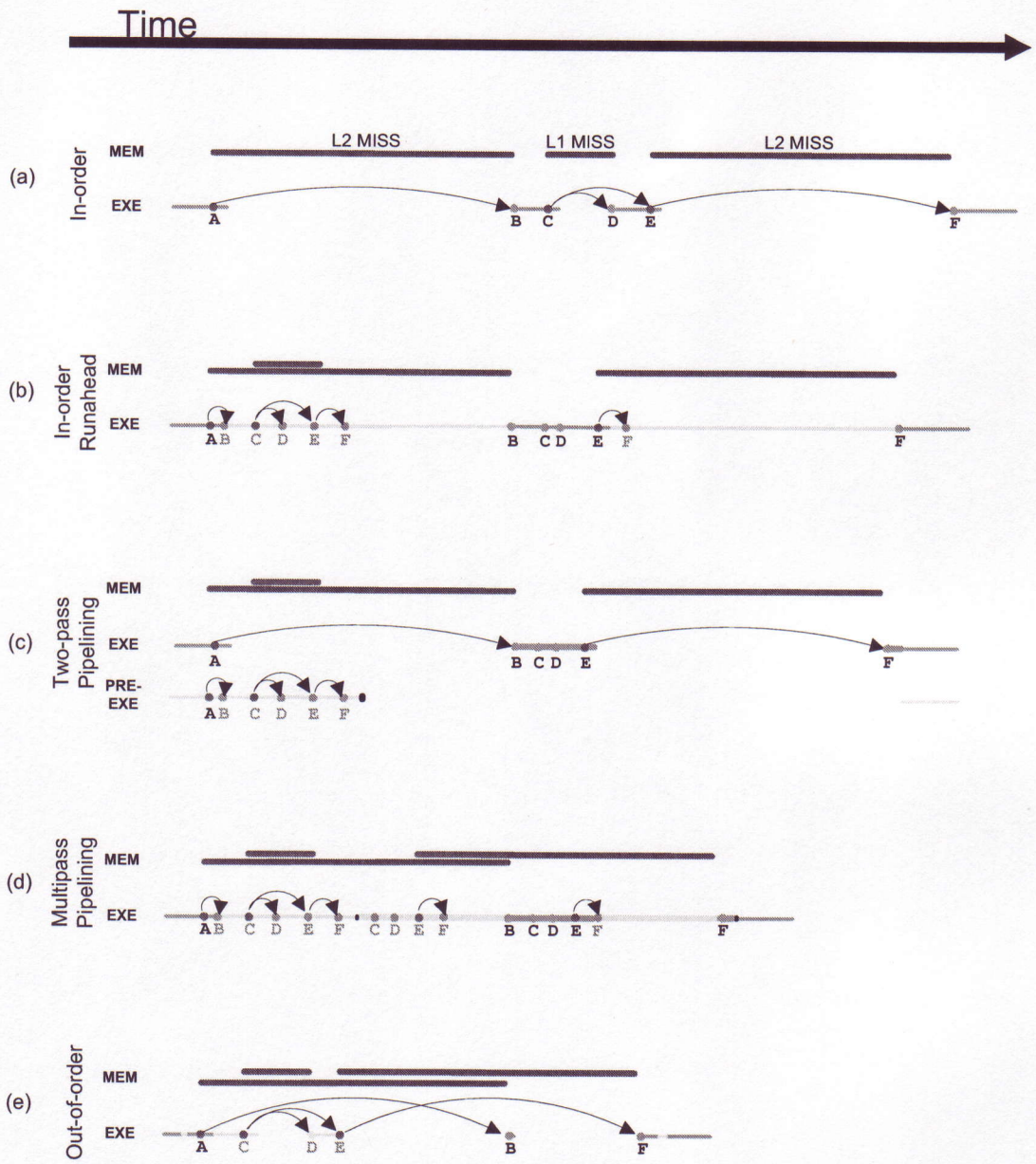


Figure 1.2 Execution and memory access time-line for five different instruction issue models. (a) In-order. (b) In-order runahead. (c) Two-pass pipelining. (d) Multipass pipelining. (e) Out-of-order.

time-line. Finally, interesting data dependences between these instructions are shown as arrows to the dependent instruction.

1.2.1 In-order time-line

The in-order time-line in Figure 1.2(a) reiterates the problem, accompanying in-order processors as detailed in Section 1.1, that instructions can become artificially, positionally stalled behind previous (in program order) load-miss-interlocked instructions. When instruction `A` misses in the data cache, its latency in lengthened by the handling of the cache miss. In the case of `A` the handling of the miss is relatively lengthy. Instructions that are independent of `A` continue to execute, until instruction `B`, the first consumer of load `A`, is reached. For the remaining duration of `A`'s miss, the in-order processor is stalled, represented in the gap in the **EXE** time-line before instruction `B`. Similarly, when `C`'s relatively short miss occurs, execution continues until instruction `D`, the first instruction dependent on `C`. Execution is stopped until the miss is handled and `D` can execute. This pattern is finally repeated for a third time with the interlock of `F` on the miss of `E`.

1.2.2 In-order runahead time-line

It is apparent from the in-order time-line that maintaining the static order of instruction execution prevents the handling of any of the example's misses in parallel, even though `C` and `E` are independent of `A`'s result. One straightforward approach to overlap the handling of cache misses that are blocked behind other interlocking instructions is to

allow execution to temporarily allow execution to run ahead past otherwise interlocked instructions in an attempt to use the subsequent independent instructions to generate data prefetches. This approach is mentioned in Section 1.1 as in-order, runahead preexecution. This continuation forward in the instruction stream is shown in Figure 1.2(b) as a faint continuation of the **EXE** line in the time-line beyond the first instance of instruction `B`.

The in-order runahead time-line in Figure 1.2(b) represents an aggressive form of the runahead preexecution described in [19]. In it, as in each of the time-lines, `A` misses in the cache. However, unlike that of the in-order time-line, execution continues beyond the consumer `B`. Since the destination register of `A` is one of the operands of `B`, `B` cannot compute any valid result. The execution of `B` is feigned, and it bypasses and writes its specially marked nonresult to its consumers and destination. Execution continuing past `B` reaches independent instruction `C`, which can thus begin the handling of its memory access, overlapping its access with that of `A`. This overlap is represented in Figure 1.2(b) by the overlapping bold lines in the **MEM** component of the time-line.

Just as `B` was skipped while it awaits an input value from memory, `D` is skipped because it requires the result of `C`'s memory access that is not available at time of `D`'s execution. Similarly, load `E` does not execute because it depends on the result of `C`. In the in-order runahead approach, execution proceeds beyond `E`, as a result also skipping `F`. In the example in Figure 1.2(b), no further independent misses can be accessed before the handling of `A`'s miss is complete. At that point the speculative, transitory runahead

period ends, and the execution time-line returns to instruction B. Normal execution begins again at that point, denoted with the reestablishment of the solid **EXE** line. All of the work done during the runahead execution (whether valid or invalid) is discarded with the restoration of normal execution. Instructions in the program execution, beginning with instruction B are processed again. When load instruction C is executed for a second time, its memory access is a hit in the cache, because its original miss was handled during runahead execution. Execution continues to instruction E. Since E's execution was skipped during runahead execution, its full miss latency is observed during the second attempt at its execution. Finally, because F is dependent on E, it is skipped to commence a second period of runahead execution. In this example, this period of execution does not incur any additional cache misses.

The goal of runahead preexecution is to use the predicted instruction stream beyond an otherwise interlocked instruction to initiate memory accesses in the hope of having a prefetching effect. In the example, instruction C is a hit in the cache during its actual execution because of this prefetching effect of its earlier precomputation. The length of the in-order runahead time-line is shorter than that of the in-order time-line by the length of the stall before D in the in-order case. This demonstrates the speedup of the runahead preexecution technique. However, two limitations of this prefetching mechanism are evident from the example time-line. The first is that once an instruction's execution is skipped, it will not be considered again for execution until normal execution begins again. In the example in Figure 1.2(b), D is skipped during runahead execution. Even

when C's miss is handled and D's operands are ready, D execution cannot be considered because execution has already proceeded farther down the instruction stream. The second limitation is that because runahead preexecution is only a prefetching technique, none of the results of the valid computation done during runahead execution are persistent. Rather than eventually becoming part of architectural state, these results are forgotten when normal execution returns.

1.2.3 Two-pass pipelining time-line

The third time-line in Figure 1.2 is that for a execution model called two-pass pipelining [34, 35]. This is the first implementation of multiple-pass pipelining. Like the simple in-order runahead approach, two-pass pipelining similarly uses preexecution of instructions beyond a potential in-order stall to accelerate normal execution. In two-pass pipelining, two fixed in-order pipelines are bridged by a first-in-first-out (FIFO) queue. The first, preexecution pipeline executes all instructions executes all instructions speculatively *without stalling.* This pipeline provides the preexecution seen in the in-order runahead model. As in the in-order runahead model, instructions dispatching without an input operand ready (because of a cache miss) are suppressed rather than incurring stalls, while other instructions execute normally. The second, execution pipeline executes instructions deferred in the preexecution pipeline and incorporates all results in a consistent order. This two-pipe structure allows cache miss latencies incurred in one

pipe to overlap with independent execution and cache miss latencies in the other while preserving in-order semantics in each.

In the example for two-pass pipelining in Figure 1.2(c), preexecution is shown as the faint **PRE-EXE** component of the two-pass pipelining time-line. In this execution model, the handling of `A`'s cache miss is initiated in the preexecution pipeline, while the execution of instruction `B` is subsequently skipped. This allows the handling of `C`'s miss to be initiated with its precomputation and thus overlapped with the miss of `A`. Like the preexecution in Section 1.2.2, the two-pass pipelining's preexecution skips instructions `D`, `E` and `F`. In the in-order runahead model, preexecution was abandoned whenever normal execution could occur. However, preexecution under two-pass pipelining is occurring in a separate pipeline from the actual execution. This preexecution will continue with its state contaminated by the nonresults of the skipped instructions until some external event causes the preexecution and execution pipelines to become synchronized. To avoid synchronizing these pipelines frequently, the preexecution in two-pass pipelining is controlled to avoid too much of its state being contaminated with nonresults preventing useful preexecution from occurring [35]. In the example, this control is exerted at the dark point ending preexecution after instruction `F`. This control halts preexecution while it awaits a critical result. Preexecution resumes right before the end of the two-pass pipelining example.

In the execution pipeline, shown with the solid **EXE** component of the time-line, the handling of `A`'s miss is not complete when `B` is reached. The **EXE** pipeline stalls until

B can execute, because this pipeline behaves just as the in-order pipeline described in Section 1.2.1. However, once the miss is handled, the instructions in the next segment of the **EXE** pipeline have all been preprocessed in the **PRE-EXE** pipeline. The results of preexecuted instructions (such as instruction C) are queued for incorporation in the **EXE** pipeline. The reuse of many preexecution results is marked in the two-pass pipelining time-line with the widening of the **EXE** line. Only instructions skipped during preexecution must be computed as normal execution. Additionally, since many results have been precomputed, a technique known as *instruction regrouping* can be used to accelerate execution while maintaining instruction order. This technique is described in Section 2.2.8.

As in the in-order runahead example described in Section 1.2.2, E was not successfully preexecuted. The handling E's miss is started in the **EXE** pipeline, and thus F must await E's result, stalling the normal execution. However, while F's execution was irrevocably deferred to the **EXE** pipeline, by halting preexecution to wait on critical results, the **PRE-EXE** pipeline is spared from allowing invalid nonresults to completely contaminate fruitful preexecution until an event synchronizes **PRE-EXE** and **EXE**. This halting is shown with the dark point terminating the first **PRE-EXE** segment in Figure 1.2(c).

The two-pass approach has two important advantages over the in-order runahead model. First, since preexecution and execution are performed on separate pipelines, these actions can occur concurrently. Misses and execution initiated on both pipelines can be overlapped with the other. The second advantage is that through the coupling

of **PRE-EXE** results to the **EXE** pipeline, valid preexecution results are preserved as mentioned above.

1.2.4 Multipass pipelining time-line

Figure 1.2(d) shows the execution profile for the second multiple-pass pipelining technique, multipass pipelining [36]. This model of execution uses a technique similar to that used in two-pass pipelining, but with both execution and preexecution occurring on the same in-order pipeline.

The behavior of the multipass approach at first appears similar to that of the simple in-order runahead model described in Section 1.2.2. When `A` misses in the data cache, instruction `B`'s execution is deferred, and preexecution proceeds in place of the interlock on the cache-miss result. This again is shown with the faint **EXE** line segment in the example. Preexecution continues on to initiate `C`'s short cache miss. As instruction computation is suppressed because of unavailable source operands, the preexecution state may become so contaminated that continued advance execution is fruitless. Rather than wasting preexecution effort in this case, preexecution is restarted whenever an instruction producing a critical result is not ready for execution. This concept of criticality is shared with the two-pass approach as explained in Section 1.2.3. The selection of instructions whose results are critical is detailed in Chapter 3.

By restarting preexecution at instruction `B` (which is still not ready), preexecution might be able to perform instructions which were not ready for execution on an earlier

pass. In the example in Figure 1.2(d), load instruction `E` could not preexecute initially because it was the dependent on missing load `C`. For the same reasons, `E` did not preexecute in the in-order runahead example shown in Figure 1.2(b). However, once preexecution begins again in the example at the dark point following instruction `F`, `C`'s miss has been handled and `D` and `F` can now execute. `F`'s preexecution allows the handling of its miss to overlap with that of `A` in a way that was not possible in the previously described models.

The example in Figure 1.2(d) demonstrates the trade-off between continuing preexecution and restarting with the first suppressed instruction. In this case, had preexecution continued, it was unlike to perform useful work because a critical result (needed directly or indirectly by a significant number of instructions) was not ready. Had preexecution not restarted, instruction `E` would have sat unexecuted until normal execution began again.

The use of criticality to restart preexecution is an obvious improvement over the two-pass approach described in the previous section, where, in the two-pass model, criticality was used to halt preexecution rather than reinitiating it. In addition, since multipass pipelining is not tied to a fixed number of pipelines, preexecution can occur for any miss, even if it is on a dependent chain from another miss. In the case of two-pass pipelining, once an instruction has produced an invalid nonresult, none of its dependents will preexecute, forcing subsequent misses in dependent chains to the execution pipeline. Like in the two-pass approach, in multipass pipelining, valid preexecuted results are preserved and eventually incorporated into execution state. This result reuse is similarly

used to accelerate execution. The only important advantage of the two-pass technique was that since it took advantage of two physical pipelines, it could overlap preexecution with normal execution in a straightforward way.

1.2.5 Out-of-order time-line

Out-of-order execution is a general dynamic strategy for instruction execution in which instructions are executed as their data dependencies are met. Figure 1.2(e) shows the example time-line for an out-of-order execution model. Because the compiler's arrangement exposed the available ILP, instructions are initially executed in their original, nonspeculative order shown with the initial, solid **EXE** component of the time-line. As more instructions become ready for execution, the out-of-order microarchitecture can occasionally reorder instructions to exploit more ILP than expressed in the compiler ordering. Speculative orderings of instructions are represented by the faint segments of **EXE** component of the time-line. However, because the compiler arranged instructions according the visible parallelism and the predicted available resources, instructions execute largely in the compiler-specified order in the absence of perturbative run-time effects.

When cache misses occur, such as that caused by instruction `A`, out-of-order executions dynamic schedule allows execution of all independent instructions in the program's instruction stream while the cache miss is being serviced. In Figure 1.2(e), this allows instruction `C` to be eventually issued speculatively, overlapping `C`'s miss with that of `A`.

Soon, no ready instructions can be found for execution because of the outstanding cache misses, shown with the gap in the **EXE** line. In the example, C's miss is quickly handled and new ready instructions are executed, including load instruction E. E's miss is then overlapped with that of A. When A's miss is finally handled, this causes several of the oldest instructions in program order to be ready for execution. Execution returns to mostly in-program-order once all cache misses are handled.

It is important to note that in the out-of-order model, since instruction are executing as soon as they are ready, instruction E executes almost immediately after C's miss is over. Instruction E is woken up for execution once all of its source operands are ready. In the multipass approach, E was executed only after preexecution returned to process instructions again a second time under A's miss. Because of the delay in reaching E, E's miss is overlapped to a lesser degree in the multipass pipelining approach than in the true out-of-order execution approach. The performance advantage of greedily executing instructions as they are ready is seen with the shortened execution time-line for the out-of-order approach.

1.3 Contributions

In this dissertation, I show that multiple-pass pipelining is a microarchitectural technique that provides beneficial tolerance of the unanticipable latency of data-cache accesses through preexecution of instructions positionally blocked behind a data-cache-interlocked

instruction. I show that the "flea-flicker" technique of performing multiple passes of *persistent* in-order execution achieves this tolerance by overlapping independent data-cache accesses and other useful work while maintaining a mostly in-order model of execution.

This work will focus on three key contributions over previous work. First, I show that multiple-pass pipelining outperforms other preexecution techniques, and approaches the performance of more general out-of-order approaches while avoiding the complexity that accompanies out-of-order designs. Second, I demonstrate the use of compiler-applied hints to direct preexecution by informing the microarchitecture which instructions produce operands are "critical." I provide a simple, but well-performing heuristic for statically applying these hints, and through the exploitation of these hints achieve significant speedups. Finally, my proposed multiple-pass pipelining techniques preserve the results of correctly preexecuted instructions to improve efficiency, hide the latency of multiple-cycle instructions and, most significantly, accelerate execution by using preexecuted results (and nonresults) to break dependencies between instructions and form new issue groups without reordering instructions.

The two-pass and multipass pipelining microarchitectures are presented in Chapter 2. The use of application of the compiler-applied criticality hints is explored in Chapter 3. An evaluation of the implementation costs and a comparison of the costs with the more-general implementations of out-of-order execution are examined in Chapter 5. An overview of related work is given in Chapter 6, and conclusions and future work are outlined in Chapter 7.

2 ARCHITECTURE

The multiple-pass pipelining schemes are designed to allow the productive processing of independent instructions during the memory stall cycles left exposed in traditional in-order pipelines. Figure 2.1(a) shows a snapshot of instructions executing on a stylized representation of a general in-order processor, such as Intel's Itanium [12, 37] or Sun's SPARC [38]. In the figure, the youngest instructions are at the left; each column represents an issue group. A dependence checker determines if instructions have ready operands and are therefore ready to be dispatched to the execution engine. If any instruction is found not to be ready, its issue group is stalled. The darkened instructions in the dependence check stage and incoming instruction queue are dependent on the stall, as indicated by arrows; the light-colored instructions, on the other hand, are data-flow-independent but nonetheless stymied by the machine's in-order execution of instructions.

Figures 2.1(b) and (c) show the proposed alternative. For the purposes of this example, the "flea-flicker" multiple-pass behavior is shown in context of exactly two passes, but this example is useful for introducing both two-pass and multipass pipelining. In

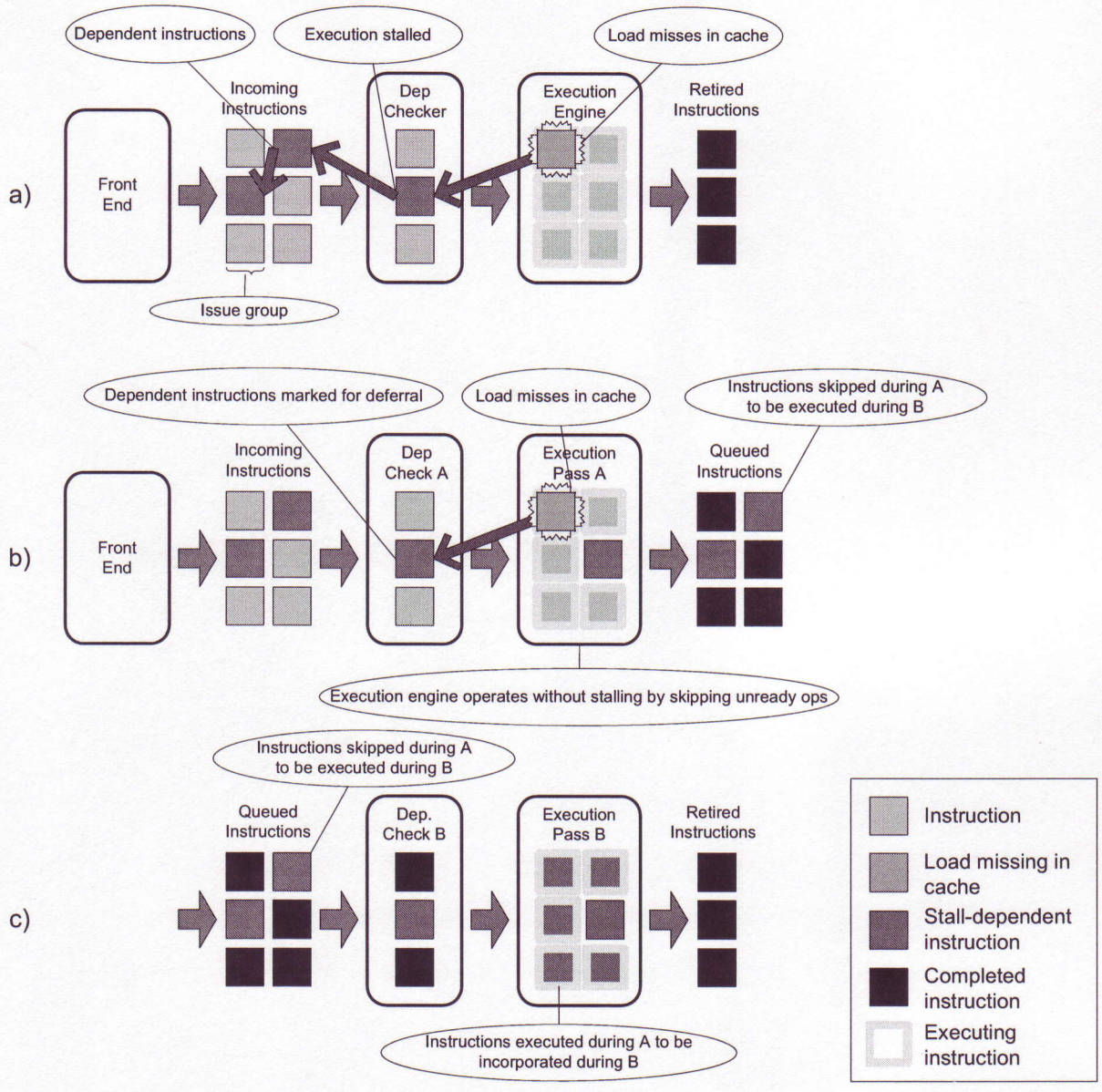


Figure 2.1 Snapshots of executions. (a) Original in-order processor; (b) multiple-pass pipelining–first pass; and (c) multiple-pass pipelining–second pass.

Figure 2.1(b), when an input operand of a "noncritical"[1] instruction is not ready at dependence check, the processor, rather than stalling, marks the instruction and all *dependent* successors (as they arrive, through a propagation of "invalid" bits in the bypass network and register file) as deferred instructions. These are skipped by the first (A) execution pass. Subsequent *independent* instructions, however, continue to execute in pass A. Deferred instructions are queued up for processing in the second (B) pass. Between passes, instructions shown as blackened have begun execution; execution of the remaining instructions has been deferred under the A pass due to unavailable operands.

These will execute for the first time in during the B pass, shown in Figure 2.1(c), when their operands are ready. The B pass also incorporates into architectural state the results of instructions previously resolved in A. In the case of long- or undetermined-latency instructions, such as loads, an instruction begun in the B pass may not be finished executing when its results are demanded during B; however, in this example, all instructions which began execution during A have completed by the B pass.

2.1 Baseline In-Order Microarchitecture

Before presenting the implementation details of the multiple-pass pipelining approaches, the baseline microarchitecture, to which the multiple-pass pipelining schemes

[1]The idea of criticality was introduced in Chapter 1. The use of criticality will be expounded upon in Sections 2.2.7 and 2.3.9. The determination of criticality is explained in Chapter 3.

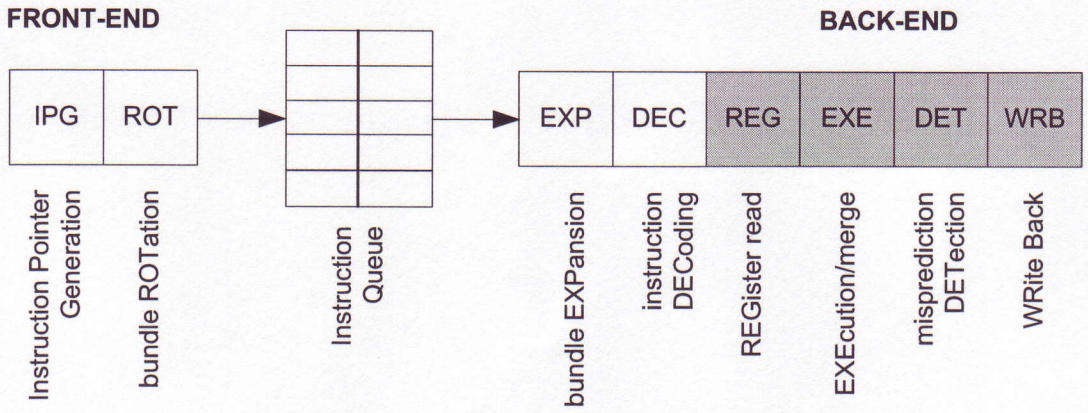


Figure 2.2 Intel Itanium 2 integer pipeline.

will be applied, will first be introduced. The baseline integer pipeline, presented in Figure 2.2, was chosen to closely match that of the Intel Itanium 2 [11], a contemporary high-performance in-order processor.[2]

The front-end portion of the pipeline,(instruction pointer generation (IPG), bundle rotation (ROT), bundle expansion (EXP), and instruction decoding (DEC), prepares instructions for execution on the many functional units of the execution core. As in the Itanium 2 pipeline, a 24-instruction buffer decouples instruction fetch from decode and execution, allowing the fetch of several cycles' worth of instructions to proceed during back-end execution pipeline stalls.

Once instructions reach the back-end portion of the pipeline, shown as the darkened stages in Figure 2.2, their operands are read from the register file or bypass network in the register read (REG) stage. In-order machines stall in this stage if the current issue group

[2]Although the proposed pipeline scheme is described in the context of implementing the Intel Itanium Architecture, it could be applied similarly to other typically in-order architectures such as SPARC [38], ARM, and TI C6x.

contains instructions with unready operands (often as the result of outstanding data-cache misses). In a typical machine, these stalls consume a large fraction of execution cycles. Instructions, once ready, execute (**EXE**), detect exceptions and mispredictions (**DET**), and finally write their register results in the register file (**WRB**). The back-end floating point and memory pipeline are similarly considered in this work, assuming pipeline lengths reflecting latencies described in Chapter 4.

2.2 The Two-Pass Pipeline Microarchitecture

This section presents the first implementation of multiple-pass pipelining, *two-pass pipelining*, in which execution passes are performed in two separate pipelines. In the terms of the example in Figure 2.1, execution of the B pass always stalls when an instruction's operands are not ready, because there is no pipeline in which to perform a "C" pass. These instructions may be unready either because they depend on a long-latency instruction whose executed started during the B pass, or because they depend on a long-latency instruction, started during the A pass, that has yet to complete. The latter situation is handled through the coupling mechanism described in the Section 2.2.1.

The two-pass arrangement effectively separates the execution of the program into two concurrently executing streams: an "advance" stream comprising instructions whose operands are readily available at the first dispatch opportunity and those that are "critical," and a "backup" stream encompassing the remainder. This section describes the

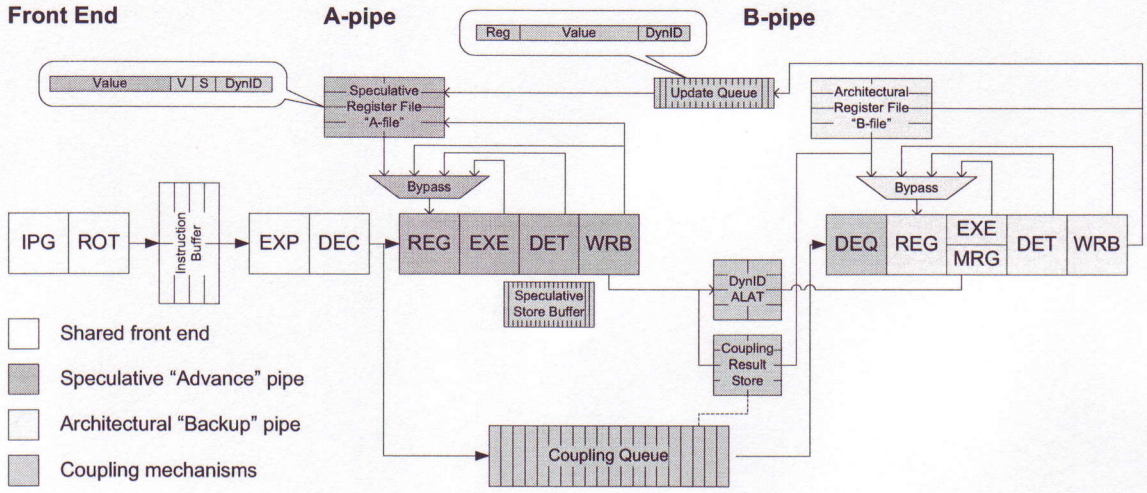


Figure 2.3 Two-pass pipeline design.

proposed pipeline scheme in detail, focusing on the management of the two streams to maintain correctness as well as to maximize efficiency and concurrency.

2.2.1 Basic mode of two-pass operation

Figure 2.3 shows a potential design of the proposed mechanism as applied to the Itanium 2-like pipeline introduced in Section 2.1. Again, while both multiple-pass approaches will be presented in the context of an EPIC pipeline, the strategies are applicable to any in-order processor. The reader will note marked similarities between the front-end and architectural pipelines with the front-end and back-end pipelines of the Itanium 2-like baseline pipelines. The remaining portions of the figure show the additions necessary to implement the proposed two-pass pipeline scheme.

The speculative pipeline in Figure 2.3, referred to as the A-pipe, executes instructions on an issue group by issue group basis. Operands are read from the register file or bypass

network in the **REG** stage. In the proposed two-pass scheme, noncritical instructions found to be unready in the **REG** stage do not stall the A-pipe; these instructions and their data-dependent successors are instead suppressed. Any subsequent, independent instructions are allowed to execute in the (**EXE**) stage, and subsequently detect exceptions and mispredictions (**DET**), and write their register results in an A-file (**WRB**).

The portion of Figure 2.3 referred to as the B-pipe completes the execution of those instructions deferred in the A-pipe and integrates the execution results of both pipes into architectural updates. The A and B pipes are largely decoupled (e.g., there are no bypassing paths between them), contributing to the simplicity of this design. The B-pipe stage **DEQ** receives incoming instructions from the coupling queue (CQ), as shown in Figure 2.3. The coupling queue receives decoded instructions as they proceed, in order, from the processor front end. When an instruction is entered into CQ, an entry is reserved in the coupling result store (CRS) for each of its results (including, for stores, the value to be stored to memory). Instructions deferred in the A-pipe are marked as deferred in CQ and their corresponding CRS entries are marked as invalid. The B-pipe completes the execution of these deferred instructions. An optional update queue is also shown in Figure 2.3. If utilized, this queue carries the results of B-pipe executions to be potentially merged into the register state of A-pipe in an attempt to increase the frequency of nondeferred executions there, as described in Section 2.2.6.

When, on the other hand, instructions complete normally in the A-pipe, their results are written both to the A-file (if the target register has not been reused) and to the CRS.

These "precomputed" values are incorporated in the merge (**MRG**) stage of the B-pipe, to be bypassed to other B-pipe instructions and written into the architectural B-file as appropriate. The destinations of instructions whose execution began in the A-pipe but is not yet complete will be scoreboarded, just as the destinations of long- or variable-latency instructions are normally scoreboarded. This handles "dangling dependences" due to instructions that begin in the the A-pipe but are not complete when they reach the B-pipe through the coupling queue. Such instructions are allowed to dispatch in the B-pipe, but with scoreboarded destinations, to be unblocked when results arrive from the A-pipe. Through this mechanism, the need to reexecute (in the B-pipe) instructions successfully preexecuted (or even prestarted) in the A-pipe is obviated. This has two effects: first, it reduces the energy-inefficient redundant execution and reduces pressure on critical resources such as the memory subsystem interface; second, the latency of long-latency, such as multiply instructions, is hidden; third, since these preexecuted instructions are free of input dependences when they arrive in the B-pipe, an opportunity for height reduction optimizations is created. In an optimization described in Section 2.2.8, the B-pipe dispatch logic can re-group (without reordering) instructions as they are dequeued, allowing instructions from adjacent, independent instruction groups available at the end of the queue to issue together as machine resources allow.

As an example of the concurrency exposed by the two-pass technique, Figure 2.4 shows four successive stages of execution of the code of Figure 1.1 on the two-pass system. Instructions flow in vertical issue groups from the front end on the left into the A-pipe and

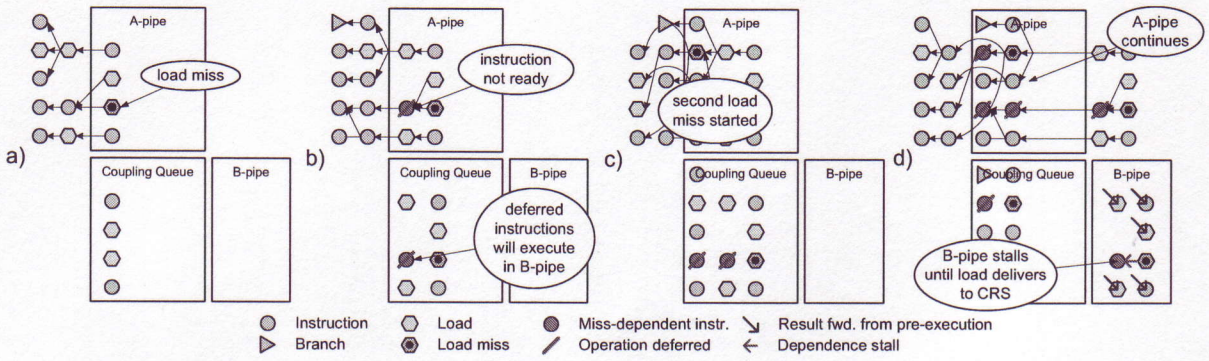


Figure 2.4 Applying two-pass pipelining to the previous *mcf* example.

coupling queue, and then into the B-pipe. In Figure 2.4(a), a load issues in the A-pipe and misses in the first-level cache. In (b), a dependent, noncritical instruction dispatches in the A-pipe. Since its operands are not ready, it is deferred and marked as such in the queue. A typical in-order pipeline would have stalled issue rather than dispatch this instruction. In (c), an additional dependent instruction is marked and a second cache miss occurs in the A-pipe. The concurrent processing of the two misses is enabled by the two-pass system. Finally, in (d), two groups have retired from the A-pipe and reexecution has begun in the B-pipe. Many preexecuted instructions assume the values produced in the A-pipe, as propagated through the coupling result store. The original cache miss, on the other hand, is still being resolved, and the inherited dependence causes a stall of the B-pipe. During this event, provided there is room in the coupling queue and result store, the A-pipe is still free to continue preexecuting independent instructions.

2.2.2 Critical two-pass design issues

Ensuring correctness and efficiency in the two-pass design requires the careful consideration of a number of issues. Chief among these is the fact that the B-pipe "trusts" the A-pipe in most situations to have executed instructions correctly; that is, the B-pipe does not confirm or reexecute instructions begun in the A-pipe, but merely incorporates their results. First, this entails that the A-pipe must accurately determine which instructions may be preexecuted and which must be deferred and must ensure that the A-file contains correct values for valid registers, even though write-after-write (WAW) stall conditions and other constraints typical to EPIC systems are to be relaxed. Second, the proper (effective) ordering of loads and stores must be maintained, even though they are executed partially in the A-pipe and partially in the B-pipe. Finally, as the new pipe includes two stages at which the correct direction of a branch may be ascertained, the misprediction flush routine needs to be augmented. Similar issues will have to be considered for multiple-pass pipelining. In the case of allowing WAW issue in two-pass pipelining, the solution I first introduced in [34] is explained in Section 2.2.3. This mechanism is tied to support for updates through the optional update queue. However, the simpler approach used for multipass pipelining can be applied to two-pass pipelining achieving similar results if the update queue is not used as examined in Chapter 4. The following sections investigate the three issues listed in this section in further detail.

2.2.3 Maintaining the A-file

The A-file, a speculative register file, operates in a manner somewhat unconventional to in-order EPIC designs, as delinquent instructions can write "invalid" results and WAW dependences are not enforced by the A-pipe through the imposition of stalls (this is legitimate only because the B-file is the actual architectural pipeline in which all register writes must be visible). Each register in the A-file is accompanied by a "valid" bit (V), set on the write of a computed result and cleared in the destination of an instruction whose result cannot be computed in the A-pipe; each also has a "speculative" bit (S), set when an A-pipe instruction writes a result and reset when an update from the B-pipe arrives. Under the design presented in [34], an additional "DynID" is added to each register. This tag indicates the ID of the last dynamic instruction to write the register, sufficiently large to guarantee uniqueness within the machine at any given moment. The V bit supports the determination in the A-pipe of whether an instruction either has all operands available, and therefore may execute normally, or relies on an instruction deferred into the B-pipe, and therefore must also be deferred to the B-pipe. The S bit marks those values written by the A-pipe but not yet committed by the B-pipe. All data so marked is speculative, since, for example, a mispredicted branch might resolve in the B-pipe and invalidate subsequent instructions in the queue and the A-pipe, including ones that have already written to the A-file. This bit supports a partial update of the A-file as an optimization of the B-pipe flush routine, as discussed later in Section 2.2.8. Finally, the dynamic ID tag (DynID) serves to allow the speculative issue of instructions

that are output-dependent on previously issued instructions which have have completed, and can optionally be used to allow updates to the A-file from instruction execution in the B-pipe.

Modern in-order processors allow instructions to potentially retire out of order. In order to maintain the proper order of writes to the same register, long-latency instructions scoreboard their destination so that future writers will suffer a WAW stall. Because such stalls would greatly interfere with preexecution, WAW stalls are not enforced. Instead, instructions executing (or being deferred) in the **EXE** stage will write their DynID to the tag for the register entry corresponding to their register destination. Only instructions writing back to register file entries matching their DynID tag are allowed to modify the register value, enabling writes only from the last issued writer. The same mechanism serves to allow the selective update of the A-file with results of retiring B-pipe instructions through the update queue. Entries in the A-file can only be updated by B-pipe retirement if its outstanding invalidation was by the particular instruction retiring in the B-pipe on its deferral in the A-pipe and described in Section 2.2.6.

The DynID mechanism was first introduced in [34] and was largely tied to the desire to allow updates of B-pipe execution to provide for future A-pipe instructions. A somewhat different approach is utilized in the multipass pipelining technique, avoiding the DynID complexity added to the register file, but requiring a mechanism integrated into the data-cache access pipeline. If B-pipe→A-pipe updates are not supported, this approach would be appropriate for two-pass pipelining.

2.2.4 Preserving a correct and efficient memory interface

Since the two-pass model can allow memory accesses to be performed out of order, the system must do some bookkeeping, beyond what is ordinarily required to implement consistency semantics, to preserve a consistent view of memory. For the purposes of presenting this issue, consider representative pairs of accesses that end up having overlapping access addresses, but where program order is violated by the deferral of the first instruction to the B-pipe. The term α indicates an instruction that executes in the A-pipe and β one that executes in the B-pipe. Three dependence cases are of interest, as follows. Seemingly violated antidependences $\mathrm{ld}[addr]^{\beta} \xrightarrow{A} \mathrm{st}[addr]^{\alpha}$ and output dependences $\mathrm{st}[addr]^{\beta} \xrightarrow{O} \mathrm{st}[addr]^{\alpha}$ are resolved correctly because loads and stores executing in the B-pipe do so with respect to architectural state and that stores executing in the A-pipe do not commit to this state, but rather write only to a speculative store buffer (an almost ubiquitous microarchitectural element), for forwarding to A-pipe loads. When A-pipe stores reach the B-pipe, their results are committed, in order with other memory instructions, to architectural state.

Preserving a flow dependence $\mathrm{st}[addr]^{\beta} \xrightarrow{F} \mathrm{ld}[addr]^{\alpha}$ requires more effort. In this case a store in B-pipe needs to forward its stored value to a load in A-pipe. As indicated earlier, the general assumption is that instructions executed in the A-pipe either return correct values or are deferred. If, when the store passes through the A-pipe, it has unknown data but is to a known (ready) address, the memory subsystem can defer the load, causing it to execute correctly in the B-pipe, after the forwarding store. If, however, the store

address is not known in the A-pipe, it cannot be determined in the A-pipe if the value loaded should have been forwarded or not. If a load executes in the A-pipe without observing a previous, conflicting store, the B-pipe must detect this situation and take measures to correct what (speculative) state has been corrupted. Fortunately, a device developed in support of explicit data speculation in EPIC machines, the Advanced Load Alias Table (ALAT) [39, 40] can be adapted to allow the B-pipe to detect when it is possible for such a violation to have occurred (since all stores are buffered, memory has not been corrupted).

With a traditional ALAT, an advanced load writes an entry into an ALAT, a store deletes entries with overlapping addresses, and a check instructions determines if the entry created by the advanced load remains [40]. In two-pass pipelining, loads executed in the A-pipe create ALAT entries (indexed by DynID rather than by destination register), stores executed in the B-pipe delete entries, and the merger of load results into the B-pipe checks the ALAT to ensure that a conflicting store has not intervened since the execution of the load in the A-pipe. If such a store has occurred, as indicated by a missing ALAT entry, corrupted speculative state must be flushed (conservatively, all instructions subsequent to the load and all marked speculative entries in the A-file), the A-file must be restored from the architectural copy, and execution must resume with the offending load.

Figure 2.5 demonstrates a simple example of this operation. In Figure 2.5(a), instruction `1`, executing in A-pipe writes the address of its access (**A**) to an appropriate

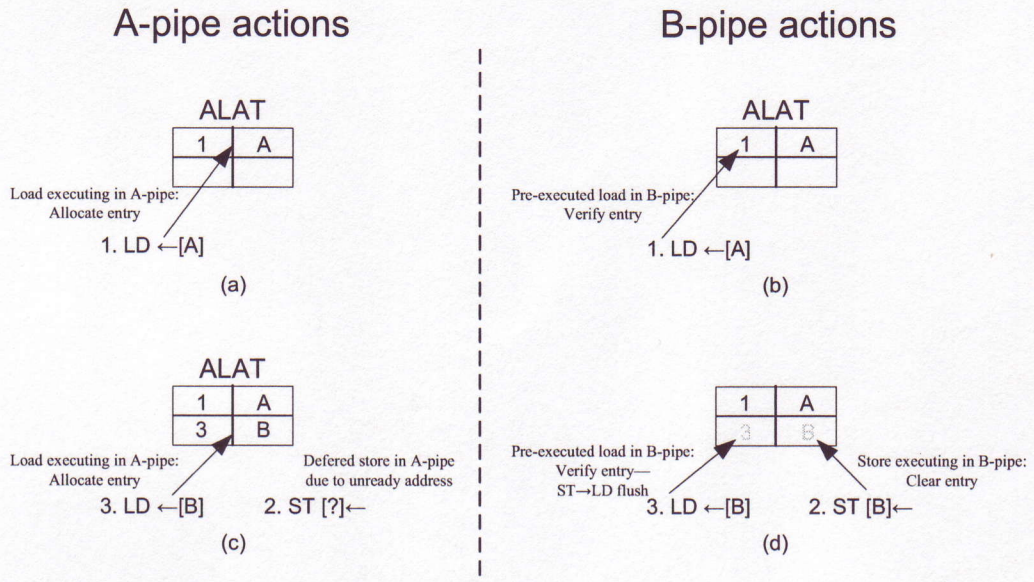


Figure 2.5 Two-pass pipelining Advanced Load Alias Table (ALAT) operation.

ALAT entry. When 1 is processed in B-pipe, in Figure 2.5(b), it verifies that no store, deferred in A-pipe but subsequently executed in B-pipe, has conflicted with 1's initial access. Figure 2.5(c) shows two instructions in A-pipe. The first, instruction 2, a store, is deferred because its address was not ready. The second, instruction 3, a load writes the address of its access **B** to an ALAT entry, just as 1 did in Figure 2.5(a). When instructions 2 and 3 are processed in B-pipe in Figure 2.5(d), store instruction 2 first evicts any entry in the ALAT with a conflicting address because 2 is executing in B-pipe due to its address being unready at the time of its A-pipe opportunity. Load instruction 3, upon finding no corresponding entry in the ALAT, has detected that a conflicting store has intervened since execution, causing a pipeline flush.

Since such pipeline flushes can have a detrimental performance effect, Chapter 4 reflects results indicating the infrequency of these events. It should also be noted that this ALAT is distinct from any architectural ALAT for explicit data speculation, and that, because of its cachelike nature, the ALAT carries the (small) possibility of false-positive conflict detections.

2.2.5 Limitation of miss resilience of two-pass approach

The two-pass model tolerates a miss by passing dependent consumers to the B-pipe for subsequent execution. Each such "tolerated" miss event potentially denies a dependent chain of operations the ability to tolerate misses, since operations executing in B-pipe have no further pipeline to which to defer. This "single-second-chance" phenomenon limits the ability of two-pass to approach ideal out-of-order levels of performance and, if a deferred miss delays detection of a branch misprediction, may even degrade performance relative to traditional in-order execution. This section illustrates this problem, and the following section explains how two mechanisms mitigate these effects: B→A update and compiler-based critical operation identification.

To illustrate the single-second-chance phenomenon, Figure 2.6 shows a dependence graph including five issue groups, totaling 13 instructions. In the graph, each edge is marked with the dependence latency. Nonunit latencies on edges A→C, D→H, F→J, and K→M indicate cache misses; each of these pairs, however, is scheduled at hit latency (1 cycle). When A's result is not available for C's dispatch, C is deferred to B-pipe, as will be

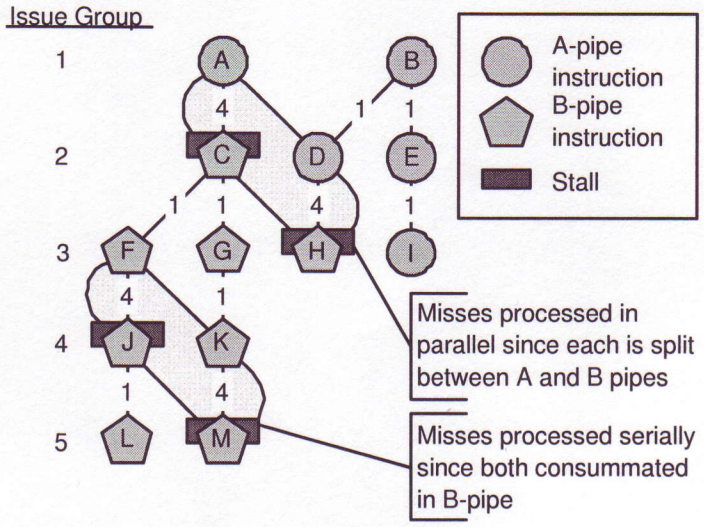


Figure 2.6 Limitation on overlap of latency events.

all its data-flow successors F, G, and so on. Because C deferred instead of stalling A-pipe, D and dependents can immediately begin execution in A-pipe, performing independent execution while C waits in B-pipe for its operand to be resolved. Miss latencies A→C and D→H are thus overlapped. Since instructions F and K execute in B-pipe, however, their unscheduled latencies to J and M, respectively, are exposed and serialized by B-pipe's in-order semantics; as deferred instructions they cannot benefit from two-pass. Here, a greedy deferral strategy that never stalls the A-pipe would be outperformed by a strategy that sometimes chooses to expose a miss cost in A-pipe to preserve later benefit.

2.2.6 Encouraging the success of A-pipe execution

One means of coping with the single-second-chance limitation is to ensure timely updates of corrected state (and corresponding V-bits) from the architectural (B) register

file to the A-file. This optional update path allows dependent instructions to get the correct operands and execute in the A-pipe, as long as their producers have executed in B-pipe by the time they go to dispatch in A-pipe. The DynID tag on each A-file register allows this update, as described in Section 2.2.3. In the initial design [34], every retirement in the B-pipe attempted to update the A-file. Since these updates may contend with retiring instructions in the A-pipe for A-file write ports, the update queue, shown in Figure 2.3, buffers updates waiting on an A-file port. Since whenever a decision is made to defer the execution of an instruction to the B-pipe, it will not write the A-file, the bandwidth required is not expected to be much higher than in a traditional EPIC design. As the V-bits of these deferred instructions' destination registers in the A-file are cleared at dispatch time, the V-bits on these registers will defer all consumers until an update arrives from the B-pipe. The obvious limitation of this update mechanism is that they are only applicable in cases where no intervening instruction in A-pipe has had the same destination. As A-pipe is continually performing preexecution, these updates may often be disallowed through the replacement of the DynID by a later writer.

Without running updates (and because of their limitations, potentially even with them), the accumulation of invalid state in the A-file state might theoretically cause ever-increasing deferral to the B-pipe, although A-file state is periodically completely updated in the restoration the follows detection of branch mispredictions in the B-file. Chapter 4 describes the performance consequences of the running B→A feedback path, showing that while this feedback is useful for minimizing the number of instructions deferred to

the B-pipe, for the majority of benchmarks, the benefits of two-pass pipelining can be achieved without needing to rely on the running update model.

2.2.7 Critical instructions in the two-pass model

Even if updates from the B-pipe are legal, they are only useful if they arrive before consumers execute in the A-pipe, assuming a large amount of slack in the schedule. When the dependence is tighter, as in the example of Figure 2.6, however, this mechanism alone is insufficient. Once an instruction is deferred to the B-pipe, its data flow-dependent instructions will generally be deferred as well, with any unscheduled latency incurred being fully exposed in an in-order manner. There are situations where it is better to stall the A-pipe to allow an instruction to receive its source operand rather than deferring the instruction to the B-pipe. This way, the latencies of it consumer instructions can be initiated in parallel from the A-pipe. (For example, stalling C in Figure 2.6 allows parallel service of subsequent latencies F→J and K→M.) While the hardware generally lacks the foreknowledge to determine when a stall should be materialized in the A-pipe to preserve later opportunities in the B-pipe, the compiler can be made to supply this crucial information through the application of "critical" hint bits as described in Chapter 3. As previously implied, critical instructions in the two-pass model stall in A-pipe when their operands are not ready. Results in Chapter 4 will show that this is an effective technique to increase useful preexecution in A-pipe.

2.2.8 Managing branch resolution

Constructing a two-pass pipeline results in two stages where branch mispredictions can be detected: **A-DET** and **B-DET**, shown as the **DET** stages of the A-pipe and the B-pipe in Figure 2.3. A conditional branch has subtly different execution semantics from other instructions in the A-pipe. When the direction of a branch cannot be computed in the execution stage of the A-pipe, the *misprediction detection* of the branch and not the effect of the branch itself is deferred to the B-pipe. This is implicit in the design of the pipelined machine, since the branch prediction has already been incorporated into the instruction stream in the front end. When a branch misprediction is detected "early" in **A-DET**, the B-pipe continues to execute instructions preceding the branch until it "catches up" to the A-pipe by emptying the coupling queue. Any subsequent instructions present in the coupling queue, if any, must be invalidated, but otherwise fetch can be redirected and the A-pipe restarted as if the B-pipe were not involved. This would result in a reduction in observable branch misprediction penalties.

When mispredicted branches depend on deferred instructions for determination of either direction or target, however, the misprediction cannot be detected until the **B-DET** stage. In this case, the A-file may have been polluted with the results of wrong-path instructions beyond the misprediction. Consequently, all subsequent instructions in both the A-pipe and the B-pipe must be flushed, all corrupted state in the A-file must be repaired from the B-file, and fetch must be redirected. The "speculative" flags in the A-file reduce the number of registers requiring repair: only the A-file entries marked as

speculative need to be repaired from B-file data. As this procedure somewhat lengthens the branch misprediction recovery path for these instructions, performance may be degraded if too many misprediction resolutions are delayed to the B-pipe. Alternatively, one could employ a checkpoint repair scheme to enable faster branch prediction recovery at a higher register file implementation cost [41]. Other invalidation or A-file double-buffering strategies could also be applied.

Issue Regrouping

Because of the (hopefully substantial) advance execution performed by A-pipe, often while an in-order processor would otherwise be stalled by cache misses, much of the B-pipe's execution consists of merely merging precomputed instruction results into the processor state. Because the B-pipe does not recompute the results of such instructions, they can be considered to no longer be dependent on the original producers of their source operands. This elimination of input dependences permits an optimization called *issue regrouping*. Rather than executing strictly according to the compiler's plan of execution–executing in each cycle the issue group the compiler explicitly prearranged–new issue groups can be formed by exploiting preexecution but without changing the compiler-specified instruction *order*. This work presents two approaches to issue regrouping, simple and full issue regrouping.

Figure 2.7 shows demonstrates simple regrouping occurring within a two-pass pipelined system. In Figure 2.7, two iterations of a three-cycle loop are shown. In this

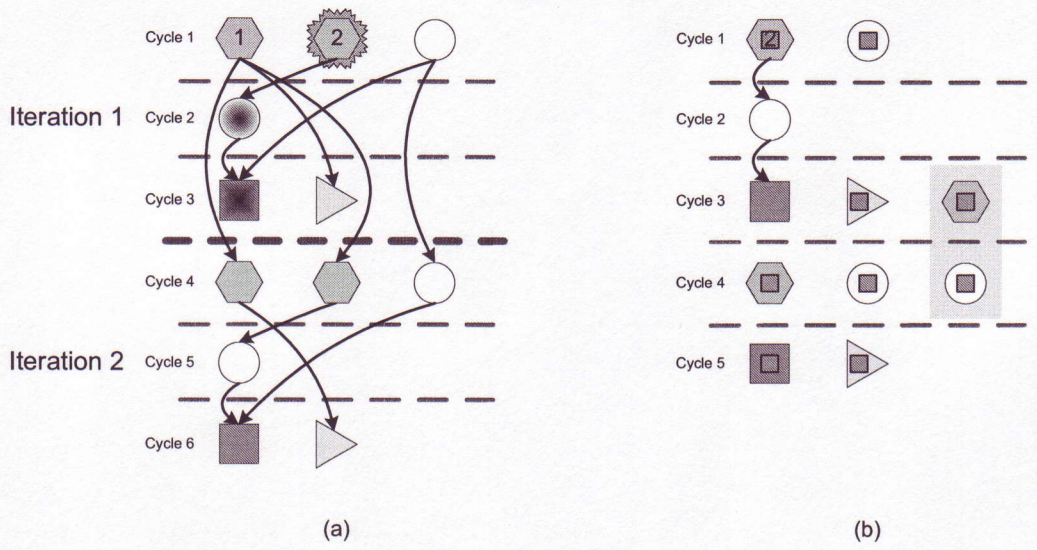


Figure 2.7 Simple regrouping. (a) A-pipe execution according to compiler's plan of execution; and (b) B-pipe issue regrouping.

example and future examples in this section and in Chapter 3, loads are represented as hexagons, stores as squares, arithmetic instructions as circles and branches as triangles. Data dependence arcs between the instructions are also shown.

Figure 2.7(a) shows the dynamic schedule, or the dynamic order of instruction execution, in the A-pipe. A-pipe's dynamic schedule occurs exactly according to the compiler's statically arranged schedule. In Figure 2.7, the load instruction marked 2 in the first iteration misses in the cache. Since A-pipe executes without interlock, execution proceeds past 2's two dependent instructions in the next two cycles. Because load instruction 1 hit in the cache, all instructions in the next iteration are ready in time for their execution in A-pipe.

Figure 2.7(b) shows the execution schedule of the same instructions in B-pipe. Instructions which have successfully preexecuted are shown in the figure with small squares

within the instruction symbol. Because of the dependences from load instruction 2, whose cache miss has now been handled, the instructions in the first iteration execute according to their original schedule. However, through the simple regrouping technique, the two highlighted instructions execute in the cycle previous to the one in which they were originally scheduled. In simple regrouping, preexecuted instructions can be merged into immediately preceding issue groups. Such mergers never generate intra-issue-group dependences because, as mentioned above, preexecuted instructions have no input dependences. By restricting the compaction of issue groups to the movement of preexecuted instruction, the complexity of issue regrouping is minimized. No additional dependence check is required for these instructions because they are guaranteed to be independent of the previous instructions in the issue group.

Simple issue regrouping captures much of the potential of the issue regrouping strategy. During B-pipe reprocessing of instructions, dependences between the instructions deferred in A-pipe tends to be rather dense, limiting the additional opportunity for general compaction of the dynamic schedule by moving these unexecuted instructions into earlier issue groups. However, there are cases where there is room for such general regrouping-based compaction. Figure 2.8 shows a single iteration of the loop from the example in Figure 2.7. In Figure 2.8(a), load instruction `1` misses in the cache during its A-pipe execution, two-cycles later deferring the branch instruction dependent on `1`. During B-pipe processing, shown in Figure 2.8(b), since the chain of instructions fed by load instruction 2 has been preexecuted, the store instruction in the example can be

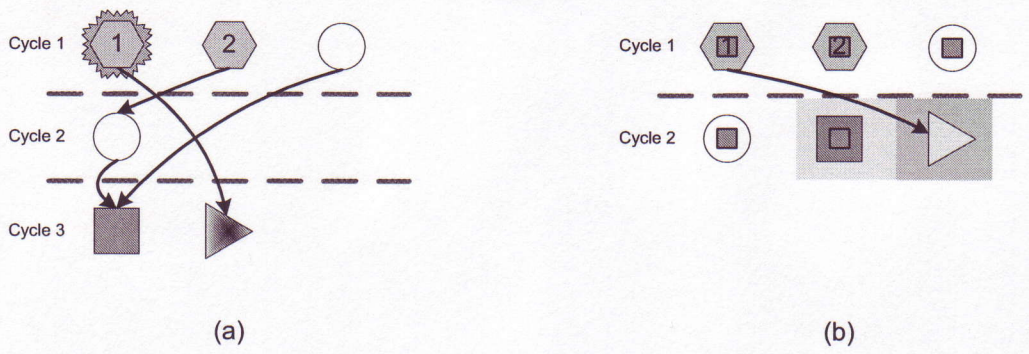


Figure 2.8 Full regrouping. (a) A-pipe execution according to compiler's plan of execution; and (b) B-pipe issue regrouping.

combined into the issue group for the second cycle using simple regrouping (shown with a light shading). In addition, even though the branch instruction was not preexecuted, it can be combined with the issue group for the second cycle if dependence checking is performed under full issue regrouping (shown with a dark shading). It can be combined with the previous issue group because it is independent of other instructions executing in the second cycle. Full regrouping is done by checking dependences on an instruction-by-instruction basis in the **REG** stage (as would be done in a non-EPIC in-order processor such as SPARC [38]), thus still respecting the compiler's ordering of instructions.

Chapter 4 demonstrates that the more complex full regrouping outperforms the simple regrouping technique, occasionally significantly. As the in-order superscalar approach to issue (i.e., relying on the compiler's ordering of instructions, but not limiting issue to the compiler's explicit cycle assignment) could be used in the baseline model, this technique was also experimentally examined. Results in Chapter 4 show that without the multiple-pass pipelining preexecution, there is little room for issue regrouping.

2.3 The Multipass Pipeline Microarchitecture

The two-pass pipelining approach presented in Section 2.2 serves as a helpful introduction to the concepts of the more general multipass pipelining approach. Multipass pipelining can be conceptually considered as a two (or more)-pass pipelining scheme which all passes implemented virtually on a single physical pipeline. Alternatively, multipass pipelining could be considered to be a significant extension to in-order runahead [19, 20]– an extension which incorporates the "flea-flicker" ideas first presented with two-pass pipelining [34].

To lay a foundation for outlining the aspects of the multipass model which have already been presented and introduce an extension designed to capture the concurrent preexecution and execution present in the two-pass model, a small example in the same style as Figure 1.2 is helpful in illustrating the ability to overlap effect of cache-misses in three different architectural models.

2.3.1 Developing the multipass pipelining model

Figure 2.9 shows execution time-lines for three different execution models, each case suffering the same pattern of four cache misses. As in Figure 1.2, activity in the execution stage of the pipeline is shown in the **EXE** component of the time-line, while the handling of cache misses is shown in the **MEM** component line. The example presented is an extension in time of the example in Figure 1.2, with the example extended to include

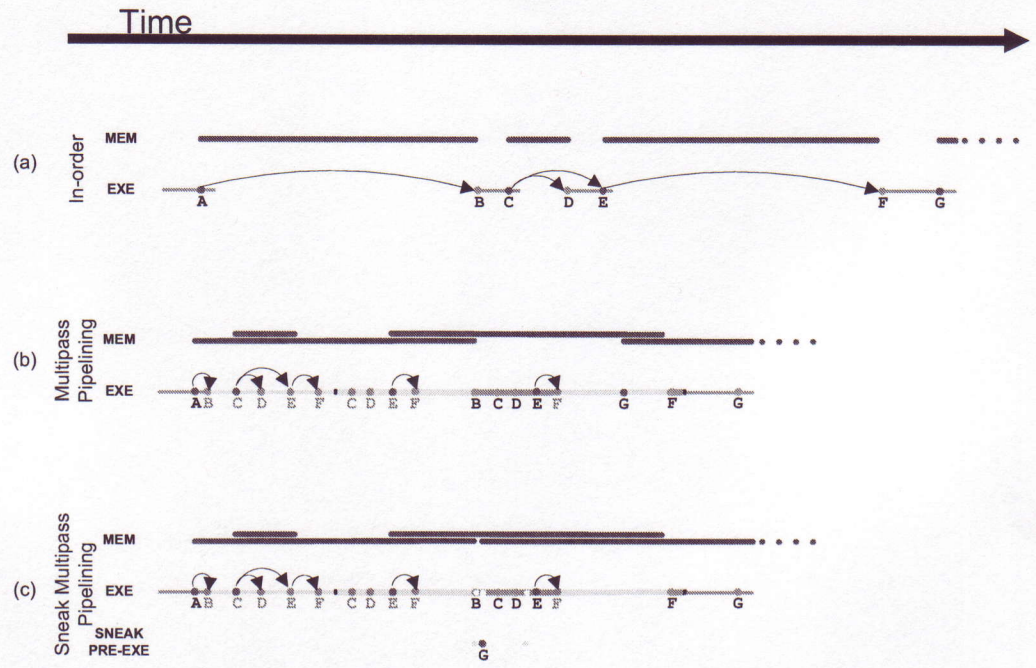


Figure 2.9 Execution time-line for multipass models of execution.

data-cache missing load instruction `G`. Note that `G` is a miss to distant level of cache. The handling of `G` takes long enough that its handling extends beyond the shown time-line.

Revisiting the in-order model

The first model, in-order, shown in Figure 2.9(a) matches closely the example in Figure 1.2. As in the previously detailed example, because of the cache-miss dependencies, `A`→`B`, `C`→`D`, and `E`→`F`, the bulk of the time in the example is spent stalled on cache misses. No stall for `G`'s miss is evident in Figure 2.9, because the example ends before any consumers of `G` are reached.

Introducing the multipass model

In the second model, base multipass pipelining, shown in Figure 2.9(b), instructions execute in order until one attempts to consume the result of an outstanding cache miss (such as `B`). While such an instruction would normally suffer an in-order stall, that instruction is simply deferred. Advance preexecution is allowed to occur while the cache miss is being serviced. Once the cache miss starting the preexecution is complete, this advance execution is terminated, and normal execution returns to the deferred consumer. Note that execution resumes with the consumer rather than the instruction immediately following the missing load (as in the simple run-ahead preexecution model presented in [19]). Since preexecution execution in the multipass model[3] occurs only on *consumption* of a missing value and not on the miss itself, a consumer scheduled at miss latency by the compiler will not be penalized unnecessarily, as opposed to the original run-ahead preexecution model.

There are three key concepts that are evident from the example in Figure 2.9(b):

1. Cache-miss resilience of multipass pipelining, even for dependent chains of misses (demonstrated in Figure 2.9(b) by the preexecution following the execution of the previously deferred load instruction `E`).

2. Acceleration of execution (and additional preexecution) through the reuse of pre-executed results (shown in Figure 2.9(b) as the widening of the **EXE** line).

[3]Similarly, preexecution in the two-pass approach defers only unready consumers, speculatively moving on to subsequent instruction after such a consumer.

3. Restart of advance preexecution after critical instruction deferral (shown in Figure 2.9(b) with immediate return to the first deferred consumer, instruction `B`, at the dark point of the preexecution segment of the **EXE** line.

The mechanism behind each of these concepts will be detailed in the subsequent sections.

The sneak multipass extension

Figure 2.9(c) shows an extension, *sneak multipass*, to the base multipass pipelining design that seeks to capture some degree of the concurrent execution and preexecution that was inherent in the two-pass model. This model utilizes the same basic design as the basic multipass approach, but uses an interesting optimization to exploit some of the potential for concurrent processing of these two streams using a technique analogous to a very limited form of simultaneous multithreading [42].

In Figure 2.9(c), when cache-missing load `A` is complete, execution returns to `A`'s consumer `B`. However, in the sneak multipass model, preexecution is also allowed to continue to occur from the last preexecuted instruction. preexecuted instructions *sneak* through available issue and functional resources (available because the execution stream of instructions does not fully utilize them). In the example, this concurrent preexecution is shown with the **SNEAK PRE-EXE** line. In this example, by sneaking preexecution during the execution segment starting with `B`, load instruction `G` is reached much earlier than in the base multipass approach.

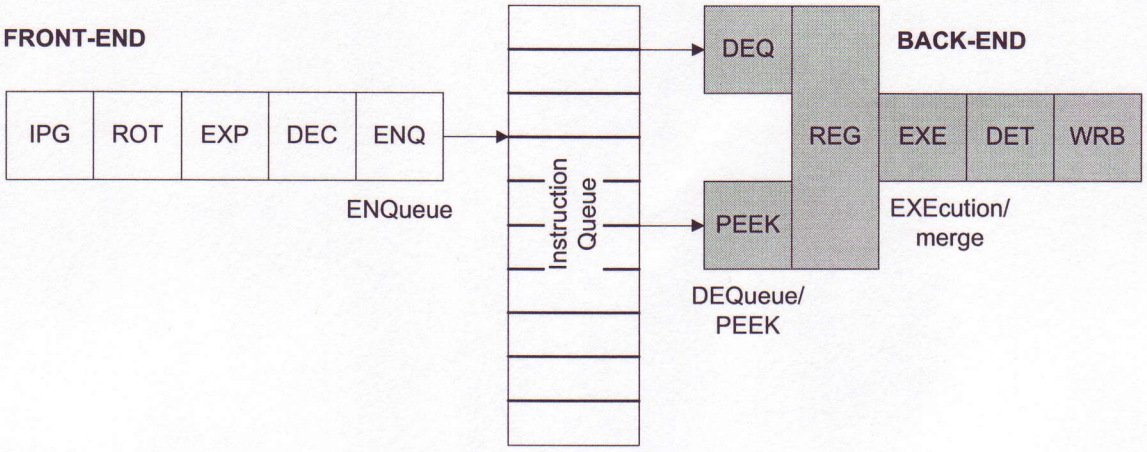


Figure 2.10 Base integer multipass pipeline.

2.3.2 Base multipass pipeline organization

Figure 2.10 shows the base multipass pipeline organization, adapted from the contemporary in-order pipeline design described in Section 2.1. The additions to the pipeline shown in Figure 2.2 occur between the four front-end stages and the four back-end stages of the base pipeline. First, the multipass pipeline extends the 24-instruction buffer in size and functionality. Rather than storing freshly fetched instructions, the multipass instruction buffer instead buffers decoded instructions. This allows the instructions to be buffered at a stage closer to the back-end pipeline. New stages are added to **EN**queue and **DEQ**ueue (or **PEEK**-at) instructions in the buffer. A third additional stage is added to perform the instruction regrouping described in Section 2.2.8.

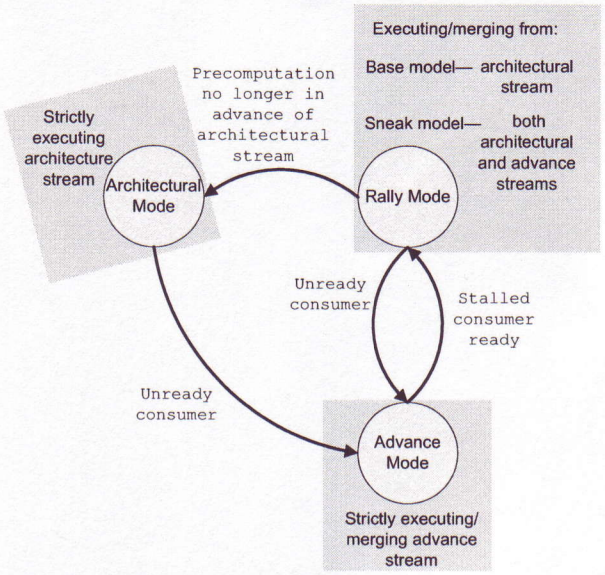


Figure 2.11 Three modes of multipass operation.

2.3.3 Modes of multipass operation

In a single pipeline, multipass pipelining performs the same kind of persistent, advance preexecution that was described for the two-pass pipelining approach in Section 2.2. Because both execution and preexecution occurs on the same physical pipeline, at different time, this pipeline operates in different modes. Figure 2.11 shows the three modes of operation of the multipass pipeline.

Architectural execution

Initially, the pipeline enters the *architectural* mode when the execution of a program starts. In the absence of run-time stalls, instructions are released from the instruction

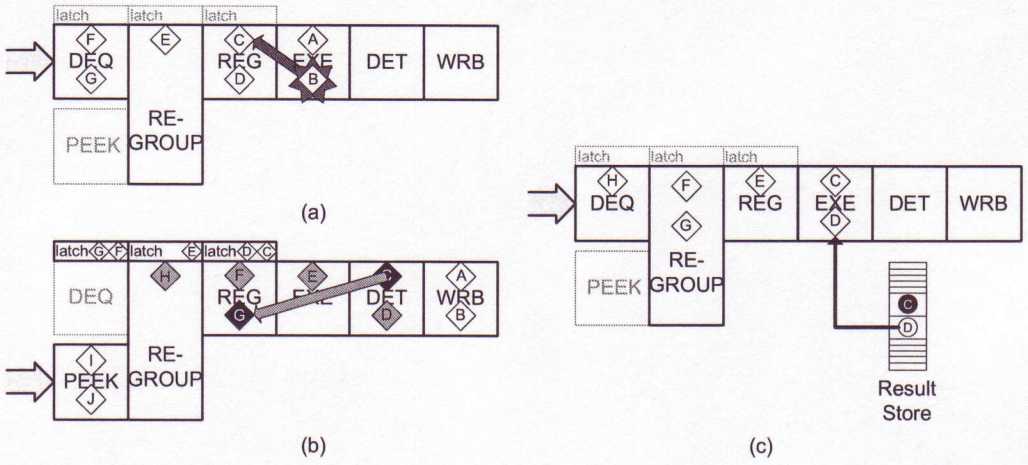


Figure 2.12 Multipass pipelining operation: (a) cache miss interrupts in-order execution, (b) peeking ahead in the instruction queue, and (c) return to *facilitated* in-order execution.

buffer using the **DEQ** pointer. The release and execution of these instructions very much resembles that of conventional in-order execution pipelines.

Advance execution

As in the general multiple-pass pipelining example in Figure 2.1, advance multipass preexecution begins with the failure of an instruction to receive a valid operand. For example, in Figure 2.12(a), the load `B` misses in cache, causing dependence checking logic in the **REG** stage to detect an unready operand for instruction `C`. At this point, the pipeline enters the *advance* mode.

In order that in-order execution can be resumed without delay when `C`'s data returns, instructions `C` and `D` at the **REG** stage; `E` at **REGROUP**; and `F` and `G` at **DEQ** are all preserved in a set of latches at these stages. These instructions are also allowed to

proceed through the pipeline as advance instructions. Additionally, the current value of the **DEQ** pointer is preserved for future use.

Another pointer, **PEEK**, is initialized with the current value of **DEQ**. **PEEK** is used to release instructions from the instruction queue during advance mode. Instructions that follow `G` will now be released from the instruction buffer as advance instructions. Dequeued instructions, `A` through `G`, and their peeked successors released from the instruction buffer form an advance stream. Any instructions failing to receive valid input operands are simply suppressed from execution. In a way similar to A-pipe deferral in the two-pass model, an invalid (I) tag is attached to their output value(s) to indicate that these instructions were deferred. This allows consumers of the suppressed instructions' result to be suppressed in turn. As in the A-pipe, the multipass pipeline in advance mode selectively executes only the advance stream instructions that can be preexecuted with ready data.

Advance stream instructions are not allowed to write their results into the architectural register file (ARF). Instead, their results are redirected to the speculative register file (SRF). The ARF is analogous to the B-file, storing the architecturally committed results. The SRF is analogous to the A-file from the two-pass model, except that the SRF will be used to store the speculative state from each pass of advance preexecution. When the pipeline enters advance mode, SRF does not contain any valid information. Advance stream instructions thus initially access ARF for their input operands. As advance stream instructions write into SRF, their consumers need to be redirected to this

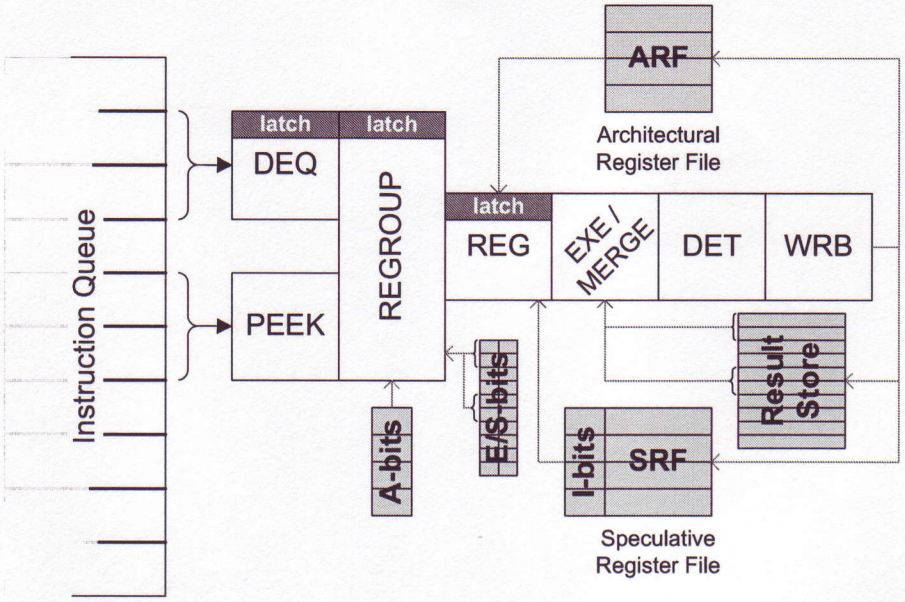


Figure 2.13 Execution subpipeline datapath.

file for input operands. This redirection is realized with a bit vector, shown as **A** in the detailed back-end multipass pipeline in Figure 2.13. Each "advance" bit (A-bit) indicates that future accesses to its associated register entry should be redirected to SRF. During normal execution, the A-bits are clear, and all instructions read operands from the ARF. When the processor enters the advance mode, each advance stream instruction sets the A-bits of its destination registers to 1, enabling subsequent consumers to read that value from SRF. Note that since the A-bit vector is read in the **REGROUP** stage, advance stream instructions may write their result to the SRF (and thus the A-bit vector) after a dependent instruction has read the A-bit's of its source operands. The logic for bypassing between in-flight advance stream instructions is detailed in Section 2.3.4.

As implied earlier this section, each SRF entry contains an I-bit to mark invalid values written by advance stream instructions. If the execution of an advance stream instruction is suppressed, the I-bit in its destination register is set. Future instructions reading this register will thus be suppressed.

An important feature of the advance mode is that results of the correctly executed advance stream instructions are preserved in a result store (RS), an analogous component to the CRS in the two-pass approach. The RS is written in addition to the advance file by preexecuting instructions during advance mode. There is a one-to-one correspondence between instruction buffer and RS entries. The RS entries corresponding to suppressed advance stream instructions are simply marked as empty. In Figure 2.13, a bit vector with entries corresponding to the RS is read in the **REG** stage. The E-bit denotes that the corresponding RS entry is empty.

Note that, when advance execution begins, three pipeline stages worth of instructions have already been dequeued from the instruction queue. If the entries occupied by these instructions have been recycled to new instructions, these instructions have no corresponding instruction queue entries (and thus no corresponding RS entries) when advance mode begins. In this work, the instruction queue entry for a dequeued instruction is not reused until that instruction has been issued from the **REG** stage in normal mode. An alternate approach would have been to disassociate instructions dequeued during normal mode from their instruction queued entries, thus preventing such instructions

from writing their results to any RS entry, slightly reducing the amount of reuse of precomputed advance results.

Rally mode

When the load miss that caused the original transition to advance mode is satisfied in the **REG** stage, the pipeline switches to *rally mode* wherein architecture-stream instructions resume execution. This is triggered by a successful bypass to the waiting architecture-stream instruction latched at the **REG** stage. Upon entering rally mode, the pipeline stops executing the advance stream.[4] The latched architecture-stream instructions at the **DEQ**, **REGROUP**, and **REG** stages are now unlatched and allowed to proceed as if in normal mode. Furthermore, the preserved **DEQ** pointer now returns to its use releasing succeeding architecture-stream instructions from the instruction buffer.

Architecture-stream instructions that have already been correctly preexecuted in the advance mode are indicated by an entry marked with a clear E-bit in the result store. They do not need to be reexecuted; rather, the **EXE** stage simply merges these results into the architectural file. Just as with the reuse of precomputed results in the two-pass model, this is beneficial for three reasons. First, the pipeline does not have to spend the energy to execute an instruction whose results are available from prior advance-mode execution. Second, long-latency instructions, such as multiply instructions, are effectively converted into single cycle instructions with this feature, further reducing potential stalls

[4]In the "sneak" model, execution of advance-stream instructions will continue at this point.

in rally mode. Third, this reuse enables instruction regrouping, just as in the two-pass approach. Specifics of multipass instruction regrouping are described in Section 2.3.8.

If any architecture-stream instruction receives an unready operand bypass value at the **REG** stage, the pipeline switches to advance mode again. The architectural stream of instructions at the stages between the instruction buffer and the **EXE** stage again need to be preserved in the latches as in the architecture mode.

Alternatively, if the **DEQ** pointer reaches the farthest point of the preserved **PEEK** pointer while in the rally mode, then the architecture stream has caught up with the advance stream. This indicates that there are no longer any instructions deferred on pending cache misses. The pipeline can now switch back to architectural mode. Note that in the base multipass model, execution in rally mode and architectural mode is, in reality, treated identically. The original PEEK pointer serves only to denote the furthest point of the most-recent advance execution.

Return to in-order execution

Peeking of instructions begins when an dispatching instruction would consume the result of a load that is still being handled in the data cache. At that point, in addition to being dispatched, these instructions (which had been dequeued from the instruction buffer) are specially latched. When the cache miss completes, these latched instructions can now be processed as part of the architectural in-order execution. These instructions are unlatched, preempting peeked instructions in the same stages. Peeking halts and

dequeuing begins again. All A-bit file entries are set (only the architectural register file values are meaningful). Bypassing from any in-flight peeked instructions is suppressed.

In Figure 2.12(c), architectural execution has resumed with the in-order dequeuing of instructions. Instructions that were correctly preexecuted simply read their result from the RS rather than reexecuting. If this preexecution consisted of cache-misses (or other long-latency operations) as in the example in 2.9, the architectural execution is potentially accelerated.

2.3.4 Maintaining correct execution of independent streams

The multipass pipelining model must accommodate in-order and advance streams in a single pipeline without comingling their values in an undesired fashion. This involves preventing spurious bypasses and respecting certain output dependences.

Spurious bypasses between advance and in-order instructions are easily prevented through the addition of an "architectural" bit (A) to each register identifier in the bypass network, to indicate whether an in-order or an advance instruction generated the value being bypassed. Instructions dequeued in architectural mode write a 1 to the A-bit of their destinations; instructions peeked during advance mode write a 0 denoting that advance preexecution has overwritten the value stored in the ARF for that register. Advance instructions accept the bypass of the most recently executed instruction; architectural instructions are insensitive to bypasses marked with the A-bit. Allowing the

bypass network to discriminate among these values, rather than performing a renaming-style labeling of source operands in an early pipeline stage, allows the straightforward implementation of predication characteristic of in-order designs rather than the more complicated schemes necessary for out-of-order pipelines [18]. This is an advantage of the multiple-pass pipeline designs—they preserve the simplicity of in-order models including ease-of-implementation of key features like predication while providing much of the cache-miss tolerance of the dynamic models.

2.3.5 Handling WAW dependences in the multipass approach

In an EPIC implementation like the Itanium 2, all instructions are issued strictly in order, but variable-latency instructions might complete out of order. Since EPIC processors do not dynamically rename register operands, a shorter-latency writer might follow a longer-latency writer of the same operand. Out-of-order instruction completions cannot be allowed to cause inconsistent register state. Thus, variable-cycle latency instructions (in particular loads) are scoreboarded to force output dependent instructions to stall.

Similarly, the architectural stream of execution stalls when write-after-write dependencies present themselves. However, in the execution of advance instructions, an alternate approach is preferred. Dynamic write-after-write dependencies are reached frequently in loops, as dynamic instances of the same static instruction are obviously output dependent. Additionally, when a write-after-write is reached in advance execution, all

consumers of the first write have already been processed (and deferred) so there is no reason to stall on these writes. A simple alternative approach (and the approach proposed here) is suppress only the register file write back of loads once a write-after-write occurs. This mechanism is implemented within the memory pipeline; when load accesses return from cache misses, they are merged into the memory pipeline to allow their write back in the **WRB** stage. Before merging an advance stream load into the memory pipeline, a check is first made to determine if a WAW had occurred, and if so the write back is simply squashed.

2.3.6 Efficiently maintaining a correct memory interface

A multipass-pipelined system maintains an underlying in-order execution model. Advance-stream instructions, since they are processed out of program order from the architecture stream, are thus speculative, and their processing does not directly affect execution state. The purely speculative processing of nonmemory instructions are handled simply with the addition of the alternate, speculative register file. As in two-pass pipelining, memory instructions must be considered more carefully.

Advance stream stores obviously cannot commit to architectural memory state—instead, like in the two-pass approach, they are only recorded in the store buffer (a common microarchitectural element) and are never written out to memory. The buffered memory writes from peeked instructions are discarded once execution returns to the in-order dequeuing. As long as peeked store values can be forwarded through this store

buffer to subsequent peeked loads, a consistent memory interface is maintained. However, stalling the peeking in the instruction queue due to the limited capacity of this buffer can easily hamper the achieved advance-execution instruction window. Additionally, peeked store instructions may be deferred due to an unavailable or invalid target address, casting the results of subsequent memory reads in doubt.

To guarantee that an inconsistent view of memory does not incorrectly affect execution, some bookkeeping beyond what is ordinarily required to implement consistency semantics is needed. Multiple straightforward approaches present themselves; the modified advanced load alias table could be used as in Section 2.2.4, with the ALAT reset every time advance mode is entered. A more general approach like that used in many out-of-order processors [2] where a content-addressable memory to detect when the processing of a load dynamically reordered with a conflicting store. However, these kinds of approaches add additional hardware complexity and are still limited in capacity. Exploiting the fact that peeked instructions will be processed again after they are dequeued in-order, multipass pipelining takes a value-based approach [43].

Figure 2.14 shows the simple approach used in the multipass pipelining model. In Figure 2.14(a), store instruction 1 is deferred. Because the address stored by 1 is unknown, such a deferral means that all future load instructions (and their dependents) are data speculative. Advance mode, data-speculative instructions mark their RS entries with the S-bit in Figure 2.13. In the example in Figure 2.14(a), a subsequent load instruction, 2 reads the value of variable A in advance mode. When these instructions are processed

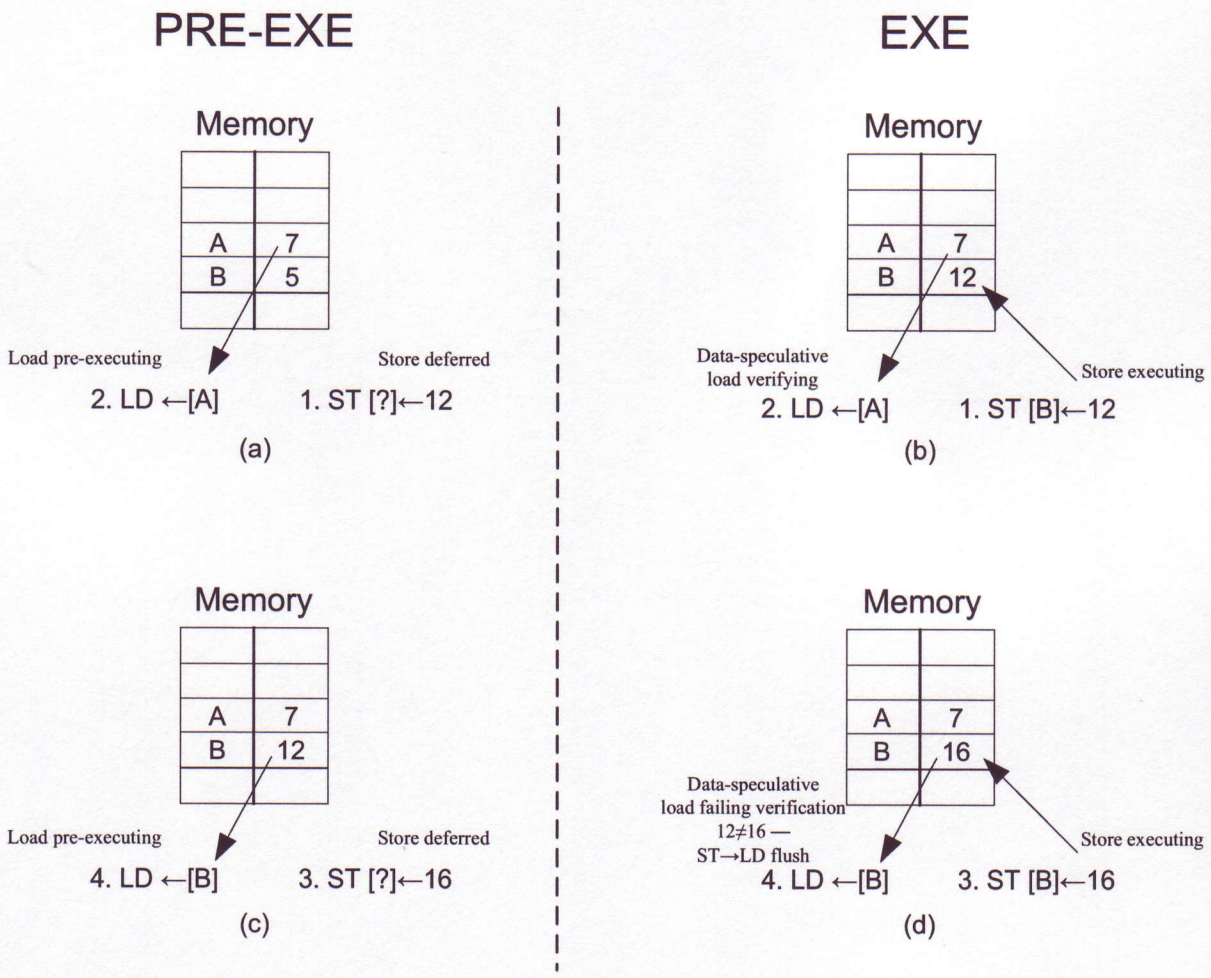


Figure 2.14 Multipass pipelining value-based memory ordering approach.

in rally mode, the previously deferred store instruction 1 executes, stores the value of variable B. Because 2's result is marked data speculative in the RS, it will reperform its memory access (most likely hitting in the first-level cache as this address was previously loaded in advance mode) and will verify that the value loaded is the same as the value loaded in advance mode. In Figure 2.14(b) because no intervening store has overwritten the value loaded by 2, its value is confirmed.

In Figure 2.14(c), a store instruction 3 is deferred because of an unknown address and a load instruction 4 data speculatively reads the value of variable B in advance mode. In the architectural mode, shown in Figure 2.14(d), the deferred store, 3, executes, overwriting the variable B in memory. When 4 attempts to confirm its value, it discovers that it loaded the wrong value in advance mode, and a pipeline flush is performed.

2.3.7 Increasing the throughput during multipass execution

This subsection will detail three specific mechanisms that improve the effectiveness of multipass execution. The first, instruction regrouping, accelerates execution as it did in the two-pass model. Second, critical operations are used in a unique way in the multipass model to improve performance. Finally, the sneak model enables a degree of the concurrent preexecution and execution, a desirable property inherent to the two-pass model.

2.3.8 Instruction regrouping in multipass pipelining

Instruction regrouping was described as an optimization in two-pass pipelining in Section 2.2.8. Simple or full regrouping can similarly be used within multipass pipelining to accelerate rally mode (and sometimes advance mode) execution. In advance or rally mode, instructions with a clear E-bit in their RS entry merge their result rather than reexecuting. Just as in two-pass pipelining, instruction regrouping can exploit this result reuse to accelerate execution.

As described in Section 2.3.6, dynamically data-speculative loads do re-access memory to confirm their results in the multipass model. It is possible that an incorrectly speculatively executed load may be grouped with a (incorrectly) preexecuted consumer. However, the ensuing pipeline flush maintains correct execution state.

2.3.9 Critical instruction restart

In the two-pass model, critical results of instructions are used to control preexecution in the A-pipe. Because it is desirable to prevent preexecution from becoming replete with invalid operands, advance preexecution is prevented from proceeding when the result of a critical instruction is not ready.

The multipass approach does not suffer from the single-second-chance problem of two-pass pipelining, and therefore it is not desirable to stall preexecution. However, when the result of a critical instruction is not ready, if preexecution continues, it is unlikely to produce much fruitful computation or memory access. Rather than continuing

into a great deal of deferred execution, *advance execution restart* occurs. As seen from the examples in Figure 1.2 and Figure 2.9, by restarting advance mode execution with the instructions latched when advance mode began can provide for additional useful preexecution. Instructions that were previously deferred might be ready for execution because the miss causing their deferral has completed. Thus, in multipass pipelining, this restart is performed whenever an unready critical instruction is reached in advance mode. The A-bit vector is cleared, latched instructions in the **DEQ**, **REGROUP**, and **REG** stages are unlatched, but preserved for future restart. Advance mode execution is thereby restarted.

The sneak multipass model: Enabling two concurrent streams

When the cache miss that inaugurated advance execution completes, rally mode commences, and, in the base model, advance execution ends. As described in Section 2.3.3, the old location of the **PEEK** pointer serves only to denote the farthest reach of the most resent advance execution. However, in the sneak model, advance execution continues concurrently with architectural execution during rally mode.

Because of limited available ILP, the compiler often cannot successfully schedule the maximum issue width of instruction every cycle. In the sneak model, instruction peeking continues in rally mode, and peeked instructions *sneak* ahead using issue and functional unit resources not consumed by architectural execution. The peeked instructions pulled from the instruction queue are grouped with the ongoing architectural execution in the

REGROUP stage according to available resources. No dependences with architectural execution need to be observed; in the sneak model the values in the speculative register file are maintained and thus the only architectural register values that the peeked instructions will read were computed before the advance execution began. This mechanism enables the simultaneous execution of instructions from the advance and architecture streams in a way reminiscent of simultaneous multithreading [42] or helper threads [44].

As in-order processors already must dispatch instruction according to limited functional resources, sneaking requires little additional complexity and is a considerably less complex operation than out-of-order execution's dynamic scheduling of instructions–reordering the execution of instructions based on both operand and functional unit availability.

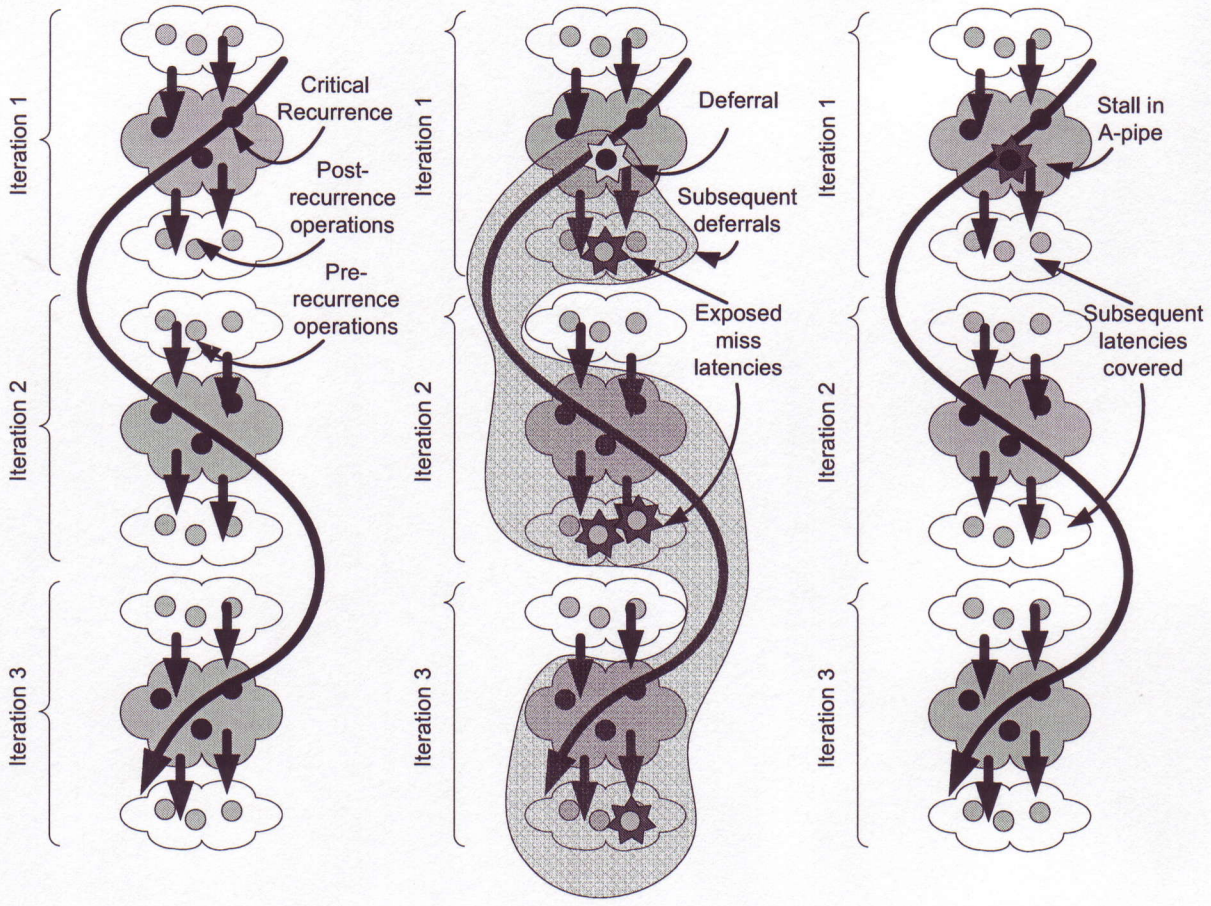
Since resource priority is given to the architectural execution, eventually the **DEQ** pointer may catch up with the **PEEK** pointer, returning the processor to architectural mode. Additionally, the architectural execution may encounter another miss-consuming instruction, setting the **PEEK** pointer to the **DEQ** pointer and beginning fresh advance execution.

3 COMPILER-BASED CRITICAL INSTRUCTION IDENTIFICATION

Critical instructions are instructions whose pre-execution deferral will lead to the vast majority of subsequent instructions also becoming deferred. The same notion of critical instructions are used in both two-pass and multipass pipelining, but as described in Chapter 2, they are used in somewhat different mechanisms reflecting the different characteristics of the two designs. The insight behind the compiler strategy for critical operation identification will first be presented within the context of the two-pass pipelining model, followed by the actual algorithm used for each of the multiple-pass pipelining approaches.

3.1 Critical Recurrences

Figure 3.1 shows an example of a situation in which the compiler can distinguish between operations that should be deferrable and those "critical instructions" that, within two-pass pipelining, should always complete execution in A-pipe, in the interest of preserving future deferral opportunities. Figure 3.1(a) shows three iterations of a stylized unrolled loop dependence graph, in which arrows indicate dependences among the clouds of instructions. The arrow running through all iterations indicates a recurrence through



(a) Unrolled loop dependence graph showing a recurrence

(b) Poisoning effect of deferral on recurrence

(c) Benefit of preventing deferral

Figure 3.1 Overlap of latency events preceding and succeeding critical recurrences.

the loop, on which at least some parts of all subsequent iterations are dependent. Given the identification of operations participating in this recurrence, other loop operations can be classified as "prerecurrence" (preceding the recurrence in the dependence graph), or "postrecurrence" (dependent on recurrence operations). This classification is significant to the operation of two-pass, as shown in Figure 3.1(b). (It also is important in the comparison of two-pass to out-of-order execution, but this will be discussed later.) Here, the marked operation in the recurrence part of the first iteration is deferred to B-pipe. This constrains all data-flow-dependent successors—all subsequent recurrence and postrecurrence operations—to be deferred as well. Any unscheduled latencies among these operations will be materialized in the B-pipe, degrading performance. The cost of these stalls may far outweigh the benefit of deferring the original recurrence-bound operation. Figure 3.1(c) shows the result of preventing the deferral of operations participating in the recurrence. Here, a stall is materialized in A-pipe, slowing down the "advance" track, but the opportunity to absorb stalls among subsequent postrecurrence operations is preserved.

Preventing the deferral of all instructions that participate in recurrences can, however, detract from performance potential by delaying the execution of subsequent prerecurrence operations, as apparent in a comparison of Figures 3.1(b) and (c). Since prerecurrence operations are not data-dependent on the recurrence, they are initiated without delay in the scheme of (b). However, in (c) the stall of A-pipe will delay the A-pipe processing of the prerecurrence operations, thus reducing the ability of the two-pass pipeline to tolerate

the latency of these instructions. There is thus a need to balance the desires to initiate subsequent operations early and to initiate a useful proportion of unscheduled-latency operations in the A-pipe.

These considerations lead to a straightforward compiler technique for marking operations for which, in the event of dispatch with unready operands, stalling in the A-pipe is a better strategy than deferring to the B-pipe. An addition to each instruction of a single bit, a hint called an *audible*[1] is assumed. Alternate strategies are considered in Chapter 5. When this bit is set, marking an instruction "critical," and an operand is unready in the dependence check stage, the A-pipe stalls rather than deferring the instruction to the B-pipe. The compiler sets these bits in the following manner: for each procedure, all strongly-connected components (recurrences) in the data dependence graph are identified. Each recurrence is evaluated to determine if its constituent operations should be marked with the "audible" bit, forbidding deferral. To do this, the compiler identifies for each recurrence the sets of pre- and postrecurrence operations, according to the data dependence graph. The total execution weight (derived from a previous profiling run, part of a normal profile-guided compilation path) of operations with latency-masking potential (loads and long-latency operations like multiplications and divisions) is computed for both sets. If the "masking potential" weight of the postrecurrence operations significantly outweighs that of the prerecurrence operations, the recurrence operations are marked critical–not to be deferred. Otherwise, the benefit of starting prerecurrence

[1]In American football an *audible* is a verbal command by the quarterback to change an offensive play on short notice.

operations early is judged to outweigh the cost of possibly deferring all postrecurrence operations, and the recurrence is not specially marked.

3.2 Code Examples

To better discern how multiple-pass pipelining microarchitectures exploit opportunities for cache latency tolerance (particularly by exploiting instructions identified as critical), it is useful to further analyze examples of real program behavior. Three examples will be presented from some of the most important loops in *mcf*, show varying potential for cache miss latency tolerance. In all three examples, execution time is dominated by frequent cache misses. The degree to which the flea-flicker mechanism can cover the cache miss latency in these examples is used to help explain the mechanisms' benefits in Chapter 4. An additional example from *gap*, will illustrate the treatment of reductions as a special type of recurrence and shows how multiple-pass pipelining garners benefits besides cache miss latency. In these examples, as in the examples in Figures 2.7 and 2.8, loads are represented as hexagons, stores as squares, branch instructions as triangles, and other simple arithmetic instructions as circles. Data dependence arcs reflecting cross-iteration dependences are dashed, rather than solid, lines.

3.2.1 Loop recurrences independent of cache misses

The first example in Figure 3.2 shows the data dependence graph of a loop from function `primal_bea_mpp()` in *mcf*, in which each iteration depends simply on a stride

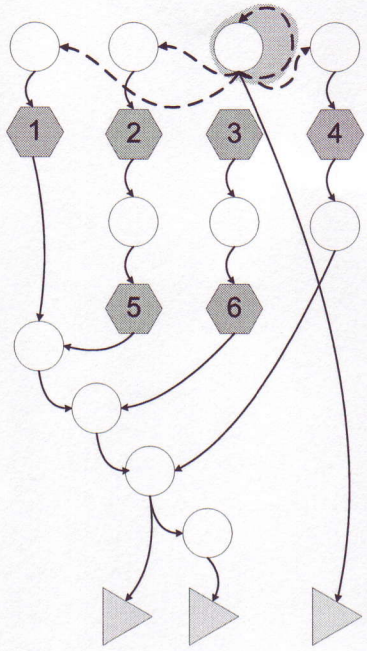


Figure 3.2 Example loop from *mcf*: strided recurrence.

computation, the recurrence highlighted in gray. In this example, loads 1 and 2 miss in the first-level cache on roughly half of their executions while 3, 4, 5, and 6 virtually always miss. Since the stride computation, independent of all the cache misses, always executes in the A-pipe, loads 1, 2, 3, and 4, whose addresses it supplies, also execute in the A-pipe. As these loads incur misses, their dependents defer to the B-pipe; because the loop-driving recurrence is independent, such deferral does not negatively affect future iterations of the loop. Loads 5 and 6, however, as dependents of loads 2 and 3 (which almost always miss), are usually deferred to the B-pipe. Their dependents suffer in-order stalls for lack of a third pipe to which to defer. Flea-flicker multiple-pass pipelining succeeds in overlapping subsequent execution with the miss latencies of loads 1, 2, 3, and

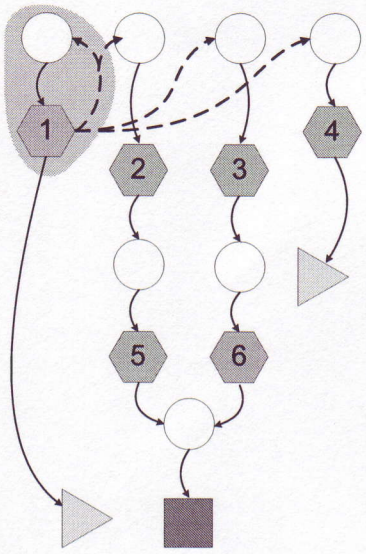


Figure 3.3 Example loop from *mcf*: pointer dereference recurrence.

4, and in so doing hides most of the memory latency exposed in an in-order execution. The two-pass approach, however, fails to hide misses in the dependence shadow of other misses, thus falling short of the multipass or out-of-order approaches.

3.2.2 Loop recurrences containing cache misses

Figure 3.3, from `refresh_potential()`, has six miss-prone loads like the previous example, but here load instruction 1 participates in the loop recurrence (again highlighted in gray). In this linked-list traversal, in which load 1 generates the pointer to the node to be processed in the next iteration, as long as preexecution (either in A-pipe for the two-pass model or during advance mode for the multipass model) is successful, the next iteration's loads 1, 2, 3, and 4 can be preexecuted, allowing misses on loads 2, 3, and 4 to be tolerated. As in the example from Figure 3.2, loads 5 and 6 are frequently deferred

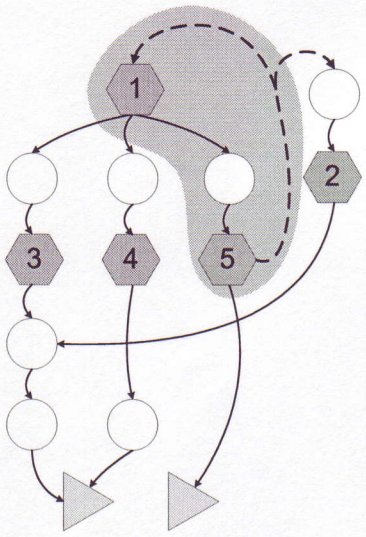


Figure 3.4 Example loop from *mcf*: no misses disjoint from recurrence.

because of the frequent misses in loads 2 and 3; thus, in two-pass pipelining, the B-pipe stall incurred by their 5 and 6's dependents cannot be overlapped with misses from those instructions in future iterations. Since, in this example, load 1 participates in the recurrence, if its miss were to result in a deferral, all subsequent loads for the duration of the loop would be deferred as well. This recurrence is therefore marked by the compiler as critical to maintain a steady-state potential for benefit in two-pass pipelining, and allow additional benefit through advance execution restart in multipass pipelining.

3.2.3 Loop with all cache misses entangled with the recurrence

Figure 3.4 shows a loop from the function `price_out_impl()` containing five loads, all usually satisfied from the L3 cache. If, during preexecution, a cache miss occurs in load 2, 3, or 4 (not part of the recurrence), the miss's dependents can be deferred,

enabling preexecution of the next iteration. In such a case, the multiple-pass pipelining mechanism would allow the overlap of the initial misses with cache misses in the next iteration. In reality, however, this loop does not yield performance benefit on any of the two-pass, multipass or out-of-order microarchitectures. Almost every dynamic load in this example is a cache miss—the latency tolerance mechanisms can do nothing to accelerate initiations of this loop beyond the sequential stalls caused by misses in load instructions `1` and `5`. Additionally, the loads in this example access related memory locations. Different fields of only two different structures are accessed by these loads, and because of this relationship, load `1` and load `2` both access the one cache line while loads `3`, `4`, and `5` all access another. Since there are no misses disjoint from the misses that occur in the recurrence, no overlap of cache misses is possible from one iteration of the loop to the next. Time spent in data cache stalls is thus not a universal indicator of potential for latency tolerance through even dynamic instruction scheduling; the dependence graph and relative data layout often limit benefit.

3.2.4 A reduction example

Figure 3.5 shows yet another data dependence graph, this one from *gap*'s `ProdInt()`. In this example, loads almost always hit in cache. The execution time of this loop is dominated by the long-latency multiply instructions shown as dark circles, and the loop recurrence is again indicated by the shaded portion. Shaded squares and wide arrows indicate stores and (spurious) memory flow dependences to subsequent (hexagonal) loads,

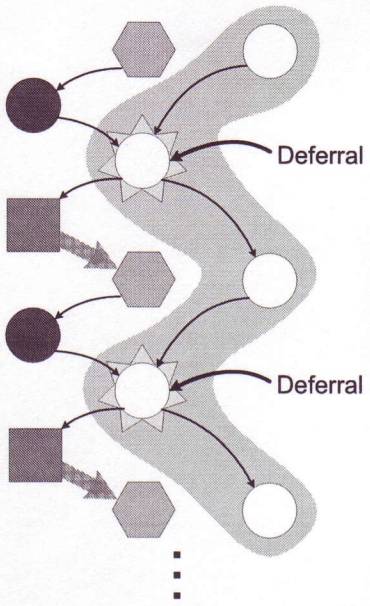


Figure 3.5 Reduction in *gap*.

respectively. These dependences reflect an inability of the compiler's pointer analysis to resolve completely the store and load addresses. In the compiled code (assuming explicit data speculation is not applied) the store-load dependences serialize the long latencies of the multiplies, resulting in a dramatic loss performance in an in-order microarchitecture compared to an out-of-order microarchitecture that does dynamic data speculation. Flea-flicker multiple-pass pipelining's compiler extension determines that this loop's recurrence is a reduction, one whose operations should be allowed to defer to capitalize on latency tolerance among the prerecurrence instructions.

During preexecution, since the indicated operations are deferred, rather than stalling, subsequent loads and multiplies can be preexecuted while the previous multiply instruction completes. This involves the implicit data speculation (mentioned in Sections 2.2.4

and 2.3.6) because the subsequent load is preexecuted before the store. Here, the compiler's recurrence-based critical instruction classification and the microarchitecture's deferral scheme work together to allow flea-flicker multiple-pass pipelining to achieve some of the benefit that an out-of-order model achieves by overlapping the long-latency multiplies.

3.3 Critical Instruction Identification Algorithm

This section describes the specific algorithm used by the compiler to assign the "audible" critical instruction bits. The basic philosophy behind this technique was described in Section 3.1. The approach is to identify recurrences and determine whether its "prerecurrence" producers or its "postrecurrence" dependent consumers have the most opportunity for hiding latency through the multiple-pass pipeline mechanism. If postrecurrence instructions have the majority of such opportunity, the instructions that compose the recurrence are marked as critical.

3.3.1 Algorithm detail

Algorithm 1 shows the specific compiler algorithm for finding and marking critical instructions. This algorithm operates on a data-flow graph [45] built for each function. Cross-iteration dependences a data-flow graph result from loop recurrences, the focus of the approach to critical instruction identification. Such dependences create strongly connected components (SCCs) of the data-flow graph made up of nodes corresponding to

Algorithm 1 Assign *audible* bits.

```
1: DFG G_DFS_node_number = 0
2: for all Node N ∈ G do
3:    DFS_number(N, G)
4: end for
5: for i = (G_DFS_node_number − 1)...0 do
6:    Set SCC = ∅
7:    SCC_selfloop = 0
8:    Reverse_DFS_find_SCC(N_DFS_number==i, CC)
9:    if (SCC_size > 1)||(SCC_selfloop == 1) then
10:      G_SCC_set ← SCC
11:   end if
12: end for
13: for all Set SCC ∈ G_SCC_set do
14:   if WEIGHT(Pred(SCC)) << WEIGHT(Succ(SCC)) then
15:      for all Node N ∈ SCC do
16:         Mark N_audible
17:      end for
18:   end if
19: end for
```

the instructions that comprise the recurrence. Thus, the critical instruction identification algorithm operates on SCCs.

Algorithm 1 consists of three steps: numbering each node according to a depth-first postorder [46] traversal of the data-flow graph, using this order to find the SCCs of the graph that make up the units of critical instruction identification, and determining for each SCC if its constituent instructions should be considered critical.

A counter, $G_{DFS\_node\_number}$, to be used for postorder numbering is first initialized in Algorithm 1. Next, in a loop over each node N in the data-flow graph G initiates a depth-first search at N by performing DFS_number() (Algorithm 2) on that node in the graph. The nodes in the graph are not assumed to be processed in this loop in any particular order. However, since this step performs a postorder numbering, when

complete, this ordering represents a reverse topological sorting of the nodes, as (ignoring cycles) successors of any node N in the graph will have an earlier ordering than N.

Algorithm 2 Depth-first search numbering.

```
1: if Node N_visited not yet visited then
2:    Mark N_visited
3:    for all Node S ∈ Succ(N) do
4:       if S_visited not yet visited then
5:          DFS_number(S, G)
6:       end if
7:    end for
8:    Node N_DFS_number = G_DFS_node_number
9:    P_DFS_node_number = G_DFS_node_number + 1
10: end if
```

In a second loop over all of the the nodes, the set of all strongly connected components in the data-flow graph, G_{CC_set}, is formed. During each iteration, the set of nodes that will comprise the current SCC is initialized, and Reverse_DFS_find_CC() (Algorithm 3) is performed on each node to find the SCC containing that node. The resultant set is discarded if it contains only one node (the current iteration's node N) and if that node is not part of a single-node loop in the data-flow graph. Each SCC found, is then added to the graph's set of SCCs, G_{CC_set}.

Algorithm 3 Reverse depth-first search to find connected components.

```
1: if Node N_reverse_visited not yet visited then
2:    Mark N_reverse_visited
3:    for all Node S ∈ Pred(N) do
4:       if S == N then
5:          SCC_selfloop = 1
6:       else if S_reverse_visited not yet visited then
7:          Reverse_DFS_find_SCC(S, SCC)
8:       end if
9:    end for
10:   Set SCC ← N
11: end if
```

The last step in Algorithm 1 is to determine for each SCC if the instructions making up the SCC should be considered critical. The weight of opportunity, computed through Algorithm 4, for the set data-flow predecessor nodes of the nodes in the SCC (Pred(SCC)), is compared to that of the data-flow successor nodes of the nodes in the SCC (Succ(SCC)). If the successors, i.e. the postrecurrence instructions vastly outweighs the predecessors, the SCC is heuristically determined to be critical. For the results in this work, if the weight of the successors was 10× that of the predecessors, the SCC was judged to be critical.

Algorithm 4 Compute weight of set of nodes.

```
1: weight = 0
2: for all Node N ∈ Set SCC do
3:    if N ∈ interesting then
4:       weight = weight + N_profile_weight
5:    end if
6: end for
```

Postorder numbering of data-flow graph nodes

Algorithm 2 details the process of performing the postordering of nodes. If a node has already been visited (and thus is already numbered) it does not need to be considered. Otherwise, a recursive depth-first traversal from that node is performed. Each of the successors of the current node are visited unless it has already been numbered or has already been visited as part of the current traversal. Each node N is numbered based on the current value of $G_{DFS\_node\_number}$ when the traversal through N has terminated. Thus,

Table 3.1 Interesting instructions for critical instruction identification.

Load
Integer Multiply
Integer Divide
Integer Remainder
Floating Point Arithmetic

the result is a graph with each node assigned a postorder numbering. The node numberings provides a reverse topological sort of the nodes. For acyclic graphs, predecessors in the graph will have a higher numbering than their successors.

Algorithm 3 performs a reverse-depth-first (or a height-first) traversal of the graph. This traversal is performed within Algorithm 1 starting with the highest numbered nodes, thus starting with the nodes at the top of the topologically sorted graph. Since the numbering provides a sorting of the nodes, nodes with untraversed predecessors (or nodes that are there own predecessor in the graph) are part of a cycle in the graph and thus an SCC.

Algorithm 4 shows the process for computing the criticality weight of a set of nodes, SCC. This algorithm is performed on both the predecessors and the successors of every SCC for determining if it is a critical SCC. The weight is computed as the simple sum of the compile-time control-flow profile of each of the interesting nodes in the set. A node is interesting if it represents an instruction that has potential for latency deferral. These instruction types are listed in Table 3.1.

3.3.2 Critical instruction identification algorithm example 1

Figure 3.6 shows an example of the application of the algorithm presented in Section 3.3.1. A data-flow graph for a seven-instruction function is shown. Dark nodes in the graph represent load instructions, while the remaining nodes represent other instructions not of the types specified as interesting for the purpose of this algorithm in Table 3.1. The numbering for is node is shown as that node's DFS_{number}, with the deepest nodes numbered first through Algorithm 2. The profile weights, generated through compile-time profiling, for each instruction in the function are shown within their corresponding node. Using Algorithm 3, node 7 is first considered. Since it has no predecessors, it is rejected for consideration as part of an SCC. During processing of node 6, the SCC in the graph is found through a traversal through 6's predecessors. The SCC in the graph, composed of four nodes, is highlighted. This SCC represents a cross-iteration dependence within a loop in the represented function. Nodes 2, 3, 5, and 6 are not reconsidered since they are already part of a discovered SCC, and nodes 1 and 4 are rejected because their only predecessors are nodes already traversed as part of the discovered SCC.

The "interesting" predecessors and successors of the nodes making up the SCC are weighed using Algorithm 4 to determine whether the SCC should be considered critical. The interesting predecessor, node 7, has a profile weight of 100, while the interesting successor, node 4, has a weight of 10 000. Since the weight of successors (10 000) outweighs that of predecessors (100), the SCC is determined to be critical.

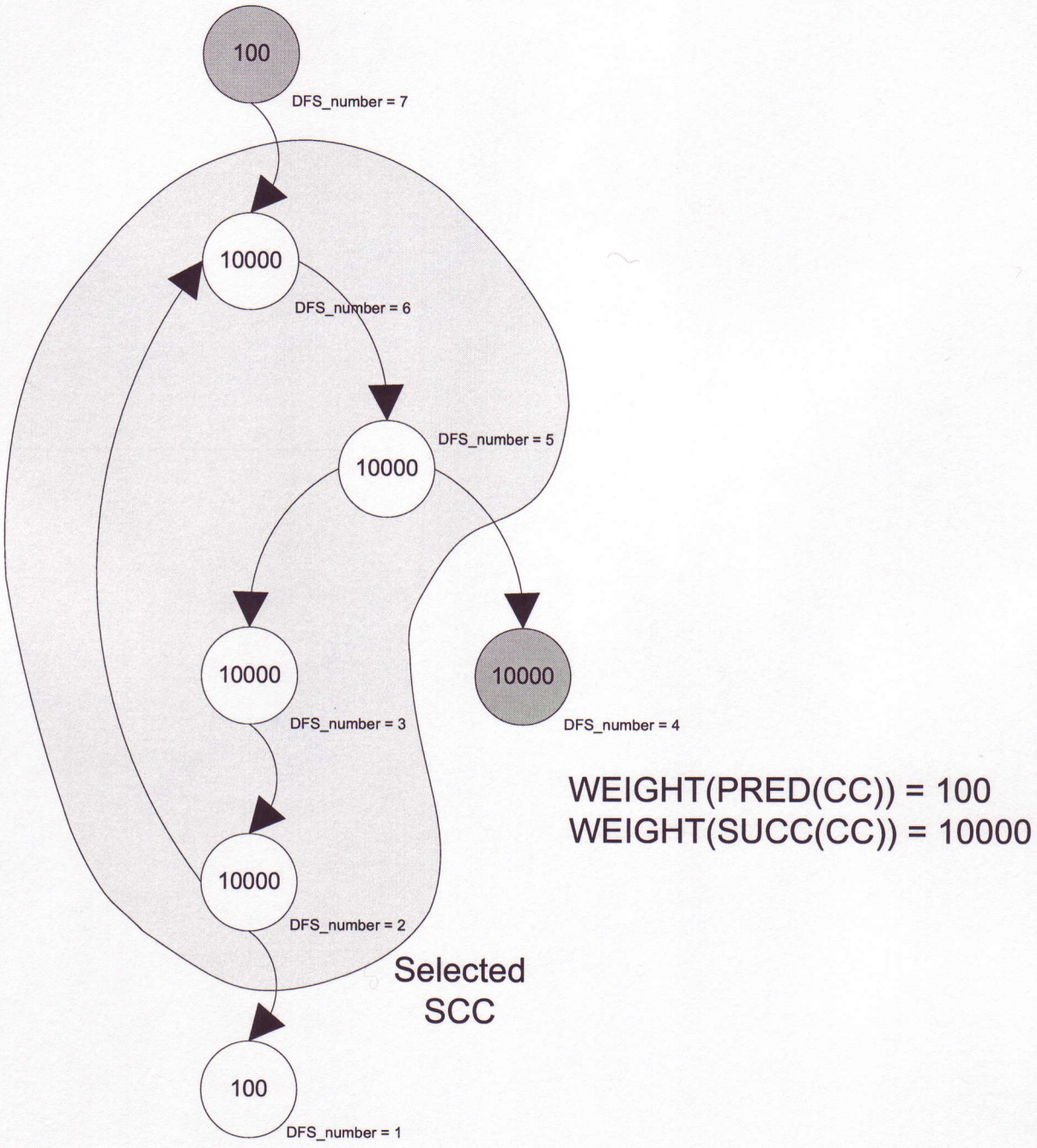


Figure 3.6 Calculation of connected component weights for "audible" hint assignment.

In this example it is undesirable to deferring instructions that comprise the SCC when their operands are not ready (in this example because of a data-cache-miss of the load instruction represented by node 1). Deferring the instructions in the SCC due to a data-cache-miss of the infrequently executed load instruction 1, will cause the deferral, in each iteration of the represented loop, of the heavily-profile-weighted load 4.

3.3.3 Critical instruction identification algorithm example 2

Figure 3.7, shows another example, similar to that in Section 3.3.2 except that for this example the proposed algorithm may identify an SCC as critical in a way that is suboptimal. In this case, an infrequently iterated loop is nested within a heavily iterated outer loop. This loop nesting is shown in Figure 3.7. The outer loop iterates 10 000 times, while the inner loop iterates only an average of 1.1 times (for example, it could have one iteration 90% of the time and two iterations 10% of the time).

The SCC contained in the inner loop is shown with the four selected nodes. Because the profile weight of the important node in the inner loop is higher than the interesting node in the outer loop, the SCC successor weight is larger than the predecessor weight in this example, and the SCC could be determined to be critical. Requiring a large magnitude of difference between the weight of successors and predecessors would prevent this example SCC from being declared critical, but given different profile weightings this problem can occur no matter how great a ratio is required. In this case, marking the recurrence in the inner loop as critical would prevent the deferral of the (infrequently

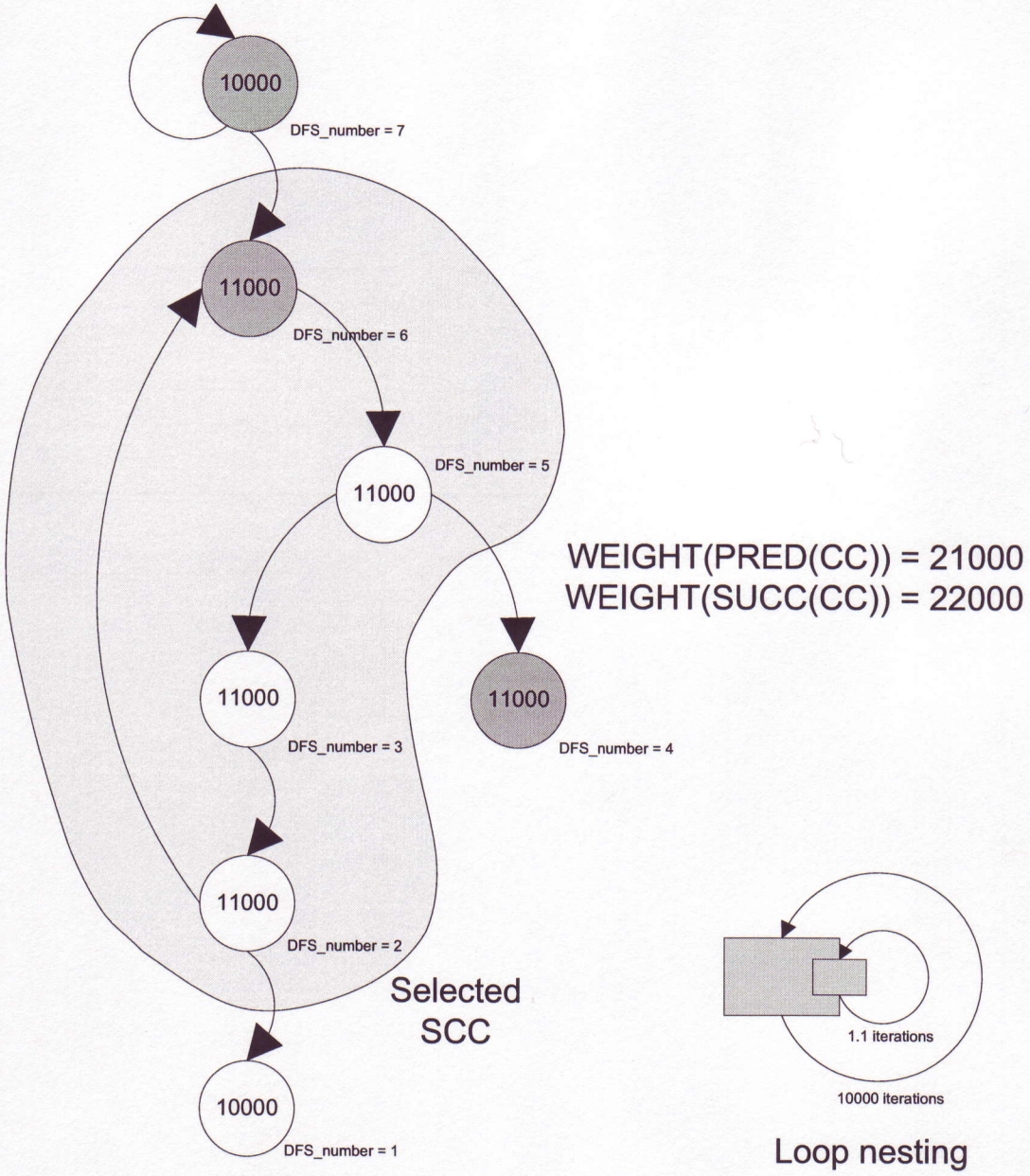


Figure 3.7 Limitation in "audible" assignment heuristic in handling of outer loops.

iterating) inner loop and could prevent the advance execution of subsequent (frequently iterating) outer loop iterations.

There are other cases where Algorithm 1 may not mark instructions as critical in a way that is also suboptimal. For example, the presented assignment algorithm focuses only on SCCs, acyclic subgraphs are ignored, while it may be preferable to have nodes at the base of a graph tree marked as critical if its deferral will lead to the deferral of many interesting instructions. However, as shown in Chapter 4, the presented heuristic allows the compiler to control the multiple-pass pipelining in a way that provides significant performance.

4 EXPERIMENTAL RESULTS

A number of experiments were conducted to test the effectiveness of multiple-pass pipelining. While the technique is applicable across in-order microarchitectures, an EPIC platform, based loosely on the Itanium 2 architecture was chosen for these studies.

4.1 Evaluation Setup

4.1.1 Benchmarks

Listed in Table 4.1 are the applications used to test the performance of multiple-pass pipelining. They represent a wide variety of application types selected from SPECint2000 [15]. Each application was compiled through the IMPACT ILP Compiler (Internal Version 12-13-2004) [16, 47, 48] using the SPEC-distributed training inputs to generate basic block profile information. Interprocedural points-to analysis [49, 50] was used to determine independence of load and store instructions enabling aggressive code reordering during optimizations. Optimizations performed include aggressive inlining, hyperblock formation, control speculation, modulo scheduling, and acyclic intra-

Table 4.1 Description of benchmarks used to evaluated multiple-pass pipelining.

Benchmark	Description
bzip2	Compression
gap	Group Theory, Interpreter
gzip	Compression
mcf	Combinatorial Optimization
parser	Word Processing
twolf	Place and Route Simulator
vortex	Object-oriented Database
vpr	FPGA Circuit Placement and Routing

hyperblock instruction scheduling [16]. Results reflect rigorously sampled [51] complete runs of SPEC reference inputs.

4.1.2 Compilation

The IMPACT research ILP compiler built each of the applications from native C code in order to provide a solid, aggressive code base for which to examine the performance of the multiple-pass pipelining approaches. The specific steps in the compilation process are as follows. Basic block control-flow profiling was performed to guide cross-file function inlining. After inlining, aggressive interprocedural pointer aliasing analysis generates memory dependence information used throughout the compilation process. A variety of architecture-independent optimizations were then applied including classical optimizations, hyperblock predicated region formation utilizing advanced predicate analysis, superblock formation and optimization, and other ILP-enhancing optimizations. Further predicate-aware optimizations were then performed, followed by two rounds of

instruction scheduling utilizing general control speculation and register allocation. Select portions of the C library, mainly portions of the string and memory manipulation and sorting functions, were also compiled through IMPACT with similar levels of aggressiveness and simulated along the each of the benchmarks.

The performance measurements reported in this work are generated by a custom software simulator, Linterpret, that performs cycle-by-cycle, full-pipeline simulation of each instruction. The simulator fully accounts for the effects of branch prediction, wrong path execution, cache utilization and pollution, varying memory latency, interlocking, and bypassing. As previously mentioned, a subset of the C library calls have been compiled and included with the application code, thus enabling cycle-by-cycle simulation of these codes as well. A number of library calls and system services cannot be compiled through IMPACT with current compiler constraints, and thus the native library codes are called on behalf of the application. Cycles spent in the native calls, therefore, are not accounted for in the simulation results, but generally represent an insignificant portion of overall application execution. Readers are directed to [52, 53] for a full description of the Linterpret simulation environment.

4.1.3 Baseline microarchitectural models

To evaluate the multiple-pass pipelining paradigm, an in-order model, a two-pass model, a multipass model, and an idealized out-of-order model were developed in Linterpret. Table 4.2 shows the relevant machine parameters, which are derived from the

Table 4.2 Experimental machine configuration.

Feature	Parameters
Functional Units	6-issue, Itanium 2 FU distribution
Data model	ILP32 (integer, long, and pointer are 32 bits)
L1I Cache	1 cycle, 16KB, 4-way, 64B lines
L1D Cache	1 cycle, 16KB, 4-way, 64B lines
L2 Cache	5 cycles, 256KB, 8-way, 128B lines
L3 Cache	12 cycles, 3MB, 12-way, 128B lines
Max Outstanding Misses	16
Main Memory	145 cycles
Branch Predictor	1024-entry gshare
Two-pass Coupling Queue	256 entry
Two-pass ALAT	perfect (no capacity conflicts)
Multipass Instruction Queue	256 entry
Out-of-Order Scheduling Window	64 entry
Out-of-Order Reorder Buffer	256 entry
Out-of-Order Scheduling and Renaming Stages	3 additional stages
Out-of-Order Predicated Renaming	ideal

Intel Itanium 2 design. This models an achievable near-term design; a futuristic design with longer cache latencies would further accentuate the demonstrated benefits. For the two-pass model, all nonmemory functional units are replicated in the advance pipeline and load and store ports are arbitrated.

Baseline in-order microarchitecture

The simulated explicitly parallel instruction set was that of the IMPACT EPIC architecture [13] which is similar to that the of the Itanium instruction set without the constraints of template bundling. One significant difference is that the IMPACT EPIC instruction set contain single-instruction long-latency operations rather than the multiple-instruction emulation found in the Itanium instruction set [10].

Out-of-order model

The out-of-order model used for comparison with multiple-pass pipelining was constructed to give an idealized indication of the performance opportunities from dynamically ordering instructions. Some of the performance limiting overheads of out-of-order execution mentioned in Chapter 5 were excluded from the model to demonstrate the relatively ideal performance potential from dynamic scheduling. For one example, because of the multiple stages needed for instruction scheduling and register file read, modern instances of out-of-order execution require speculative wake up and dispatch to allow back-to-back dispatch of a producing and consuming instruction. In the simulated out-of-order execution model, both scheduling and register file read are done in the **REG** stage, eliminating the need for speculative wakeup. Additionally, since predication complicates renaming in an EPIC processor because multiple potential producers can exist for each consuming instruction, an ideal renamer was used, avoiding the performance cost of a realistic implementation as described in like [18]. Finally, unconstrained reordering of loads and stores is performed in the scheduler of our out-of-order model. Since no prediction (oracle or realistic) is used to prevent loads from reordering with stores to unresolved addresses, there is the potential for costly data misspeculation flushes. As later examined in Section 4.2.3, in our presented results, these flushes only seriously impact one benchmark, *vpr*, almost doubling the number of front-end stalls.

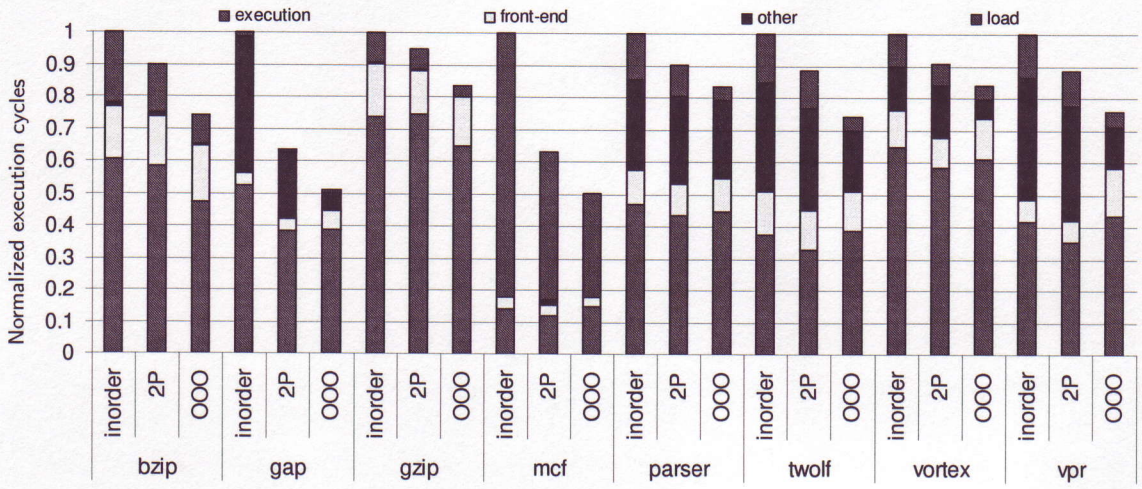


Figure 4.1 Normalized execution cycles; baseline (base), two-pass (2P), and out-of-order (OOO).

4.2 Two-Pass Pipelining Evaluation

Benchmark execution cycle counts are shown in Figure 4.1 for baseline (**base**), two-pass pipelining (**2P**), and out-of-order (**OOO**) configurations, normalized to the number of cycles in the baseline machine. Within each bar, execution cycles are attributed to four categories: *execution*, in which instructions are issuing without delay; *front-end*, stalls including branch misprediction flushes and instruction cache misses; *other*, stalls on multiplies, divides, floating-point arithmetic, and other non-unit-latency instructions; and *load*, stalls on consumption of unready load results. For two-pass pipelining, these are measured in the B-pipe so that the architectural pipeline of the two-pass pipelined system is compared with that of the baseline. Cycles that out-of-order execution does not execute a single instruction are attributed to the cause of the stall of its oldest instruction (or as a front-end stall in the case of an empty instruction queue).

For each benchmark, a significant number of memory stall cycles are eliminated by two-pass pipelining. This improvement results in a reduction in executed cycles for the **2P** system. For example, *mcf* shows a 45% reduction in memory stall cycles and a 37% reduction in overall cycles. On average, two-pass pipelining reduces cache miss stalls by 40% relative to an in-order model. Not all of this reduction, however, results in performance increase. The removal of a load-miss stall may occasionally expose another previously hidden stall. For example, in *vortex*, load stalls are reduced by 32%, while other stalls are almost doubled, resulting in only a 9% net reduction in total cycles. Additionally, a small number of the eliminated load-miss stall cycles have been converted to fruitful execution cycles. The average reduction in total stall cycles (both load and nonload) is 32%, yielding a 1.2× average speedup.

The idealized out-of-order execution model reduces load stall cycles by an average of 62%, and total stall cycles by 60%. The largest magnitude of difference between **OOO** and **2P** load stall benefits occurs in *mcf*. As described in Section 3.2, the loops from *mcf* in Figures 3.2 and 3.3 both contain two tiers of loads that are not part of the recurrence. While **2P** gets significant benefit from the ability to overlap cache misses in loads from the first tier, the miss penalties of the deferred, second-tier loads are serialized in the B-pipe. Out-of-order execution does not share the "single-second-chance" limitation and is free to overlap these misses as well. Where **2P** achieves a 2.1× speedup on the first loop and a 2.2× speedup on the second, **OOO** achieves 3.4× and 2.9× speedups, respectively,

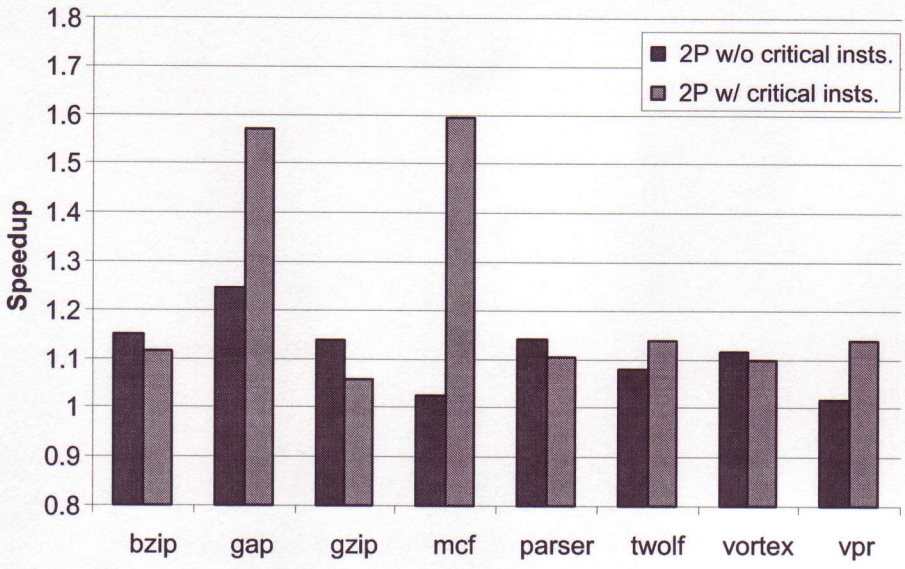


Figure 4.2 Two-pass speedup without and with the use of "audible" hints.

in these experiments. This results in a 1.6× benchmark speedup for **2P** in *mcf* compared to the potential 2× total speedup shown by **OOO**.

The benefit of the compiler-based critical operation identification described in Section 2.2.7 is presented in Figure 4.2. This benefit is clearly seen in *mcf* and *gap*. Without the use of audible hint bits here, **2P** would only achieve a 1.03× speedup over the baseline in-order model on *mcf* and a 1.25× speedup on *gap*. The benchmark *vpr* is the only other benchmark that achieves substantial benefit from use of this technique, in which it increases the speedup from 1.02× to 1.13×. For *bzip2*, *gzip*, *parser*, and *vortex*, stalling the A-pipe on unready critical instructions costs a fraction of two-pass potential performance benefit. In *twolf*, only a slight speedup is seen. In each of these benchmarks, register-dependence recurrences are much less significant and they do not suffer frequently from chained cache misses. It is also important to note that these comparisons

are made against a two-pass model utilizing the update queue. The update queue serves to decrease the amount of deferred execution in the A-pipe as will be explored in Section 4.2.4. Without the update queue, the performance impact of the critical "audible" hints is larger.

Much of the load stall time that is suffered by **OOO** (and by **2P** represents the memory access latency that simply cannot be tolerated through execution of independent instructions. In the example from *mcf* in Figure 3.4, load misses occur on the critical path through the loop, and there are no disjoint cache accesses that can begin early and be overlapped. In this case, **base** performs as well as **OOO** or **2P**. Where **OOO** benefit outstrips that of **2P**, this performance improvement often comes not from the memory tolerance targeted by **2P**, but from motion of instructions in the presence of compile-time scheduling barriers such as potential store to load memory dependences and control dependences [54].

As shown in the example from *gap* in Section 3.2.4, the reordering of potentially conflicting loads and stores can significantly reduce the schedule height and yield substantial performance improvement. While, in the example of Figure 3.5, the previously serialized multiplies can be overlapped in the **2P** model, instructions are processed according to the original schedule and thus only one multiply can start every 4 cycles. Through dynamic scheduling, the **OOO** model can begin two multiplies each cycle (assuming the independent instructions have been fetched into its scheduling window). Thus, while this example highlights its ability to tolerate some nonload stall cycles, it also highlights it

inability to move instructions earlier in the schedule. In another example, an almost identical loop in `SumInt()` from *gap*, a loop performing additions is serialized by false load-store dependences. In this case, there are no opportunities to defer instructions because addition is scheduled for its single cycle latency. Since the A-pipe executes instructions according to the original schedule, it can only defer instructions which are not ready. It cannot execute ready instructions any earlier than their compiler-specified placement, thus limiting the potential for **2P** benefit. In this example, while **OOO** reduces its execution cycles through overlapping several iterations of this loop, the loop executed in the **2P** model exactly as it would in the base.

Another benchmark exhibiting a non-memory-tolerance speedup on **OOO** that is much more significant than its reduction in load-stall cycles is *vpr*. The most significant loop occurs in `try_swap()` for one of the two reference inputs. This loop spans 93 cycles and four separate hyperblocks, the last of which contains 30 cycles of floating point computation that is not consumed by any further iteration of the loop. Trace expansion was performed in the compiler, but replication is limited by code expansion constraints. Because each hyperblock represents the compile-time "scheduling scope," the compiler was not able to hide the latency of this computation in the last hyperblock in the loop. **OOO** is able to overlap this computation with the next iteration of the loop. It is due to opportunities like this that **OOO** achieves a 56% reduction in nonmemory stalls. These nonmemory opportunities are already targeted by proposed mechanisms providing quasi-dynamic scheduling described in Section 6.2. Since these benefits are largely orthogonal,

one of the multiple-pass pipelining techniques could be applied to a quasi-dynamically scheduled system.

4.2.1 Distribution of memory access initiations

Figures 4.3-4.6 show the distribution of the initiation of memory accesses to the A- and B-pipes. Four charts are shown, one for each level of the cache hierarchy. To aid in interpreting this data, Table 4.3 shows, for each benchmark, the proportion of loads satisfied by each level of cache. For every benchmark, the majority of accesses are initiated in the A-pipe, indicating that it is largely successful in preexecuting loads, with a smaller portion of accesses being deferred to the B-pipe. Notable is the significant portion of the L3 cache misses in *mcf* started in the A-pipe. The benefit of overlapping the handling of these cache misses is clearly reflected in the performance improvement for *mcf*. The benchmark *vpr*, on the other hand, executes most of its substantial number of main memory accesses in the B-pipe, and thus displays only a small decrease in data-cache miss stall cycles.

Because all execution in this pipeline is speculative (potentially down a wrong path of execution) there is a risk this speculative execution will result in an increase in the total number of memory accesses made. As can be seen in Figures 4.3-4.6 this increase is, in general, a small percentage. The overall increase in number of loads is 5.4%. However, two-pass execution does result in a significantly increased number of accesses that go all the way to main memory. For example, *gzip* has more than a ninefold increase in its small

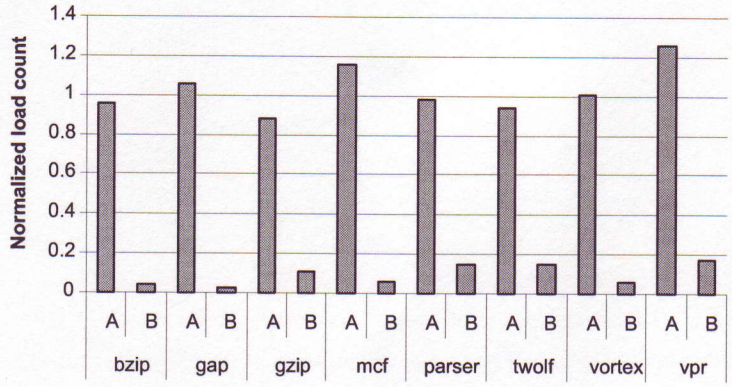


Figure 4.3 Distribution of initiated accesses to A and B pipes: L1 hits.

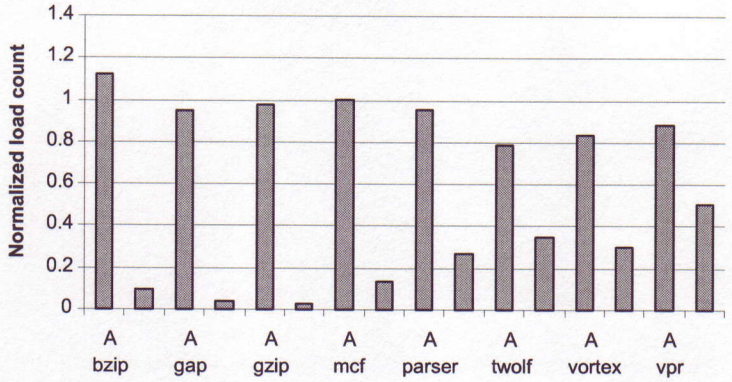


Figure 4.4 Distribution of initiated accesses to A and B pipes: L2 hits.

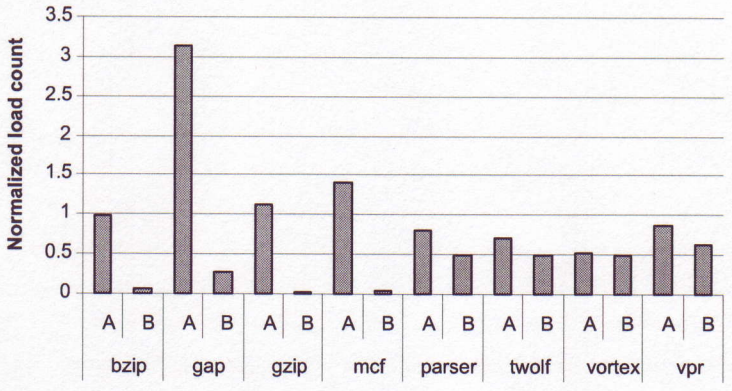


Figure 4.5 Distribution of initiated accesses to A and B pipes: L3 hits.

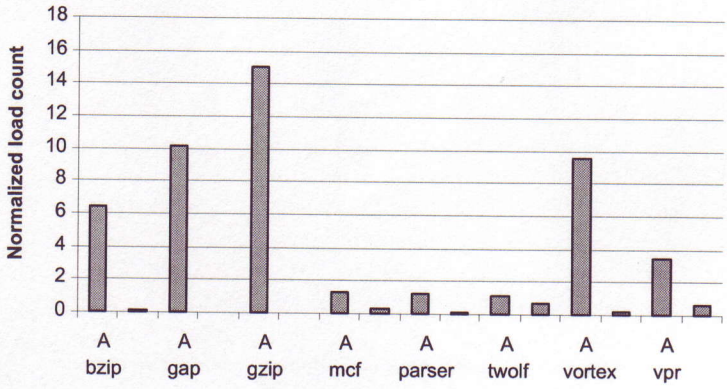


Figure 4.6 Distribution of initiated accesses to A and B pipes: Main memory.

Table 4.3 Distribution of cache access to satisfying level and percentage of load latencies deferrable.

Benchmark	Memory access satisfaction levels L1	L2	L3	MM	Dynamic proportion of noncritical loads
gzip	81.8%	18.1%	0.0%	0.1%	74.7%
vpr	78.2%	15.4%	5.1%	1.3%	84.6%
mcf	57.1%	19.6%	17.8%	1.5%	68.5%
parser	91.1%	7.1%	1.7%	0.1%	90.0%
gap	97.7%	1.8%	0.2&	0.3%	97.5%
vortex	94.0%	4.5%	1.2%	0.3%	99.9%
bzip2	95.5%	2.5%	1.8%	0.2%	91.75%
twolf	82.2%	9.7%	8.1%	0.0%	96.77%

number of main memory accesses. These accesses that are not handled by any level of the cache are likely "wild loads" to invalid addresses, such as dereferencing a pointer off the end of an array or a linked-list traversal. In most cases, while the percentage increase in main memory accesses is significant, such accesses make up only a tiny fraction of all loads (0.1% for *parser*). Therefore, while these accesses are undesirable, they do not have a significant performance impact in our simulated results. Depending on the design of a realistic system, however such misses have the potential to cause performance-impacting translation look-aside buffer (TLB) misses and faults [16].

Table 4.3 also presents the percentage of dynamic loads in each benchmark with a deferrable latency (i.e., the percentage of dynamic loads with no consumers marked critical). Marking too many consumers would keep the majority of load initiation in the A-pipe, but would eliminate opportunities for benefit. The benchmark *mcf* which has the largest improvement from the application of the "audible" hints, is also the benchmark with the lowest percentage of loads whose consumers can be deferred (68.5%). For the remaining benchmarks, more than 75% (and in most cases more than 90%) of the dynamic loads fall into this category. Therefore, the high rate of loads initiated in A-pipe is not simply the result of an over application of these hints.

4.2.2 Modes of benefit

Two modes of benefit from tolerating cache misses with two-pass pipelining have been posited. First, as demonstrated in the examples from *mcf*, long latency memory

instructions that would have otherwise been blocked by preceding stalled instructions can be started early in the two-pass system. This allows multiple long latency loads to be overlapped rather than processed sequentially. The second mode of benefit is that continuing execution beyond the consumer of a delinquent load allows the absorption of short cache misses. Since the code has been scheduled by a compiler assuming hit latencies, loads that miss in the first level of cache are often followed in quick succession by consuming instructions; in two-pass pipelining these instructions can be deferred to the B-pipe, hiding the latency of these misses. Both of these techniques reduce the number of cycles in which the processor reports being stalled on load misses, as demonstrated in Figure 4.1. When an application has poor cache locality, the benefit of overlapping long accesses dominates the benefit of hiding shorter ones (as in *mcf*). For other benchmarks, like *gzip*, there are relatively few long latency misses. The performance gain seen in **2P** for *gzip* is likely due to the second source, the absorption of latencies from short but ubiquitous misses.

4.2.3 Negative performance effects

Other than the potential increase in memory traffic, two-pass execution has the potential to degrade performance in two other situations. First, it extends the effective pipeline length for any misprediction detected in the B-pipe, increasing misprediction recovery cost. In the experimental simulations, an average of 68% of branch mispredictions were discovered and repaired in the A-pipe. The effects of these mispredictions are

less severe than in the single-pipe design, as the B-pipe may continue to process instructions during the redirection of the A-pipe as long as the coupling queue has instructions remaining. Second, store-conflict flushes are incurred whenever a store initiated in the B-pipe conflicts with a programmatically subsequent load that was already initiated in the A-pipe, as discussed in Section 2.2.4. Initiating loads in the A-pipe (even in the presence of deferred, ambiguous stores) is advisable, as 98.9% of all load accesses initiated in the A-pipe while a deferred store is in the queue are free of store conflicts. Overall, only 0.2% of all stores are deferred to the B-pipe and eventually cause a conflict flush.

Taking into account misprediction and store flushes, the results of Figure 4.1 show by the reasonable size of the front-end stall segment that neither substantially erodes the performance gained from two-pass pipelining. In fact, by executing branches that would were positionally blocked by cache-miss stalls, **2P** often sees a reduction in front-end stall time. **OOO** in general sees an even greater reduction in front-end stalls with the notable exception of *vpr*. The more aggressive reordering of instructions in **OOO** resulted in 10 times as many load/store conflict flushes in the **OOO** compared to the **2P** model. This has resulted in more than a doubling of the front-end stall cycles over the baseline for *vpr*. While the idealized **OOO** shows an average decrease in front-end stalls, this would be offset in a realistic implementation by its additional pipeline stages.

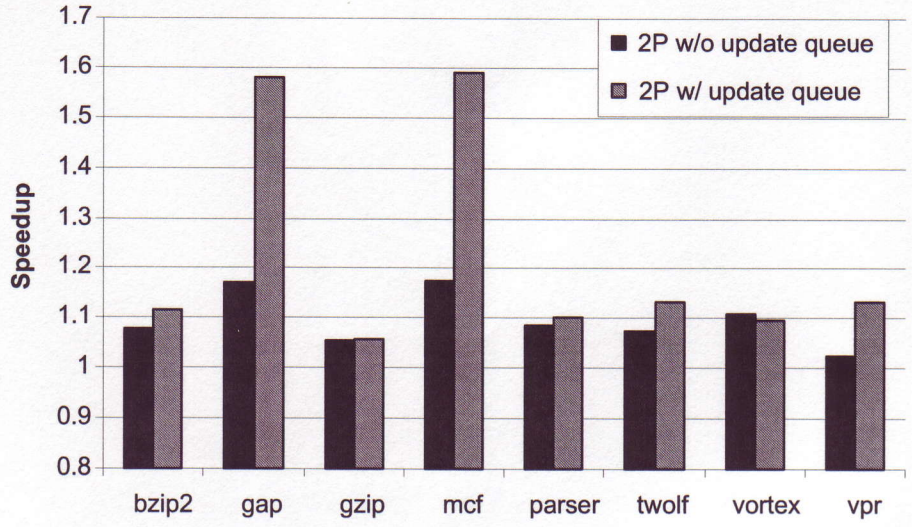


Figure 4.7 Two-pass speedup without and with update queue feedback.

4.2.4 Update queue

Finally, continued successful preexecution might require that committed results in the B-pipe be fed back in a timely manner into the A-pipe to prevent the deferral of ever-greater numbers of instructions. As this interpipe communication may add some degree of complexity, the total effect of this update path was evaluated. The speedup of two-pass pipelining with and without update queue feedback is shown in Figure 4.7. In many cases, the total omission of update had a small impact on performance. The benchmark most impacted by the lack of A-file update was *gap* which saw more than a doubling in cycles when the update queue was removed from the **2P** model. In the case of *gap*, timely updates from B-pipe are needed to maintain sufficient execution in A-pipe to support the overlap of the long-latency multiplies from the `ProdInt()` example.

4.3 Multipass Pipelining Evaluation

Benchmark execution cycle counts are shown in Figure 4.8 for baseline in-order (**in-order**), multipass pipelining (**MP**), and out-of-order (**OOO**) configurations, normalized to the number of cycles in the baseline machine. Within each bar, execution cycles are attributed to the same four categories as in Figure 4.1. During advance mode in multipass pipelining, cycles when no new instruction executions occur (as opposed to merges or deferrals) are attributed to the unsatisfied latency that initiated advance mode. This is analogous to the cycle accounting in the out-of-order execution model where stall cycles are attributed to the cause of the stall of its oldest instruction.

As in the two-pass results presented in Figure 4.1, a significant number of memory stall cycles are eliminated through multipass pipelining for each benchmark. This improvement results in a reduction in executed cycles for the **MP** system. Though the hardware overhead required by multipass pipelining is less than that of two-pass pipelining, this reduction is generally greater than that the **2P** system evaluated in Section 4.2. For example, *mcf* shows a 57% reduction in memory stall cycles (as opposed to a 40% reduction in **2P**) and a 40% reduction in overall stall cycles (as compared to the 32% reduction in **2P**) relative to the in-order baseline. The benchmark *mcf* suffers from frequent cache misses that often occur in dependent chains of misses for which multipass pipelining provides tolerance in cases where two-pass pipelining cannot.

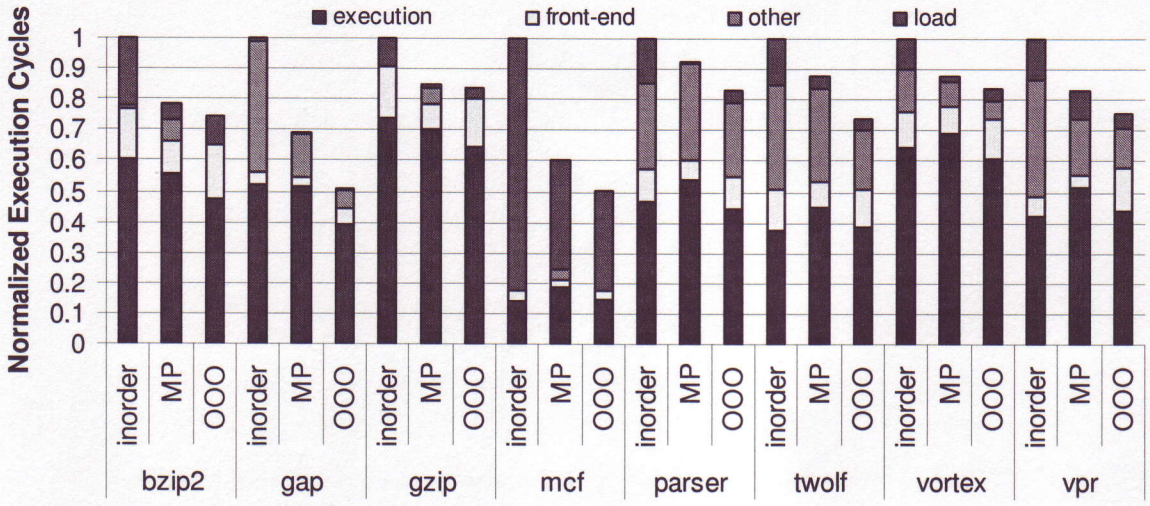


Figure 4.8 Normalized execution cycles; baseline (base), multipass (MP) and out-of-order (OOO).

The average reduction in total stall cycles (both load and nonload) due to application of multipass pipelining is 37%, yielding a 1.27× average speedup. The results in Figure 4.8 demonstrate that even though multipass pipelining avoids the pipeline replication that constitutes two-pass pipelining, the average speedup is more than that of two-pass pipelining. The one benchmark where **2P** significantly outperforms **MP** is in *gap*. There, the dual pipelines of **2P** achieve benefit from concurrently performing preexecution and execution. The fixed, long-latency instructions in *gap* are preexecuted. The dependent reduction is deferred during preexecution but can be simultaneously executed in the execution pipeline. On *gap*, **MP** achieves only a 1.45× speedup, wheres **2P** achieved a 1.58× speedup. Overall, out-of-order execution only achieves a 1.11× speedup over a multipass pipelined system from its ability to find ILP by reordering instruction executions and its more general tolerance of run-time latency.

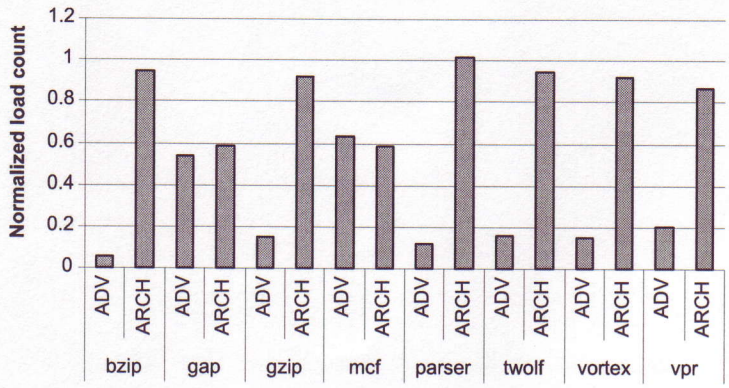


Figure 4.9 Distribution of accesses between advance and architectural modes: L1 hits.

4.3.1 Distribution of memory access initiations

Figures 4.9-4.12 show the distribution of the initiation of memory accesses during advance or architectural mode multipass execution. Four charts are shown, one for each level of the cache hierarchy, similar to Figures 4.3-4.6 for two-pass pipelining. Unlike in two-pass, where the majority of memory accesses are initiated as part of advance execution, advance mode execution only occurs when the processor would be otherwise stalled (a small percentage of total execution for many benchmarks. Similar to the two-pass model, multipass does not result in a large increase in memory accesses handled at each level of cache, other than main memory. As in the case of two-pass pipelining, while an increase in main memory access is seen, this increase is over a very small number of main memory accesses to begin with for most benchmarks. The number of main memory accesses seen with multipass pipelining is similar to that seen in out-of-order execution.

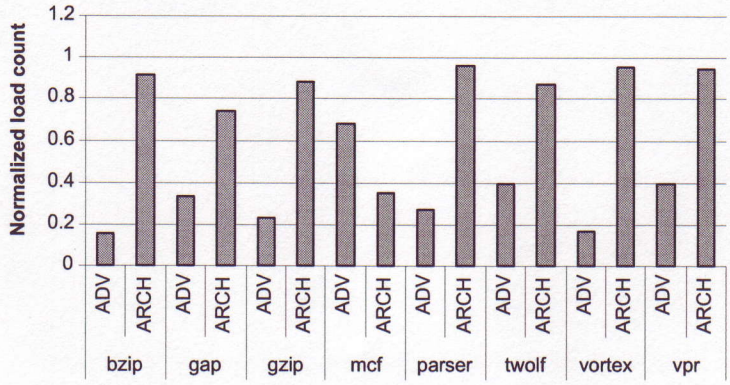


Figure 4.10 Distribution of accesses between advance and architectural modes: L2 hits.

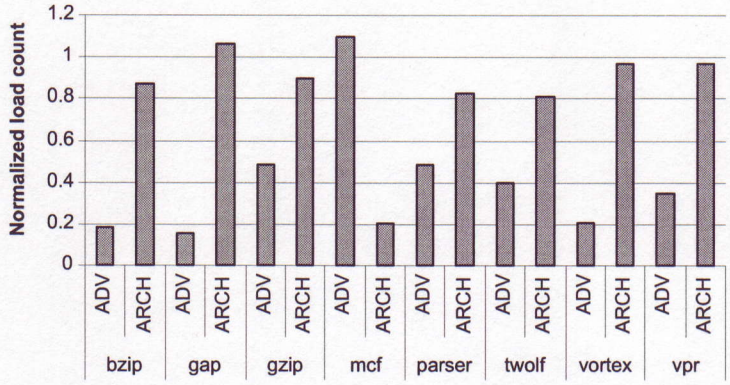


Figure 4.11 Distribution of accesses between advance and architectural modes: L3 hits.

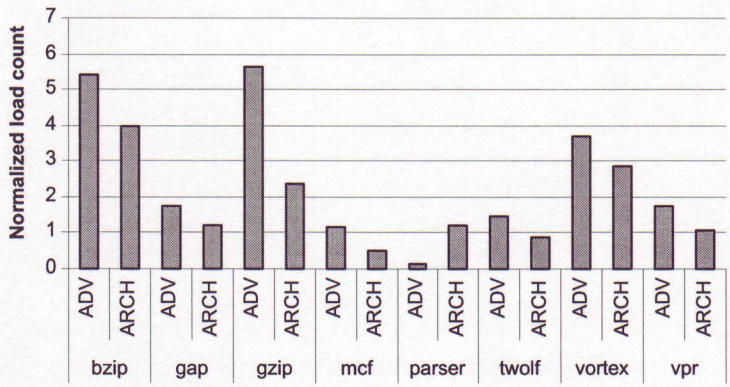


Figure 4.12 Distribution of accesses between advance and architectural modes: Main memory.

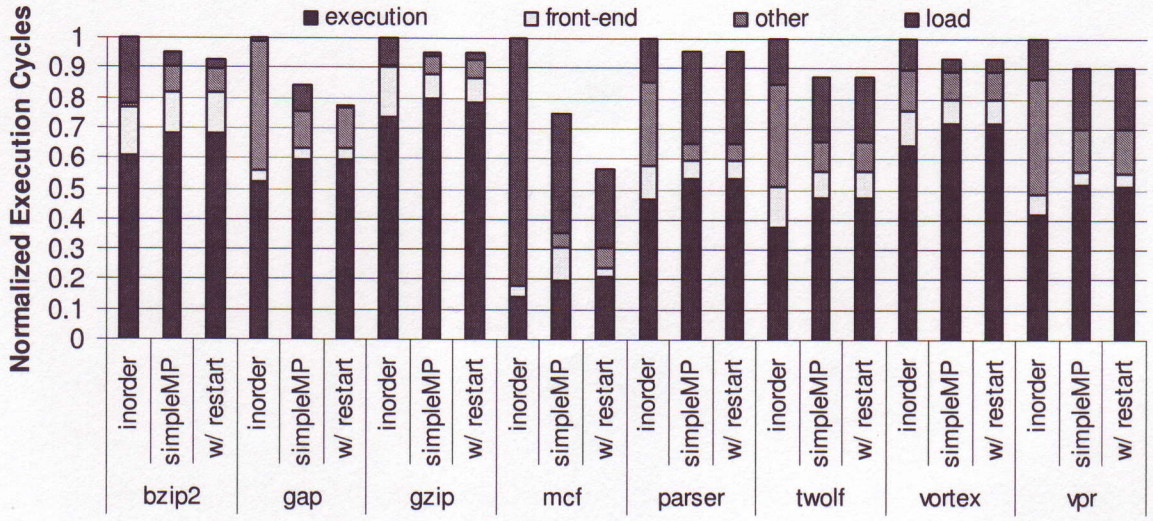


Figure 4.13 Normalized execution cycles; in-order, simple multipass (simpleMP) and simple multipass with critical restart (w/ restart).

4.3.2 Critical instruction restart in multipass pipelining

One of the unique features of multipass execution is the use of the compiler-directed critical instruction hints to direct advance execution. Figure 4.13 shows the performance effect of this critical instruction restart technique. Execution cycles, normalized to the number of cycles executed in the in-order baseline, are presented for a simple multipass implementation without critical instruction restart, instruction regrouping or "sneaking" of peeked instruction, and a similar implementation with the addition of critical instruction restart. The in-order baseline is also shown for comparison. Note that the simple multipass implementation could be considered an extension of the runahead preexecution proposed by Dundas and Mudge [19]. In this multipass implementation execution results are preserved, hiding the latency of long-latency, preexecuted instructions, but these results are not further exploited through instruction regrouping.

As with the two-pass approach's use of critical instructions to control A-pipe, the biggest performance impacts of the critical instruction restart approach are seen for *mcf* and *gap*. For *mcf*, large performance improvements are enabled by critical instruction restart. During advance execution initiated by the consumption of a long-latency cache miss, critical instruction restart improves the performance of *mcf* by providing for the execution of instructions which were earlier deferred but whose operands are now available (independent of the missing load that initiated advance mode). Similarly, critical instruction restart hides most of the load latency stalls in *gap* exposed by the simple multipass technique. To a smaller extent, instruction restart allows the overlap of shorter misses in *bzip2* slightly improving its performance. For the other benchmarks, the performance impact of this technique is minimal. Based on the results of the application of critical instruction bits in Figure 4.2, this is not unexpected as in these benchmarks, the important strongly connected data-flow behavior is most pronounced. However, in *bzip2*, a performance loss was seen under two-pass pipelining from stalling on unready critical instructions. Because of the somewhat different application of critical instructions–restart advance preexecution rather than stalling–the application of critical instruction information is a performance win in *bzip2*. On average, a 1.06× speedup is seen from the application of critical instruction restarting.

4.3.3 Effect of issue regrouping

The last performance impact measured for multipass pipelining is instruction regroup-

ing. Figure 4.14 compares the cycle accounting breakdown for the in-order baseline, the simple multipass implementation as in Section 4.3.2, and the simple implementation with the addition of full instruction regrouping. By forming new issue groups at runtime, a performance improvement is seen on each benchmark. For *mcf*, whose performance is dominated by cache-miss stalls, finding new issue groups exploiting the reuse of pre-computed results makes a minimal performance improvement. Other benchmarks like *bzip2* see more dramatic effects. In these results, instruction regrouping was performed in all three modes of execution, not just in Rally mode, although only in Rally mode (or additional passes of Advance mode) are preexecuted results available for reuse. An average speedup of 1.11× is seen from instruction regrouping.

The full instruction regrouping examined in Figure 4.14 requires a dependence check before issue to determine which instructions (whether preexecuted or not) are ready for issue. This expands the complexity of dependence checking beyond what would normally be done in an EPIC processor. Simple regrouping explained in Section 2.2.8 does not significantly complicate the dependence checking process because only preexecuted instructions are allowed to execute with instructions in earlier issue groups. The input dependences of such instructions can simply be ignored. The slowdown of multipass pipelining with simple regrouping is shown in Figure 4.15. For most benchmarks, this slowdown is within 3%, however, for three benchmarks (each with large performance wins from instruction grouping) the slowdown is closer to 10%.

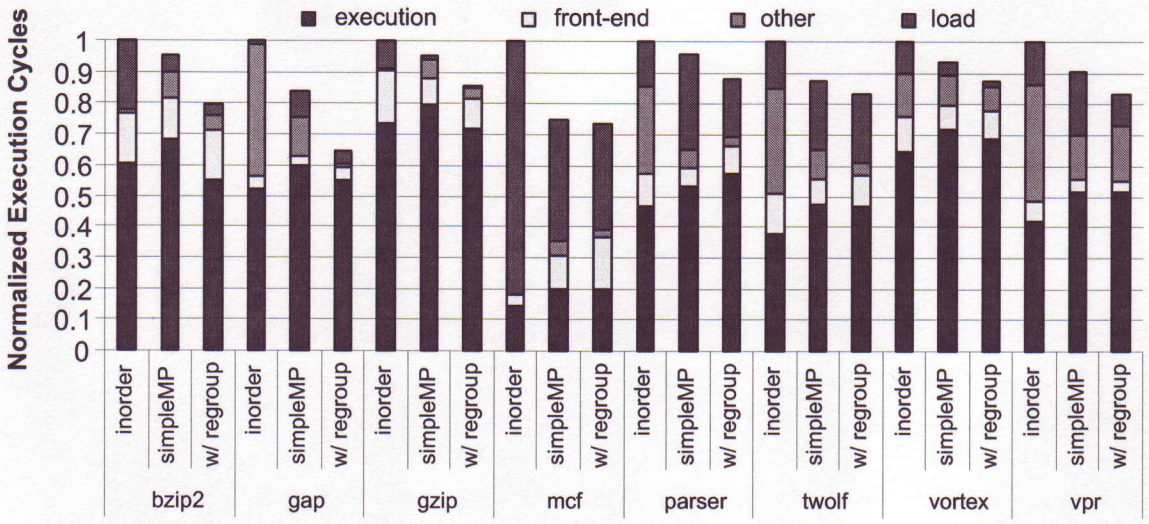


Figure 4.14 Normalized execution cycles; in-order, simple multipass (simpleMP) and simple multipass with instruction regrouping (w/ regroup).

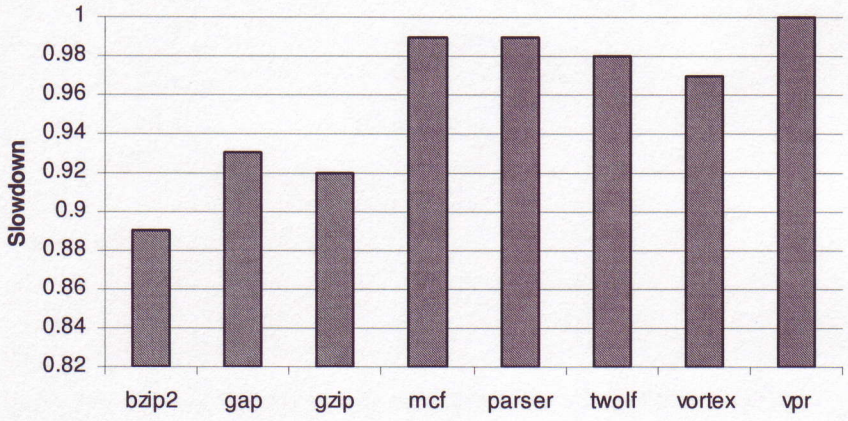


Figure 4.15 Slowdown of multipass pipelined microarchitecture with simple regrouping.

5 IMPLEMENTATION COST

Justifying the overhead of the multiple-pass pipelining implementations requires not only an examination of the performance potential demonstrated in Chapter 2, but also a detailed evaluation of the complexity, power, and area required. Because the more conventional approach to tolerate dynamic memory latency is through out-of-order execution, the complexity of multiple-pass pipelining will be examined relative to the complexity of an out-of-order design. The relative expense of analogous structures and rough estimations for structures unique to the particular designs can provide insight into the relative cost of these implementations.

5.1 The Cost of Out-of-Order Execution

Since the implementation cost of multiple-pass pipelining will be evaluated in terms of the relative cost of out-of-order execution, the implementation cost of out-of-order execution will first be considered. First, the structures central to out-of-order design can be very expensive. For example, in the POWER4 processor, the instruction schedulers and register renaming hardware alone account for more than 10% of total core power and the integer issue queue has the highest power density of any unit [55]. In another

processor, the Alpha 21264 [56], the out-of-order logic consumes 18% of the total power– as much as all of its integer and floating point units combined. While aggressive out-of-order implementations are desired to maximize the tolerance of cache miss latency, power concerns are driving more conservative implementations of out-of-order processors [57].

Decomposing the costs of out-of-order execution, a typical out-of-order implementation consists of three processes: register renaming, dynamic scheduling, and instruction reordering for retirement. To eliminate false dependences created because the reuse of the same architectural register names, register renaming [4] is needed to make dynamic scheduling effective. With dynamic scheduling, the processor itself decides the order of instruction execution, issuing instructions when their data-flow dependences are met. Lastly, to insure that instruction execution affects architectural state in a way consistent with the original program, the results of instruction execution are buffered for incorporation in program order.

5.1.1 Register renaming

In register renaming, the destination register operand of every instruction is assigned a physical register destination. The source operands of subsequent consumers will be renamed so that the correct physical register operand is read by each consumer. While there are multiple implementations of register renaming, each has similar overhead. The out-of-order evaluation in this work assumes an efficient register renaming strategy similar to that of the Pentium 4 [3], in which register values do not have to be moved from their

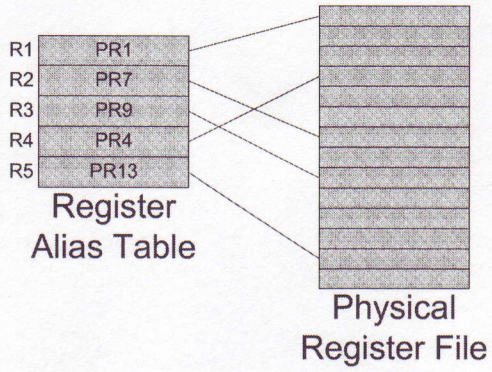


Figure 5.1 Register renaming table.

physical location at retirement. The register renaming structures are shown in Figure 5.1. The register alias table (RAT) is a RAM which stores for every architectural register name the mapping to a physical register name. At register rename time, instruction destinations are assigned a physical register location, and the RAT is updated to reflect the current mapping. In the implementation assumed in this work, a separate RAT is used at the retirement to reflect the register locations that reflect architecturally committed state. The physical register file thus must be large enough to store the result of every in-flight instruction while maintaining the last committed value of every architected register.

Since the RAT is basically a RAM structure, the access time of the RAT is directly related to both its size (which is determined by the number of architected registers) and the number of ports (with is determined by the issue width of the processor) [58]. EPIC processors with large architectural register files and wide-issue thus require large, complex renaming hardware. In addition, predication, an assumed feature of EPIC processors, seriously complicates register renaming [18]. Since at the time of register

renaming it is unknown whether a predicated instruction will write its register destination or whether it will be squashed, it is unknown whether the RAT should be updated to reflect the renaming of that instruction. Implementations of register renaming for predicated architectures thus include significant additional complexity. As mentioned in Chapter 4, the out-of-order performance results in this work assumes prescient renaming of predicated instructions.

5.1.2 Dynamic scheduling

Dynamic scheduling allows the processor itself to determine the order of instruction execution based of the dynamic availability of operands. This provides a very effective mechanism for tolerating cache-miss latency, as instructions that are independent of any cache miss can be dynamically scheduled even if they are later in program order than instructions that are dependent upon the cache miss.

In conventional dynamic scheduling, instructions are buffered while they await their register operands. They are woken up when their operands become available and dynamic scheduling selects woken up instructions for issue to execution units. Figure 5.2 shows the mechanism behind the dynamic wakeup of instructions. As instructions execute, the tag of each renamed register destination is compared with the tag of the source operand of every instruction buffered in the scheduling table. Once both source operands of an instruction are available, the instruction is woken up. Because of the large number of dynamic comparisons required every cycle, wakeup is typically the most complex

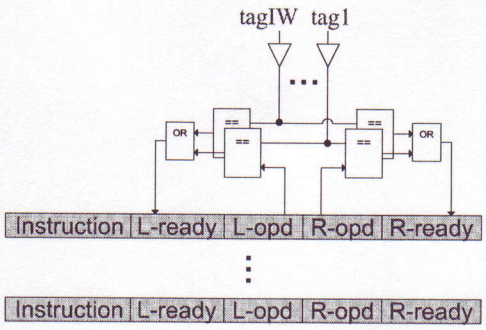


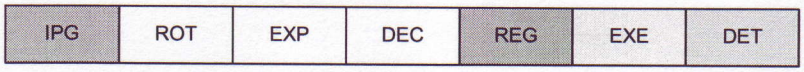
Figure 5.2 Dynamic scheduling wakeup comparisons.

component of out-of-order execution [58]. The number of comparisons is determined by the number of instructions executed each cycle (determined by the issue width) and the size of the scheduling table. The complexity of wakeup limits the practical size of scheduling tables, and this complexity is compounded for wide-issue EPIC processors.

5.1.3 Reorder buffer

Even though the processor reorders instructions dynamically, the effects of these instruction executions need to occur in the programmer specified order. To accomplish this, a reorder buffer maintains the necessary record keeping to commit instruction results to architectural state in program order. Because there are potentially a large number of in-flight instructions in the processor, the reorder buffer maintaining the state of all of these instructions will also be large. In the implementation mentioned above, the overhead of the RAT is also replicated for retirement so that a separate table maps the architectural register names to the location of the physical register for the architecturally committed value for that register name.

In order: Itanium II (7 stages)



Out-of-order equivalent: (10 Itanium II stages)

Figure 5.3 Pipeline comparison between in-order and out-of-order processors.

5.1.4 Pipeline overhead

Figure 5.3 compares the pipeline (up to the branch misprediction resolution stage) of an in-order and an out-of-order pipeline. In the figure, additional stages have been added for register renaming, insertion into the scheduling table and dynamic scheduling. While obviously implementation dependent, the overheads of out-of-order execution typically add an overhead of roughly 30% additional pipeline length [18]. The pipeline of the Pentium 4 [3] shows a proportional pipeline overhead.

5.2 The Cost of Multiple-Pass Pipelining

5.2.1 Overhead of two-pass pipelining

In justifying the use of both an advance and backup pipeline in the two-pass design, is important to consider the contribution of the execution pipeline's cost with respect to the entire processor. The actual execution pipelines of a typical, contemporary microprocessor consumes only a small fraction of the chip's transistor count and power

consumption. For example, the Intel Itanium 2 integer pipeline and integer register file together consume an average of less than 5% of the total chip power and occupy less than 2% of the total chip area [59]. Even when the additional overhead of pipeline control is included, the impact of adding a second register file and an additional pipeline can be a reasonable trade-off for the cache miss tolerance that it provides. Because two-pass pipelining's relatively independent pipelines internally maintain in-order semantics, it has some implementation advantages. For example, for predicated instruction sets, the in-order semantics make two-pass pipelining a more natural implementation that out-of-order execution with register renaming.

Within the replicated pipelines, the doubling of memory units would imply additional data-cache read ports. The nonlinear relationship between cache size and porting makes this an expensive proposition. However, the two-pass pipelining design lends itself readily to partial replication of expensive components. In the implementation assumed in this work, the A-pipe and B-pipe memory units arbitrate for the same port resources. As shown in Chapter 4, the total number of memory accesses is very similar to a single pass approach because, despite two passes, no dynamic memory instruction accesses the cache twice. For this reason, there is only a negligible penalty for arbitrating load ports rather than replicating them. Some subpipelines, such as the floating-point subpipeline, can be fairly large. If necessary, arbitration could also be used there, allowing only partial replication or even complete sharing of any large functional units.

As stated in Section 2.2, the two-pass microarchitecture requires both an advance and an architectural register file. Additionally, each register in the advance file requires valid and speculative bits, and a DynID. In contrast, typical out-of-order designs have a monolithic, physical register file with a size equal to the number of architected registers plus the planned number of in-flight instructions. This capacity could be compared to the size of the coupling result store in addition to the coupling result store which stores results of in-flight instructions awaiting merger into architectural state. Such large capacity is critical to supporting effective renaming and a large instruction window, and may be comparable in number of registers to that used in a two-pass processor. Additionally, if an out-of-order processor utilizes a monolithic register file, such a register file is substantially larger and requires more power than two smaller register files [60, 61].

The storage of the coupling queue (in terms of number of bits) is the same as an identically sized scheduling table, as both store complete instructions for execution processing. However, this queue is likely much less complex than an out-of-order architecture's queue of instructions awaiting scheduling. While the A-pipe to B-pipe coupling structure is a simple FIFO queue, with a single, wide port for adding instructions to the queue and another for reading instructions from the queue, a conventional scheduling table requires a number of read port equal to the processor's issue width and contains comparators for the wakeup process described in Section 5.1.2.

Out-of-order processors require a reorder buffer to support, at a minimum, bookkeeping to enable the reordering of retired instructions into program order. Implementations

like that considered in this work involve an additional file with entries for every physical register (thus corresponding to every in-flight instruction). This file requires two read ports for every instruction issuing in a cycle and one write port for every instruction retiring. The storage in this file could be considered to be matched by additional bookkeeping that might be kept for pre-executed instructions within the coupling result store, however, as described above the coupling result store structure is much simpler. In addition, other reorder buffers implementations involve power-hungry, associatively addressed storage [62].

Another component of two-pass pipelining is the update queue used to feed retired values from the B-pipe back to the A-file. This structure is also a simple FIFO. Furthermore, not only is this feedback path very localized and latency-tolerant, simplifying its design, but, as the results in Chapter 4 indicate, the majority of the benefit from a two-pass design can be achieved without this queue. Because this queue necessitated the DynID, eliminating the updates queue from the design could reduce significant complexity with a nominal degradation in performance. As register file access is often tied closely to cycle time, these DynIDs are the component of two-pass pipelining that is most likely to have cycle-time impact. When eliminated, the simple design of two-pass pipelining is unlikely to affect achievable processor frequency.

Finally, both two-pass and out-of-order designs require hardware to ensure correctness in the presence of load and store reordering. The ALAT hardware is used in two-pass pipelining and is similar in complexity to the content-addressable load and store buffers

Table 5.1 Side-by-side comparison of hardware structures differentiating the multiple-pass pipelining and out-of-order designs.

Two-pass pipelining	Multipass pipelining	Out-of-order execution
Replicated register file (A-File) Coupling result store	Speculative register storage (ARF) Result store	Large monolithic renamed register file
Coupling queue	Instruction queue	Scheduling table Reorder buffer
In-order dependence check	In-order dependence check	Rename-time dependence check
ALAT	Value-based re-execution	LD/ST Ordering Hardware
Replicated back-end pipeline		
Update queue		
		Renaming logic

used by aggressive out-of-order implementations [2] to detect when a load is dynamically reordered with a conflicting store.

5.2.2 Overhead of multipass pipelining

In large part, the implementation overhead of multipass pipelining is a subset of the that of the two-pass pipelining described in the previous section. Multipass pipelining does not require the largest augmentation of two-pass pipelining: the additional execution pipeline. Table 5.1 shows a breakdown comparison between the specific additions of two-pass, multipass and out-of-order designs.

Analogous to the two-pass coupling queue, the multipass instruction queue is a large, but simple, FIFO queue. The multipass queue is slightly more complex in that it it must support peeking of instructions in the queue (with sneak-multipass, simultaneous dequeuing and peeking must be supported), but this queue requires only two read ports for dequeuing and peeking and one write port for enqueueing. It is still significantly

simpler than the general scheduling table required by dynamic scheduling. As described in Section 5.1.2, the scheduling table supports the reading of any general instruction in the table, and thus needs a port for every instruction issued simultaneously in a given cycle. Additionally, instruction dequeuing (or peeking) occurs strictly in-order from the queue, while conventional scheduling tables perform tag comparisons between every register destination generated by execution and every register source of instructions awaiting issue.

Similar to the relationship between the multipass instruction queue and two-pass pipelining's couping queue, the multipass pipelining result store is incrementally more complex than the two-pass coupling result store. Both result stores have a one-to-one relationship with queued instructions. Since the multipass instruction queue can be read from exactly two locations (the DEQ or PEEK pointers into the queue), the result store must also support two read ports (where each port supports reading of a number of operands equal to the issue width).

Finally, the overhead of insuring the proper semantic ordering of memory loads and stores is greatly reduced in the implementations proposed for multipass pipelining. Instead of the content-addressable ALAT structure used in two-pass pipelining, multipass pipelining allows the reordering of loads and stores and verifies that the value loaded by data speculative loads is correct. The results in Chapter 4 demonstrate that front-end stalls are not significantly increased by the pipeline flushes caused by the maintenance

of semantic memory ordering since conflicts between the loads and stores seldom occur in practice.

While the two-pass approach's A-file is a speculative copy of the register file, the advance register file could be implemented in a somewhat different manner. Instructions issuing in the single physical pipeline read either the advance or the architectural register file for each of their operands. Thus, ports can be shared for both the advance and architectural register storage. In the latest Itanium 2 processor, a similar register file with storage for two register values for every architectural register has been implemented to support simultaneous multithreading [42]. Because only the storage is doubled and not the tag decoding, read or write ports, this implementation has only a 15% increase in area and virtually no impact on timing [63]. Such a register file could be similarly used for multipass pipelining.

5.2.3 Critical instruction identification

Another overhead to be considered for both multiple-pass pipelining models is the addition of the "audible" hint bits on each instruction to indicate that the instruction is critical. These hints are similar in spirit to the two-bit hints on load instructions in the Itanium instruction set architecture [11], communicating the desired capability of the loaded data to the hardware. Similarly, Itanium branch instructions contain hint bits that the compiler can use to indicate the desired amount of instruction prefetching to be performed at the target of the branch.

While the critical instruction hints fit well within the EPIC philosophy [8] of compiler-directed execution, instruction sets which have already been defined are unlikely to have unused opcode space for the purposes of these new hints. In an EPIC implementation, templates specify the instruction types of several bundled instructions and the explicit parallelism between them. In the Itanium instruction set there are some unused template specifications that might be usable by the compiler to specific criticality. The compile-time scheduler choosing the bundle template [64] could consider criticality as an additional constraint.

A final alternate implementation of the complier-specified criticality would be through explicit critical operand hint instructions. Rather than marking hint bits on every instructions, hint instructions could be inserted, indicating that its source operands are critical (stalling the two-pass model on an unready operand or restarting advance execution in the multipass model). These instructions would slightly increase code size, but only loads (or other long-latency instructions) would potentially need hints to consume their produced destinations. As shown in Table 4.3, only a small percentage of load instructions have critical destination operands. Additionally, since hint instructions could have multiple source operands this would additionally reduce the number of instructions that would need to be inserted.

Another approach to achieving similar benefits in multiple-pass pipelining is through the use of a heuristic prediction of criticality. In the two-pass approach, a history-based critical instruction predictor like those described in the the related work in Chapter 6

Table 5.2 Instruction executions, merges, and deferrals per cycle.

	bzip2	gzip	gap	mcf	parser	twolf	vortex	vpr
Architecture mode executions	1.34	2.12	0.56	0.26	0.56	0.65	2.06	0.74
Rally mode executions	0.45	0.40	0.71	0.12	0.54	0.53	0.36	0.61
Rally mode merges	0.56	0.42	0.78	0.35	0.26	0.27	0.48	0.55
Rally mode sneak executions	0.85	0.59	0.89	1.10	0.34	0.62	0.38	0.73
Rally mode sneak deferrals	0.39	0.30	0.43	0.46	0.09	0.19	0.36	0.20
Rally mode sneak merges	0.32	0.02	0.17	2.51	0.02	0.08	0.00	0.45
Advance mode executions	0.28	0.22	0.55	0.04	1.01	0.96	0.20	0.99
Advance mode deferrals	0.27	0.16	0.47	0.02	0.54	0.32	0.33	0.54
Advance mode merges	0.02	0.00	0.19	0.00	0.02	0.03	0.00	0.28

could be used to predictively stall the processor before the advance execution was inundated with deferred execution. Such a predictor could similarly be used in the multipass approach. Additionally, in this approach, a simple heuristic based on the number of deferred instructions could be used to initiate advance execution restart.

5.2.4 "Flea-flicker" reprocessing of preexecuted instructions

The final overhead of multiple-pass pipelining to be examined is the efficiency of the reprocessing of preexecuted instructions. Since power is a paramount concern in modern processor design, significant reprocessing effort for preexecuted instructions is undesirable. Figure 5.2 shows a breakdown by execution mode in the multipass pipelined system evaluated in Chapter 4, broken down by benchmark. On average 21% of instruction processing in these benchmarks is the simple merging of persistent preexecution results stored in the result store. Unlike previous preexecution techniques, these dynamic instructions merely commit their precomputed values rather than performing any reexecution.

Likewise, 15% of instruction processing is the advance deferral of unready instructions. Energy wasteful execution can also be avoided for instructions while the invalid condition of their destination registers is propagated. Thus, while the number of times an instruction is processed is greater than that in an in-order processor, a great deal of efficiency savings is achieved by not executing preexecuted or deferred instructions.

6 RELATED WORK

6.1 Runahead Preexecution Approaches

This work is not alone in proposing a mechanism for improving the tolerance of variable-latency instructions. Dundas and Mudge [19, 20] proposed an in-order runahead implementation that relied upon checkpointing and repair. In a single-issue, short-pipeline machine, Dundas examined runahead preexecution in a special mode during any L1 cache miss. When a miss occurred, a runahead mode began. Execution continued without stall in the hope of initiating additional cache misses early. In a manner similar to the multiple-pass pipelining approach, Dundas' runahead design utilized a second register file to store runahead results. When the cache miss as handled, execution returned to the checkpointed state following the cache miss. Since in Dundas' model, runahead preexecution begins when a cache miss occurs, not, as in this work, when a consuming instruction executes, Dundas' mechanism entered runahead unnecessarily in cases where the consumers of a load are scheduled farther away than the load's hit latency. Aside from the memory accesses initiated, this approach discarded all results of runahead execution. (In [20] the preservation of branch outcomes was preserved and shown to slightly

improve performance by partially hiding branch misprediction flushes.) Since runahead work is not preserved, register state is repaired at the end of each runahead effort and instructions are refetched from the instruction immediately following the cache miss.

Mutlu et al. [65] presented an implementation called runahead execution that targets long-latency misses in out-of-order machines. The technique attempts to accommodate such misses efficiently without over-stressing out-of-order instruction scheduling resources. When an out-of-order instruction window is exhausted because of a long-latency cache miss, the oldest instruction in the window is artificially released with an invalid operand, allowing subsequent dependent instructions to be issued, further invalidating their destinations in a way much like invalid nonresults are propagated in the flea-flicker approaches.

The potential for preserving and reusing the correct results produced during out-of-order runahead preexecution was examined in [66]. Only insignificant performance improvements were seen from the preservation of these results. Such preservation was less important than is observed for multiple-pass pipelining, largely because the amount of reuse in this approach was small. This reuse opportunity in [66] was small for two reasons. First, because runahead execution only occurs after the reorder buffer-limited instruction window was full, a small percentage of overall time is spent in runahead. Second, because the instructions preexecuting in this model are many instructions away from the oldest nonretired instruction, there is a reasonably high probability that an intervening unready branch will have been mispredicted, making the preexecuted instructions

off the correct path of execution. Note that the critical restart in multipass pipelining detailed in Section 2.3.6 was proposed in part to increase the amount of successfully preexecuted (and thus reusable) results.

Continual flow pipelines [67] use an approach to out-of-order execution that subsumes runahead execution. Through the use of nonblocking dynamic scheduling similar to that of the Pentium 4 [3] and reorder buffer checkpointing [68], a very large instruction window is achieved with an implementable scheduling table and register file. While continual flow pipelines achieve the large instruction window of runahead approaches while performing only persistent execution, continual flow pipelines require the complexity of dynamic scheduling and register renaming unlike multiple-pass pipelining approaches.

6.2 Thread-Based Approaches

In addition to runahead approach, several thread-based approaches perform preexecution of application code or subset of an application code to achieve similar benefits. Such approaches utilize simultaneous subordinate microthreading [44] (SSMT), sharing resources between execution and preexecution. SSMT adds "microthreads" the sole purpose of which is to help the microarchitecture execute the main thread more efficiently. Idle threads on machines supporting simultaneous multithreading [42] (SMT) are used to perform the preexecution of portion of the running program. This preexecution performs speculative address generation and prefetching and resolves difficult-to-predict branch outcomes [69] to accelerate the main thread. Like the runahead architectures, these

threads can initiate memory accesses early with the goal of reducing the cache stalls of the main thread. Similarly, Zilles and Sohi [70] proposed targeting performance degrading events such as branch mispredictions and cache misses with microthreads by executing speculative program slices [71] that compute the outcome of such problem instructions.

Collins et al. [72] proposed software-based speculative pre-computation and prefetching targeted to particular delinquent loads. Dubois [73] used threads to perform strided software prefetching. Annavaram et al. [74] proposed a dynamic mechanism to generate prefetching microthreads for preexecution. Such techniques have been demonstrated to provide significant performance improvements on real SMT systems [75, 76]. However, these techniques require code generation [77, 78] or dynamic slice extraction for specific delinquent loads and thus cannot address the diffuse serialization of occasional misses that are tolerated through multiple-pass pipelining.

The thread-based approach most similar to two-pass pipelining was proposed by Balasubramonian [79]. In this SMT architecture, available registers were dynamically allocated between the primary program thread and a future thread. The future thread was to examine a larger instruction window and jump far ahead to execute ready instructions. Results were communicated back to the primary thread by warming up the register file, instruction cache, data cache, and an instruction reuse buffer, and by resolving branch mispredictions early.

Master/slave speculative parallelization (MSSP) [80] and Slipstream processors [81] are other approaches with similarities to two-pass pipelining. Both techniques attempt

to exploit additional instruction-level parallelism (ILP) by selecting program threads for preexecution. MSSP assumed a chip multiprocessor system in which one processor executed an approximate version of the program to compute selected program values while additional slave processors checked the execution results by executing the entire program. In slipstream processors, predictably-useless instructions are squashed out of the "advance" stream. As in two-pass pipelining, these approaches partition program execution which the strategy of achieving better parallelism. In these approaches, the leading thread performs persistent program execution; these systems, however, use a much coarser mechanism for partitioning program streams than two-pass pipelining's fine-grained, cycle-by-cycle mechanism. Unlike these thread approaches that attempt to execute all useful work in the leading thread, the flea-flicker technique specifically defers useful computation to avoid stalling the leading, in-order thread on the consumers of load misses.

6.3 Quasi-Dynamic Scheduling

Another approach overcoming the limitations of static scheduling is to utilize dynamic trace-based rescheduling. The schedules of these run-time created traces are often better than the compile-time static schedule as they can dynamically adapt to run-time events, in particular the run-time control-flow profile that led to the creation of the dynamic trace. Trace-based dynamic scheduling, or "quasi-dynamic" scheduling maintains advantages over true out-of-order execution in that execution traces are only occasionally

formed and scheduled and this trace formation occurs offline separately from the current, on-line execution. Traces can executed many times on an in-order pipeline between scheduling occasions in contrast to the continual rescheduling done in an out-of-order processor.

One example, dynamic instruction formating (DIF) [82], collects instructions executed along frequent paths and dynamically bundles them into sets of independent parallel instructions. The sets are then formed into atomically executed groups and stored in a special hardware DIF cache.

Merten et al. [53, 83] proposed a run-time optimization architecture (ROAR) serving a similar purpose, but supporting persistent, arbitrarily long traces. ROAR generated scheduled and optimized traces that composed a run-time detected program phase or "hot spot." This mechanism stored persistent traces in memory for each program hot spot to avoid rescheduling the same dynamic trace. This mechanism exploited the phase behavior of programs, choosing optimized and scheduled traces based on the current program phase.

Another proposed system, the rePLay framework [84], provides a microarchitecture in which instructions are collected into much longer traces and optimized and scheduled by hardware. In this system, an enhanced trace cache [85] delivers units of execution called *frames* to the processor core. Each frame consists of a long sequence of instructions selected such that there is high probability of executing through to the end of the trace. Thus, instruction reordering can be performed without regard to side-exits from the

trace. A checkpoint of architectural state prior to a frame's execution is kept, since a frame must completely execute in order to commit any changes to register or memory state. The rePLay framework detects and forms new frames much more frequently than ROAR, in part because rePLay does not persistently store generated traces but rather keeps them in a cache. Because of the larger amount of trace formation compared with ROAR, a greater percentage of dynamic execution occurred within rePLay's traces.

A further approach using rePLay allowed overlap of two traces (frames) in the processor issue queue, increasing the amount of exposed ILP [86]. Further breaking with an underlying in-order model, this issue queue was basically a complexity-effective dynamic scheduler [58].

Transmeta's Crusoe processors [87, 88] and IBM's BOA [89] use a software approach to quasi-dynamic scheduling. These systems translate, profile, schedule and optimize programs at run time. Software profiling monitors execution and scheduled traces of frequently executed blocks are formed.

While these quasi-scheduling approaches capture some of the benefit of out-of-order execution, their scheduling benefit focuses only on exploiting the dynamic profile. Dynamic scheduling in out-of-order processors, apart from the tolerance of cache misses, is significant [54], but falls short of the more important benefits of tolerating data-cache miss latency. Addressing this problem, prefetching has been performed within run-time optimization systems [90], but like compiler-directed prefetching such techniques only effectively address predictable, long-latency cache misses.

6.4 Critical Instructions

Previous work has examined the utilization of criticality information about individual instructions. Because these critical instructions were used in somewhat different manners, the definition of instruction criticality in these works differ from that established in Chapter 3. The definition most similar to that of this work cite was that of Srinivasen [91] who identified critical loads as those which precede in data-flow cache-missing loads, mispredicting branches or the majority of subsequent execution. The performance of applications was demonstrated to be highly dependent upon the miss behavior of these critical loads.

A more frequent definition of instruction criticality refers to instructions that are on the data-flow critical path. Focusing out-of-order processor resources on these critical instructions has been shown to improve performance [92]. Dynamic approaches have been proposed to predict which instructions are critical under this definition [93, 94]. While a dynamic approach could be used to identify instructions that should be considered critical within a multiple-pass pipelined system, this work demonstrates a static, compiler approach to identify beforehand those instructions that are likely to precede a large number of instructions in dynamic data-flow. This static approach avoids the hardware expense of a criticality predictor.

7 CONCLUSIONS

Because of the disparity between processor logic and memory speed, tolerating cache misses through dynamic scheduling has become almost a ubiquitous characteristic of modern processors. While out-of-program-order execution can tolerate variable memory-instruction latency, it adds hardware components that are problematic for power-conscious design and whose complexity limits the practical ability to reorder instructions. Unfortunately, while compilers have generally proven adept at planning useful static instruction-level parallelism, alternative architectures that rely solely on the compiler's instruction arrangement have been found to suffer because of this dependence when cache misses occur. This work proposes the multiple-pass "flea-flicker" microarchitectural techniques that exploits a static compiler's meticulous scheduling while providing for advance execution beyond otherwise stalled instructions.

The first implementation, two-pass pipelining, was a novel two-pipeline organization designed to carry out useful work during unscheduled latencies while retaining the basic simplicity of an in-order pipeline model. With detailed simulations, an implementation of this technique achieves 70% the total stall cycle reduction of an idealized out-of-order design in an Itanium 2-like EPIC machine. This result is a 1.27× average speedup

over in-order and 1.67× on *mcf* (the SPECint2000 benchmark with the most cache-miss related stalls). This improvement is significant, as real out-of-order implementations for such an EPIC instruction set architecture would be heavily penalized due to features such as predication and an already-large architectural register space, as well as pipeline and power constraints, shrinking the remaining gap between two-pass pipelining and out-of-order execution.

Analysis and examples of SPECint2000 benchmarks illuminated the effects of two-pass pipelining and out-of-order execution in different dependence contexts, showing how each's potential is impacted by miss interdependence. This analysis led to compiler-placed annotations that drastically improved the gain on *mcf*, the most miss-laden application in SPECint2000, by helping the microarchitecture maintain a steady-state opportunity for benefit in important loops. Opportunities exist in the two-pass model both to reduce replication cost and to mitigate the "single-second-chance" phenomenon by applying the flea-flicker paradigm to a single-shared pipeline providing an arbitrary number of virtual "passes."

Multipass pipelining builds upon the initial two-pass pipelining model. In an in-order multipass pipeline, instructions dispatching with unready (because of a cache-miss) operands are deferred, rather than stalled, allowing subsequent instructions to proceed. When deferred instructions finally become ready, these instructions are revisited. Unlike most preexecution schemes, multipass pipelining provides for the persistence of valid advance execution results. Reusing these results increases efficiency and, through a novel

mechanism, accelerates in-order execution. The same accompanying compiler analysis that was useful for two-pass pipelining marks critical instructions, further enhances the handling of miss latencies and reduces fruitless speculative execution by indicating when there is little opportunity for advanced execution. By using only a single physical pipeline, multipass pipelining overcomes many of the impediments of two-pass pipelining, both improving performance and reducing complexity. Excluding the one outlier benchmark, out-of-order execution achieves only a 1.1× speedup over multipass pipelining with significantly more hardware complexity.

Each of the "flea-flicker" multiple-pass pipelining implementations tolerate long-latency (in particular unanticipated data-cache memory latency) without the overhead associated with dynamic scheduling or register renaming. Multiple-pass pipelining techniques preserve the results of correctly preexecuted instructions to improve efficiency, hide the latency of multiple-cycle instructions and accelerate execution by using preexecuted results to break dependencies between instructions and form new issue groups without reordering instructions. Multiple-pass pipelining is an effective implementation of limited dynamic execution (since though instructions are processed in-order, the instructions successfully execute in a dynamic order) that outperforms other preexecution techniques while maintaining the efficiencies of in-order pipelines.

REFERENCES

[1] MIPS Technologies, Inc., *MIPS R10000 Microprocessor User's Manual*, Sept. 1996.

[2] R. E. Kessler, "The Alpha 21264 microprocessor," *IEEE Micro*, vol. 19, pp. 24–36, March/April 1999.

[3] G. Hinton, M. Upton, D. J. Sager, D. Boggs, D. M. Carmean, P. Roussel, T. I. Chappell, T. D. Fletcher, M. S. Milshtein, M. Sprague, S. Samaan, and R. Murray, "A 0.18-μm CMOS IA-32 processor with a 4-GHz integer execution unit," *IEEE Journal of Solid-State Circuits*, vol. 36, pp. 1617–1627, Nov. 2001.

[4] R. M. Keller, "Look-ahead processors," *ACM Computing Surveys*, vol. 7, pp. 177–195, December 1975.

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

[6] J. E. Smith and A. R. Pleszkun, "Implementation of precise interrupts in pipelined processors," in *Proceedings of the 12th Annual International Symposium on Computer Architecture*, June 1985, pp. 36–44.

[7] W. W. Hwu and Y. Patt, "Checkpoint repair for high performance out-of-order execution machines," *IEEE Transaction on Computers*, vol. C-36, pp. 1496–1514, December 1987.

[8] M. Schlansker and B. R. Rau, "EPIC: An architechure for instruction parallel processors," Hewlett-Packard Laboratory, Palo Alto, CA, Tech. Rep. HPL-1999-111, Feb. 2000.

[9] P. H. Wang, H. Wang, J. D. Collins, E. Grochowski, R. M. Kling, and J. P. Shen, "Memory latency-tolerance approaches for Itanium processors: Out-of-order execution vs. speculative precomputation," in *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture*, Feb. 2002, pp. 167–176.

[10] P. Markstein, *IA-64 and Elementary Functions: Speed and Precision.* Hewlett-Packard Professional Books. Upper Saddle River, New Jersey: Prentice Hall PTR, 2000.

[11] Intel Corporation, *Intel Itanium 2 Processor Reference Manual for Software Development and Optimization*, Apr. 2003.

[12] C. McNairy and D. Soltis, "Itanium 2 processor microarchitecture," *IEEE Micro*, vol. 23, pp. 44–55, March 2003.

[13] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B.-C. Cheng, P. R. Eaton, Q. B. Olaniran, and W. W. Hwu, "Integrated predicated and speculative execution in the IMPACT EPIC architecture," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, July 1998, pp. 227–237.

[14] J. Bharadwaj, W. Chen, W. Chuang, G. Hoflehner, K. Menezes, K. Muthukumar, and J. Pierce, "The Intel IA-64 compiler code generator," *IEEE Micro*, vol. 20, pp. 44–53, October 2000.

[15] Standard Performance Evaluation Corporation, "SPEC CINT2000 benchmarks," 1999, http://www.spec.org/cpu2000/CINT2000.

[16] J. W. Sias, S. Zee Ueng, G. A. Kent, I. M. Steiner, E. M. Nystrom, and W. W. Hwu, "Field-testing IMPACT EPIC research results in Itanium 2," in *Proceedings of the 31th Annual International Symposium on Computer Architecture*, July 2004, pp. 26–37.

[17] T. Karkhanis and J. E. Smith, "A day in the life of a data cache miss," in *presented at the 2nd Annual Workshop on Memory Performance Issues*, (Anchorage, Alaska), May 2002.

[18] P. H. Wang, H. Wang, R. M. Kling, K. Ramakrishnan, and J. P. Shen, "Register renaming and scheduling for dynamic execution of predicated code," in *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, Jan. 2001, pp. 15–25.

[19] J. Dundas and T. Mudge, "Improving data cache performance by pre-executing instructions under a cache miss," in *Proceedings of the 11th Annual International Conference on Supercomputing*, June 1997, pp. 66–75.

[20] J. D. Dundas, "Improving processor performance by dynamically pre-processing the instruction stream," Ph.D. dissertation, The University of Michigan, 1998.

[21] W. Y. Chen, S. A. Mahlke, P. P. Chang, and W. W. Hwu, "Data access microarchitectures for superscalar processors with compiler-assisted data prefetching," in *Proceedings of the 24th Annual International Symposium on Microarchitecture*, November 1991, pp. 69–73.

[22] T. C. Mowry, M. S. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," in *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct 1992, pp. 62–73.

[23] R. D. Barnes, R. Chaiken, and D. M. Gilles, "Feedback-directed data cache optimizations for the x86," in *Proceedings of the Second Workshop on Feedback-Directed Optimization*, November 1999, pp. 91–99.

[24] S. Rusu and G. Singer, "The first IA-64 microprocessor," *IEEE Journal of Solid-State Circuits*, vol. 35, pp. 1539–1544, November 2000.

[25] S. Naffziger, G. Colon-Bonet, T. Fletcher, R. Riedlinger, T. J. Sullivan, and T. Grutkowski, "The implementation of the Itanium 2 microprocessor," *IEEE Journal of Solid-State Circuits*, vol. 37, pp. 1448–1460, November 2002.

[26] S. Rusu, J. Stinson, S. Tam, J. Leung, H. Muljono, and B. Cherkauer, "A 1.5-GHz 130-nm Itanium 2 processor with 6-MB on-die L3 cache," *IEEE Journal of Solid-State Circuits*, vol. 38, pp. 1887–1895, November 2003.

[27] J. Chang, S. Rusu, J. Shoemaker, S. Tam, M. Huang, M. Haque, S. Chiu, K. Truong, M. Karim, G. Leong, K. Desai, R. Goe, and S. Kulkarni, "A 0.13μm triple-V_t 9MB third level on-die cache for the 1.7GHz Itanium 2 processor," *IEEE Journal of Solid-State Circuits*, vol. 40, pp. 195–203, Jan. 2005.

[28] Y. Wu, "Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, June 2002, pp. 210–221.

[29] S. Naffziger, B. Stackhouse, and T. Grutkowski, "The implementation of a 2-core multi-threaded Itanium-family processor," in *IEEE International Solid-State Circuits Conference Digest of Technical Papers*, Feb. 2005, pp. 182–183,592.

[30] International Technology Roadmap for Semiconductors, 2003, http://www.public.itrs.net.

[31] B. S. Amrutur and M. A. Horowitz, "Speed and power scaling of SRAMs," *IEEE Journal of Solid State Circuits*, vol. 35, pp. 175–185, Feb. 2000.

[32] B. Chatterjee, M. Sachdev, S. Hsu, R. Krishnamurthy, and S. Borkar, "Effectiveness and scaling trends of leakage control techniques for sub-130nm CMOS technologies," in *Proceedings of the 2003 International Symposium on Low Power Electronics and Design*, Aug. 2003, pp. 122–127.

[33] S. Borkar, T. Karnik, and V. De, "Design and reliability challenges in nanometer technologies," in *Proceedings of the 41st Annual Conference on Design Automation*, June 2004, p. 75.

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

[35] R. D. Barnes, J. W. Sias, E. M. Nystrom, S. J. Patel, N. Navarro, and W. W. Hwu, "Beating in-order stalls with "flea-flicker" two-pass pipelining," *IEEE Transactions on Computers, Special Issue on Embedded Systems, Microarchitecture and Compilation Techniques in Memory of B. Ramakrishna (Bob) Rau*, to be published.

[36] R. D. Barnes and W. W. Hwu, "Multi-pass pipelining," in *Proceedings of the Fourth Workshop on Explicitly Parallel Instruction Computing Architectures and Compiler Technology*, Mar. 2005, http://rogue.colorado.edu/EPIC4/index.html.

[37] H. Sharangpani and K. Arora, "Itanium processor microarchitecture," *IEEE Micro*, vol. 20, pp. 24–43, October 2000.

[38] Sun microsystems, *UltraSPARC IIIi Processor User's Manual*, June 2003.

[39] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. W. Hwu, "Dynamic memory disambiguation using the memory conflict buffer," in *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1994, pp. 183–193.

[40] R. Zahir, J. Ross, D. Morris, and D. Hess, "OS and compiler considerations in the design of the IA-64 Architecture," in *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2000, pp. 213–222.

[41] 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*, July 1987, pp. 18–26.

[42] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995, pp. 392–403.

[43] H. W. Cain and M. H. Lipasti, "Memory ordering: A value-based approach," in *Proceedings of the 31th Annual International Symposium on Computer Architecture*, July 2004, pp. 90–101.

[44] 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*, July 1999, pp. 186–195.

[45] S. S. Muchnick, *Compiler Design and Implementation.* San Francisco, CA: Morgan Kaufmann Publishers Inc., 1997.

[46] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms.* Cambridge, MA: The MIT Press, 1997.

[47] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "IMPACT: An architectural framework for multiple-instruction-issue processors," in *Proceedings of the 18th International Symposium on Computer Architecture*, May 1991, pp. 266–275.

[48] UIUC OpenIMPACT Effort, "The OpenIMPACT IA-64 Compiler," 2005, http://gelato.uiuc.edu/.

[49] B. C. Cheng and W. W. Hwu, "Modular interprocedural pointer analysis using access paths: Design, implementation, and evaluation," in *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, June 2000, pp. 57–68.

[50] E. M. Nystrom, H.-S. Kim, and W. W. Hwu, "Bottom-up and top-down context-sensitive summary-based pointer analysis," in *The Proceedings of the 11th Static Analysis Symposium*, Aug. 2004, pp. 165–180.

[51] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "SMARTS: Accelerating microarchitectural simulation via rigorous statistical sampling," in *Proceedings of the 30th Annual International Symposium on Computer Architecture*, June 2003, pp. 84–95.

[52] R. D. Barnes, "Extracting hardware-detected program phases for post-link optimization," M.S. thesis, University of Illinois at Urbana-Champaign, 2002.

[53] M. C. Merten, "Run-time optimization architecture,", Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2002.

[54] R. D. Barnes, J. W. Sias, E. M. Nystrom, and W. W. Hwu, "EPIC's future: exploring the space between in- and out-of-order," in *Proceedings of the Third Workshop on Explicitly Parallel Instruction Computing Architectures and Compiler Technology*, Mar. 2004, http://www.cgo.org/cgo2004/html/workshops/epic3.html.

[55] P. Bose, D. Brooks, A. Buyuktosunoglu, P. Cook, K. Das, P. Emma, M. Gschwind, H. Jacobson, T. Karkhanis, P. Kudva, S. Schuster, J. E. Smith, V. Srinivasan, and V. Zyuban, "Early-stage definition of LPX: A low-power issue-execute processor," in *Proceedings of the Second International Workshop on Power-Aware Computer Systems*, vol. 2325 of *Lecture Notes in Computer Science*, 2003, pp. 1–17.

[56] M. K. Gowan, L. L. Biro, and D. B. Jackson, "Power considerations in the design of the Alpha 21264 microprocessor," in *Proceedings of the 1998 Design Automation Conference*, June 1998, pp. 726–731.

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

[58] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-effective supersclar processors," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997, pp. 206–218.

[59] E. S. Fetzer, M. Gibson, A. Klein, N. Calick, C. Zhu, E. Busta, and B. Mohammad, "A fully bypassed six-issue integer datapath and register file on the Itanium-2 microprocessor," *IEEE Journal of Solid-State Circuits*, vol. 37, pp. 1433–1440, Nov. 2002.

[60] V. Zyuban and P. Kogge, "The energy complexity of register files," in *Proceedings of the 1998 International Symposium on Low Power Electronics and Design*, August 1998, pp. 305–310.

[61] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger, "Clock rate versus IPC: The end of the road for conventional microprocessors," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, July 2000, pp. 248–259.

[62] D. Ponomarev, G. Kucuk, and K. Ghose, "Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources," in *Proceedings of the 34th Annual International Symposium on Microarchitecture*, Nov. 2001, pp. 90–101.

[63] E. S. Fetzer, L. Wang, and J. Jones, "The multi-threaded, parity-protected 128-word register files on a dual-core Itanium-family processor," in *IEEE International Solid-State Circuits Conference Digest of Technical Papers*, Feb. 2005, pp. 382–383,605.

[64] S.-Z. Ueng, "Template bundling for EPIC architectures," M.S. thesis, University of Illinois at Urbana-Champaign, 2004.

[65] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: An alternative to very large instruction windows for out-of-order processors," in *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture*, Feb. 2003, pp. 129–140.

[66] O. Mutlu, H. Kim, J. Stark, and Y. N. Patt, "On reusing the results of pre-executed instructions in a runahead execution processor," *Computer Architecture Letters*, vol. 4, pp. 1–4, Jan. 2005, www.cs.virginia.edu/~tcca.

[67] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton, "Continual flow pipelines," in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2004, pp. 107–119.

[68] H. Akkary, R. Rajwar, and S. T. Srinivasan, "Checkpoint processing and recovery: Towards scalable large instruction windows processors," in *Proceedings of the 36th International Symposium on Microarchitecture*, Dec. 2003, pp. 423–434.

[69] R. S. Chappel, F. Tseng, A. Yoaz, and Y. N. Patt, "Difficult-path branch prediction using subordinate microthreads," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, May 2002, pp. 307–317.

[70] C. Zilles and G. Sohi, "Execution-based prediction using speculative slices," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, July 2001, pp. 2–13.

[71] C. B. Zilles and G. S. Sohi, "Understanding the backward slices of performance degrading instructions," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000, pp. 172–181.

[72] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen, "Speculative precomputation: Long-range prefetching of delinquent loads," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, July 2001, pp. 14–25.

[73] M. Dubois, "Fighting the memory wall with assisted execution," in *Proceedings of the First Conference on Computing Frontiers*, Apr. 2004, pp. 168–180.

[74] M. Annavaram, J. M. Patel, and E. S. Davidson, "Data prefetching by dependence graph precomputation," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, July 2001, pp. 52–61.

[75] D. Kim, S. Liao, P. Wang, J. del Cuvillo, X. Tian, X. Zou, H. Wang, D. Yeung, M. Girkar, and J. Shen, "Physical experimentation with prefetching helper threads on Intel's hyper-threaded processors," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization*, Mar. 2004, pp. 27–38.

[76] P. H. Wang, J. D. Collins, H. Wang, D. Kim, B. Greene, K.-M. Chan, A. B. Yunus, T. Sych, S. F. Moore, and J. P. Shen, "Helper threads via virtual multithreading on an experimental Itanium 2 processor-based platform," in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2004, pp. 144–155.

[77] S. S. Liao, P. H. Wang, H. Wang, G. Hoflehner, D. Lavery, and J. P. Shen, "Post-pass binary adaptation for software-based speculative precomputation," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, June 2002, pp. 117–128.

[78] D. Kim and D. Yeung, "A study of source-level compiler algorithms for automatic construction of pre-execution code," *ACM Transactions on Computer Systems*, vol. 22, no. 3, pp. 326–379, 2004.

[79] R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi, "Dynamically allocating processor resources between nearby and distant ILP," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, July 2001, pp. 26–38.

[80] C. Zilles and G. Sohi, "Master/slave speculative parallelization," in *Proceedings of the 35th Annual International Symposium on Microarchitecture*, Nov. 2002, pp. 85–96.

[81] Z. Purser, K. Sundaramoorthy, and E. Rotenberg, "A study of slipstream processors," in *Proceedings of the 33rd Annual International Symposium on Microarchitecture*, Nov. 2000, pp. 269–280.

[82] R. Nair and M. E. Hopkins, "Exploiting instruction level parallelism in processors by caching scheduled groups," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997, pp. 13–25.

[83] M. C. Merten, A. R. Trick, R. D. Barnes, E. M. Nystrom, C. N. George, J. C. Gyllenhaal, and W. W. Hwu, "An architectural framework for run-time optimization," *IEEE Transactions on Computer Systems*, vol. 50, pp. 567–589, June 2001.

[84] S. J. Patel and S. S. Lumetta, "rePLay: A hardware framework for dynamic optimization," *IEEE Transactions on Computers*, vol. 50, pp. 590–608, June 2001.

[85] E. Rotenberg, S. Bennett, and J. E. Smith, "Trace cache: A low latency approach to high bandwidth instruction fetching," in *Proceedings of the 29th International Symposium on Microarchitecture*, December 1996, pp. 24–34.

[86] F. Spadini, B. Fahs, S. Patel, and S. S. Lumeta, "Improving quasi-dynamic schedules through region slip," in *Proceedings of the 2003 International Symposium on Code Generation and Optimization*, Mar. 2003, pp. 149–158.

[87] A. Klaiber, "The technology behind Crusoe^{TM} processors," Transmeta Corporation, Tech. Rep., January 2000, http://www.transmeta.com.

[88] M. J. Wing and G. P. D'Souza, "Gated store buffer for an advanced microprocessor," U. S. Patent No. 6,011,908, January 2000.

[89] M. Gschwind, E. Altman, S. Sathaye, P. Ledak, and D. Appenzeller, "Dynamic and transparent binary translation," *IEEE Computer*, vol. 33, pp. 54–59, March 2000.

[90] J. Lu, H. Chen, R. Fu, W.-C. Hsu, B. Othmer, P.-C. Yew, and D.-Y. Chen, "The performance of runtime data cache prefetching in a dynamic optimization system," in *Proceedings of the 36th Annual International Symposium on Microarchitecture*, Dec. 2003, pp. 180–190.

[91] S. Srinivasan, R. Ju, A. R. Lebeck, and C. Wilkerson, "Locality vs. criticality," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, July 2001, pp. 132–143.

[92] B. Fields, S. Rubin, and R. Bodik, "Focusing processor policies via critical-path prediction," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, July 2001, pp. 74–85.

[93] E. Tune, D. Liang, D. M. Tullsen, and B. Calder, "Dynamic prediction of critical path instructions," in *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, Jan. 2001, pp. 185–195.

[94] E. S. Tune, D. M. Tullsen, and B. Calder, "Quanitifying instruction criticality," in *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2002, pp. 104–114.

AUTHOR'S BIOGRAPHY

Ronald D. Barnes, Jr. was born in Tulsa, Oklahoma. He began electrical engineering studies at the University of Oklahoma, where he received his B.S. *summa cum laude* in 1998. Ron continued his studies at the Univerisity of Illinois, working with Professor Wen-mei Hwu in the IMPACT research group since 1998. In 2002, he received his M.S. from the University of Illinois, and he expects to receive his Ph.D. in Electrical Engineering in 2005. As a graduate student, he was the holder of the *Tau Beta Pi* Fellowship, Unversity of Illinois' University Fellowship, the Ernest A. Reid Fellowship in Electrical Engineering and the Rambus Computer Engineering Fellowship. Ron has published at several symposia and workshops on topics including complexity-effective microarchitectures, program-phase detection, and binary and run-time optimization. Ron is a student member of both the IEEE and the ACM. After completion of his Ph.D. degree, Ron will join the faculty of the Department of Electrical and Computer Engineering at George Mason University as an assistant professor.